

Manual del Desarrollador de XMLEye

Antonio García Domínguez
Universidad de Cádiz
nyoescape@gmail.com

20 de marzo de 2008

Copyright © 2008 Antonio García Domínguez.

Legal notice

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

Índice general

Índice general	5
1. Compilación del código fuente	7
1.1. XMLEye: visor XML genérico	7
1.2. ACL2::Procesador: módulo y guión Perl	8
2. Creación de tipos de documentos	11
2.1. Introducción	11
2.2. Creación de un convertidor	11
2.3. Creación de un descriptor de tipo	12
3. Creación de hojas de usuario	15
3.1. Introducción	15
3.2. Diseño del sistema de hojas de usuario	15
3.3. Localización de una hoja de usuario	16
3.4. Herencia a partir de un tipo base	17
3.5. Hojas de preprocesado: particularidades	18
3.6. Hojas de visualización: particularidades	19
Bibliografía	21

ÍNDICE GENERAL

1 Compilación del código fuente

Una nota: aunque hay instantáneas disponibles del código fuente, éstas son más para los usuarios que los desarrolladores. En caso de querer participar como desarrollador, recomiendo encarecidamente usar una copia de trabajo del repositorio Subversion. Bastará con instalar el paquete `subversion` y seguir algunas instrucciones sencillas. Para más información, véase el excelente libro [CSFP07].

1.1. XMLEye: visor XML genérico

Antes de nada, necesitaremos un JDK que implemente J2SE 6.0 o superior, como IcedTea, la herramienta de construcción Ant, y el entorno de pruebas de unidad JUnit (ha de ser una versión 3.X, o no funcionará):

```
sudo aptitude install icedtea-java7-jdk ant junit
```

Ahora tendremos que crear una copia de trabajo local de la última revisión de la rama principal de desarrollo del repositorio de Rediris:

```
svn checkout https://forja.rediris.es/svn/cs12-xmleye/XMLEye/trunk xmleye
```

Ya podemos introducirnos en `xmleye` y aprovechar los objetivos ya definidos en el fichero `build.xml` de Ant: compilar todo el código y ejecutar

clean Limpia el árbol de directorios existente.

compile Compila todo el código fuente.

dist Compila las fuentes ,ejecuta las pruebas de unidad y genera una distribución autocontenida en el subdirectorio `dist`.

dist-jar Tras compilar y ejecutar las pruebas de unidad, genera un fichero `.jar` bajo `dist`, pero no llega a empaquetarlo con todo lo demás.

docs Genera la documentación del API en formato HTML en el subdirectorio `docs` a través de Javadoc.

run Se trata del objetivo por defecto (ejecutado a través de `ant`). Compila el código y ejecuta la versión así compilada de XMLEye.

run-about Compila el código y ejecuta únicamente la ventana de `^cerca de`". Útil a la hora de diseñar la interfaz.

1 Compilación del código fuente

run-find Como la anterior, pero para el diálogo de búsqueda.

run-types Más de lo mismo, pero para el diálogo de edición de tipos.

test Ejecuta las pruebas de unidad. La salida de cada conjunto de pruebas se halla bajo el fichero **TEST-*** correspondiente.

Un par de notas adicionales: esta aplicación depende de InfoNode Tabbed Panel 1.5.0 (licenciado bajo la GPL para uso no comercial) y del look and feel JGoodies Looks 2.1.4 (licenciado bajo BSD), disponibles en <http://www.infonode.net/index.html?itp> y <https://looks.dev.java.net/> respectivamente. De todas formas, los ficheros `.jar` necesarios se hallan en el propio repositorio, por lo que no hay que hacer nada al respecto.

Además, el desarrollo puede hacerse mucho más cómodamente si se emplea el plugin Subclipse para Eclipse, disponible en <http://subclipse.tigris.org/>, y se importa a través de él el proyecto Eclipse desde el repositorio. Así podemos contar con sus funcionalidades de refactorización y notificación de errores y avisos de compilación en directo.

1.2. ACL2::Procesador: módulo y guión Perl

Este módulo se halla escrito usando las prácticas habituales de diseño de paquetes de Perl, por lo que no sorprenderá a nadie con cierta experiencia en la materia. Por supuesto, necesitaremos Perl, y además unos cuantos más paquetes del CPAN. También necesitaremos el paquete de la biblioteca de análisis de XML de GNOME, LibXML:

```
sudo aptitude install perl libxml2-dev
```

Otra opción, si hemos habilitado los repositorios de XMLEye y compañía, es aprovechar la información disponible en el paquete fuente de ACL2::Procesador, `libacl2-procesador-perl`:

```
sudo apt-get build-dep libacl2-procesador-perl
```

A continuación obtendremos una copia local de trabajo del repositorio:

```
svn checkout https://forja.rediris.es/svn/cs12-xmleye/pprocACL2/trunk pprocACL2
```

Nos introducimos en dicho directorio y aprovechamos el `Makefile.PL` para crear el `Makefile` con el que trabajaremos, y asegurarnos de que tenemos todas las dependencias:

```
cd pprocACL2
sudo perl Makefile.PL
sudo chown -R 'id -un'.'id -gn' *
```

1.2 ACL2::Procesador: módulo y guión Perl

La segunda orden era para devolver todos los ficheros creados con permisos de superusuario por el `Makefile.PL` a nosotros. La primera orden usó `sudo` ya que podría tener que instalar módulos del CPAN en nuestro sistema (no debería de tener por qué si usamos los paquetes de desarrollo de XMLEye).

Ya podemos compilar con `make` y lanzar las pruebas de regresión con `make test`.

1 Compilación del código fuente

2 Creación de tipos de documentos

2.1. Introducción

XMLEye, como el nombre indica, se trata de un visor genérico de documentos XML. Pero ello no lo limita a únicamente ficheros XML: si se le proporciona la información necesaria acerca de cómo distinguir un cierto tipo de fichero, y cómo convertirlo a formato XML, podrá abrirlo de forma transparente tal y como abre un fichero XML.

La información referente a cómo identificar y, opcionalmente, qué hacer para convertir un formato determinado se halla en un *descriptor de tipo*. Este descriptor puede incluir una referencia a cualquier ejecutable que haga las veces de convertidor a un documento XML.

En este capítulo veremos qué condiciones debe cumplir un convertidor, y cómo habría que escribir posteriormente el descriptor para integrarlo con XMLEye. En cuanto a su instalación, véase el Manual de Usuario: realmente es tan sencillo como asegurar que el convertidor esté bajo la ruta correcta y copiar el descriptor de tipos a su sitio.

2.2. Creación de un convertidor

Un convertidor puede ser cualquier cosa que podamos ejecutar: desde un programa en código máquina hasta guiones Perl o Python. Yo en particular prefiero usar Perl, ya que se halla mejor ajustado a la tarea de procesar texto, pero cualquier cosa sirve.

Salvado el tema del lenguaje, las restricciones que ha de cumplir nuestro convertidor son:

1. Debe de poder aceptar a través de la línea de órdenes la ruta absoluta del fichero a convertir.
2. Debe de ofrecer el resultado a través de la salida estándar, e imprimir errores por la salida estándar de errores.
3. Ha de comportarse como cualquier ejecutable tipo UNIX, devolviendo el código de estado 0 en caso de éxito y distinto de cero en caso de error.
4. Ha de instalarse dentro de alguno de los directorios en el PATH del usuario que vaya a ejecutar XMLEye. Puede ser algún directorio común a todos los usuarios, como `/usr/bin` o `/usr/local/bin`, o puede ser específico a un usuario.

Sería recomendable que además contara con su propia ayuda, en caso de aceptar más opciones, y su página *man*, pero no es imprescindible.

2 Creación de tipos de documentos

Un ejemplo muy simple de qué podría hacerse puede extraerse del guión `yaml2xml` del módulo `YAXML::Reverse`:

```
#!/usr/bin/perl

use strict;
use warnings;
use YAXML::Reverse qw(ConvertFile);

# Argument processing
if (scalar @ARGV != 1)
    print "Usage: $0 [path to .yaml]\n";
    exit 1;

my ($path) = @ARGV;

# Convert the YAML file
print ConvertFile($path);
exit 0;
```

Como puede verse, no es más que un guión Perl normal y corriente: la primera línea indica con qué intérprete debería ejecutarse al intentar ejecutarse. Es decir, si intentamos hacer esto:

```
./yaml2xml
, realmente haremos esto:
/usr/bin/perl ./yaml2xml
```

A continuación activamos las opciones habituales de comprobación de errores de Perl *strict* y *warnings*, e importamos la función `ConvertFile` del módulo `YAXML::Reverse`, que implementa realmente toda la funcionalidad.

Ofreciendo esta funcionalidad separada en un módulo aparte nos aseguramos de que pueda ser reutilizada en un futuro a través de otros medios.

A continuación procesamos los argumentos, recibiendo la ruta absoluta al fichero `.yaml` que antes mencionamos, y mostrando un mensaje de uso (junto con el código de estado correspondiente) si no se ha proporcionado.

Por último imprimimos el resultado de la conversión por la salida estándar e indicamos mediante el código de estado 0 que todo ha ido bien.

2.3. Creación de un descriptor de tipo

Para decirle a `XMLEye` que acepte un nuevo tipo de fichero, y que se integre con un determinado editor y convertidor, todo lo que hemos de hacer es escribir un corto fichero XML como el siguiente:

2.3 Creación de un descriptor de tipo

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE type SYSTEM "accepted-doc.dtd">
<type xmlns="http://xmleye.uca.es/xmleye/accepted-doc">
  <name>Fichero YAML/JSON</name>
  <name language="en">YAML/JSON file</name>
  <edit_cmd>emacs %s</edit_cmd>
  <import_cmd>yaml2xml %s</import_cmd>
  <extensions>
    <extension>yaml</extension>
    <extension>yml</extension>
    <extension>json</extension>
  </extensions>
</type>
```

Yendo línea a línea por el fichero anterior, nos encontramos con lo siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Se trata de la declaración que todo fichero XML debiera (aunque no tiene por qué) incluir. Indica que estamos empleando la versión 1.0 del estándar: aunque también existe la versión 1.1, las diferencias no nos importan en este caso.

```
<!DOCTYPE type SYSTEM "accepted-doc.dtd">
```

Esta línea indica al analizador XML de XMLEye cómo puede validar el documento. El fichero `accepted-doc.dtd` es parte de XMLEye y se halla preinstalado junto con el descriptor de tipo de documento XML, `xml.type`.

```
<type xmlns="http://xmleye.uca.es/xmleye/accepted-doc">
...
</type>
```

El descriptor de documento viene representado globalmente por un elemento de nombre `type` y del espacio de nombres de descriptores de tipos de XMLEye.

```
<name>Fichero YAML/JSON</name>
<name language="en">YAML/JSON file</name>
```

Estas dos líneas indican el nombre del tipo de documento que se le mostrará al usuario. Un detalle importante es el atributo *language*: esto nos permite localizar la misma descripción a distintos idiomas e incluso dialectos, si así lo deseamos. El algoritmo de búsqueda es bastante sencillo:

1. Sea *P* el código del país (según el estándar ISO-3166) de la localización del usuario, detectada a partir de los ajustes de su sistema operativo, y *L* el código de idioma, según el estándar ISO 639.

2 Creación de tipos de documentos

2. Se busca una entrada cuyo atributo `language` tenga la forma 'L_P', coincidiendo tanto el país como el lenguaje. Esto nos permite, por ejemplo, usar distintas entradas para inglés americano e inglés británico.
3. A continuación se busca solamente por el idioma ('L').
4. Si aun así no hemos conseguido nada, tomaremos el nombre por defecto (aquel sin atributo `language`).

```
<edit_cmd>emacs %s</edit_cmd>  
<import_cmd>yaml2xml %s</import_cmd>
```

Éstas son las órdenes que XMLEye debe de usar para editar el fichero original (y *no* el fichero XML resultado de la conversión), y para convertir el fichero a XML. De hecho, XMLEye invocará al convertidor cada vez que considere que ha habido cambios en el fichero original, si se ha activado la opción `[moreinfo = none]` Actualización Automática.

```
<extensions>  
  <extension>yaml</extension>  
  <extension>yml</extension>  
  <extension>json</extension>  
</extensions>
```

Por último, el elemento `extensiones` incluye una serie de subelementos `extension` que relacionan ciertas extensiones de fichero con el tipo de documento en cuestión. En otros estándares parecidos como la especificación de `shared-mime-info` de `Freedesktop.org` ([Fre08]) implementan también un sistema basado en el contenido del fichero, pero por simplicidad no lo he considerado.

3 Creación de hojas de usuario

3.1. Introducción

A través de los tipos de documento, XMLEye puede efectivamente abrir cualquier tipo de fichero para el que podamos imaginarnos una conversión a XML. Con ello ya disponemos no de un visor genérico de XML, sino de un visor genérico *en* XML.

Sin embargo, no sería tan interesante si sólo pudiéramos ver el resultado de la conversión tal y como queda: es posible que queramos poner icono o una etiqueta amigable a algunos nodos, ocultar otros, añadir nuevos nodos, o simplemente reorganizarlos. Éstas son las tareas de cualquier *hoja de preprocesado*.

Incluso así, seguimos teniendo un árbol XML normal y corriente. ¿Y si quisiéramos ver en detalle toda la información concerniente a un nodo determinado, con hipervínculos a todo aquello que nos pueda ser de interés? Ahí es donde entran en juego las *hojas de visualización*, que crean esos resúmenes en formato XHTML para cada nodo del árbol que sea visible al usuario.

En esencia, estas hojas son ficheros XSLT (eXtensible Stylesheet Transformations), sobre los cuales XMLEye implementa una serie de extensiones, permitiendo su fácil localización y reutilización. Varios ficheros XSLT pueden reunirse en un único ente coherente, que aquí llamaremos *hoja de usuario*.

Para aquellos a los que les preocupe el rendimiento: estas hojas son compiladas a bytecode Java en su primera ejecución, siendo mucho más rápidas en las posteriores transformaciones. Dicha representación se conoce en Xalan con el nombre de *translets*. De hecho, si quisiéramos, podríamos generar versiones empaquetadas de esas hojas y usarlas dentro de la línea de órdenes.

3.2. Diseño del sistema de hojas de usuario

Todas las hojas de usuario empleables por XMLEye se hallan instaladas en lo que llamaremos un repositorio de hojas de usuario. Normalmente este repositorio se halla bajo el subdirectorio `xslt` de donde esté `xmleye.jar`, o en su defecto en el directorio desde el cual se lance XMLEye: en el paquete Debian se trata de `/usr/share/xmleye`, por ejemplo.

Mirando en su interior, veremos tres ficheros XSLT:

`preproc.xsl` Por sencillo que parezca, éste es el punto de entrada a partir del cual siempre se comienza toda transformación de preprocesado. Además de importar la hoja `util.xsl` y definir variables con rutas a iconos predefinidos, se importa una hoja llamada `current_preproc`.

3 Creación de hojas de usuario

Tal hoja no existe en ninguna parte: de hecho, se trata de una de las extensiones propias de XMLEye. Es una URI especial que se resuelve a la ruta de la hoja de preprocesado seleccionada automáticamente por el usuario. Bueno, casi, pero ya lo veremos después.

view.xsl De forma análoga al caso anterior, éste es el punto de entrada para toda visualización en formato XHTML. Además de importar la hoja de utilidades y la hoja de visualización elegida por el usuario, activa el modo de salida en XHTML y, lo que es más importante, inicia la transformación por el nodo que haya seleccionado el usuario (proporcionado mediante el parámetro `selectedUID`).

¿Y de dónde viene este UID? Pues del preprocesado, precisamente. Es una de las pocas cosas que absolutamente toda hoja de preprocesado debe hacer. De todas formas, si hacemos que nuestra hoja de preprocesado herede de la hoja base `xml`, no tendremos que preocuparnos por esto.

Un detalle importante: se puede imaginar uno fácilmente que buscar por todo el documento el nodo que tenga un determinado identificador tiene que ser bastante costoso, y de hecho, lo es. Por ello, esta hoja crea un índice a nivel del documento completo de todos los identificadores y sus nodos correspondientes, con lo que la búsqueda posterior a la primera será mucho más rápida.

util.xsl Esta hoja ya no cumple un papel tan importante: simplemente define algunas funciones de utilidad para la manipulación de cadenas, como `util:to-lower` (cambio a minúsculas), `util:to-upper` (cambio a mayúsculas) o `util:substring-after-last` (que toma la subcadena posterior a la última aparición de la clave, y de lo contrario la cadena completa).

En un futuro puede que estas funciones se hagan redundantes si realizo el cambio a XSLT 2.0, pero las dejaré ahí de todas formas, ya que no creo que hagan daño.

Las hojas de usuario son subdirectorios de `view` (para las hojas de visualización) y `preproc` (para las de preprocesado), y aparecerán bajo XMLEye con el mismo nombre que tenga el subdirectorio.

Toda hoja de usuario tendrá como mínimo un fichero XSLT, pero puede tener varios. De hecho, en el siguiente capítulo veremos por qué nos interesaría hacerlo así.

3.3. Localización de una hoja de usuario

Una preocupación constante durante el diseño de XMLEye ha sido su internacionalización, es decir, la implantación de la infraestructura necesaria para poder localizarlo a distintos lenguajes, atendiendo incluso a la presencia de dialectos distintos entre países.

Las hojas de usuario generan contenidos que el usuario podrá ver, así que no son ninguna excepción. Pero, ¿cómo podemos localizarlas entonces?

La idea es realmente muy simple: antes vimos que las hojas de preprocesado o de visualización en general tienen un punto de entrada bien definido: `preproc.xsl` y `view.xsl`, respectivamente.

3.4 Herencia a partir de un tipo base

Bien, pues lo mismo ocurre con el interior de toda hoja de usuario. Si recordamos la forma en que se buscaba el nombre de un tipo de documento determinado, el método es muy similar. Sea *L* el código de idioma de la localización actual según ISO 639 y *P* el código de país según ISO 3166. Si el usuario ha seleccionado actualmente la hoja *H* de usuario de preprocesado, se intentará resolver la URI `current_preproc` a uno de estos ficheros XSLT, en el mismo orden:

1. `xslt/preproc/H/H_L_P.xml`
2. `xslt/preproc/H/H_L.xml`
3. `xslt/preproc/H/H.xml`

Una consecuencia de este esquema es que si vamos a presentar varias traducciones del texto generado, no deberíamos repetir la funcionalidad de la hoja varias veces, sino dividirla en dos partes:

1. El punto de entrada antes mencionado definirá una serie de variables y/o plantillas con nombre para localizar el texto generado, e importará a la hoja principal que proporcione la verdadera funcionalidad.
2. La hoja principal con toda la funcionalidad no generará texto por su cuenta, sino que quedará completamente dependiente de las variables de la hoja que actúe de punto de entrada.

Esto le resultará muy familiar a cualquiera que haya trabajado con *gettext*, por ejemplo: por un lado tenemos el diccionario de cadenas, y por otra el código propiamente dicho.

Este es el enfoque que se ha seguido en la hoja de visualización por defecto, `xml`, por ejemplo: se tiene el fichero `xml.xml`, que actúa como punto de entrada por defecto en español, y `principal.xml`, que realmente implementa la funcionalidad necesaria.

NOTA

A la hora de importar ficheros XSLT concretos, hay que emplear la ruta relativa completa a partir del directorio `xslt`: así, cuando `xml.xml` importa a `principal.xml`, ha de emplear la ruta relativa completa, aunque se halle en el mismo directorio:

```
<xsl:import href="view/xml/principal.xml"/>
```

3.4. Herencia a partir de un tipo base

El propio estándar XSLT ya define mecanismos para la herencia: a través del elemento `xsl:import` podemos importar todas las reglas de otro fichero XSLT, con menos precedencia que las actuales, de tal forma que podamos efectivamente especializar dicha hoja, redefiniendo y ampliando ciertos aspectos de forma parecida a la herencia del mundo orientado a objetos.

3 Creación de hojas de usuario

Sin embargo, no se debe usar directamente de esa forma: el estándar sólo hace referencia a URL estáticas, con lo que perderíamos la capacidad de emplear la infraestructura de internacionalización que antes mencionamos.

Para poder seguir usando el esquema de resolución dinámica de puntos de entrada para la hoja que vayamos a especializar, tendremos que seguir usando URI especiales. En particular, URI de la forma `view_X` se resuelven al punto de entrada correcto de la hoja `X` de usuario de visualización. Las URI de la forma `preproc_X` son sus análogos para las hojas de preprocesado.

La hoja de preprocesado para `ACL2`, `ppACL2`, especializa la hoja de preprocesado `xml` así:

```
<xsl:import href="preproc_xml"/>
```

Resulta interesante mencionar que la especialización se puede hacer a muchos niveles, como de costumbre. Así, `summaries` y `reverse` son a su vez especializaciones de `ppACL2`.

3.5. Hojas de preprocesado: particularidades

Ya hemos comentado anteriormente que la principal responsabilidad de toda hoja de preprocesado era añadir el atributo `UID` con identificadores únicos para cada nodo del árbol, para que el usuario pudiera así seleccionarlo y conseguir una visualización en formato `XHTML` de su contenido relevante.

También comentamos que escribiendo nuestra hoja de preprocesado como una especialización de la hoja `xml` no tendríamos que preocuparnos de ello. Sin embargo, hemos de tener en cuenta un poco el diseño de esta hoja para ver cómo se usa.

En cuanto arranca, esta hoja cambia al modo `process-node`, en el cual dispone de una regla por defecto para todo elemento que:

1. Crea una copia del elemento actual, junto con sus atributos.
2. Añade el atributo `UID` con el identificador único del elemento.
3. Invoca las plantillas del usuario sobre el nodo actual, para que añada nuevos atributos, elementos o en general lo que quiera. La única condición es que sean parte del modo por defecto de `XSLT` (es decir, carezcan de atributo `mode`). Por defecto no hace nada.
4. Copia todos los hijos de tipo texto.
5. Invoca otra vez plantillas del usuario, esta vez bajo el para determinar qué hijos se procesarán a continuación, y en qué orden. Por defecto continúa en el mismo orden, por todos los hijos del nodo actual.

Como vemos, esta hoja sigue el patrón de diseño Método Plantilla, en una versión del principio de Hollywood: "No nos llames; nosotros te llamaremos". Cualquier hoja

3.6 Hojas de visualización: particularidades

que especialice a ésta debe únicamente de definir las plantillas del modo por defecto y modo `process-children` como lo vea conveniente para modificar y expandir su comportamiento.

Ahora que sabemos cómo añadir nuevos atributos, elementos, o incluso cómo quitarlos (no recorriéndolos bajo el modo `process-children`), podemos pasar a hablar de una serie de atributos especiales que nos permiten hacer cosas sobre cómo ve el usuario nuestro documento:

hidden Si este atributo se pone a "1", el elemento no será visible por el usuario.

leaf Si este atributo se pone a "1", el elemento será considerado como hoja, y sus *hijos* no serán visibles al usuario.

Un ejemplo, extraído de la hoja de visualización de YAML, que oculta todos los nodos con nombre `_key`:

```
<!-- Hide all yaml:_key elements -->
<xsl:template match="*[local-name(.)='_key']">
  <xsl:attribute name="hidden">1</xsl:attribute>
</xsl:template>
```

nodeicon Contiene la ruta relativa a partir del mismo directorio donde se halla el repositorio de hojas de usuario (es decir, el directorio `xslt`) al icono de 16x16 píxeles que se desee mostrar para dicho nodo.

La hoja de visualización de demostraciones de ACL2 usa esto para mostrar de forma sencilla qué elementos han tenido éxito en su demostración y en cuáles no.

nodelabel Contiene la etiqueta a usar para el nodo en cuestión. Esto permite levantar así algunas restricciones típicas sobre los árboles XML normales, en los cuales los nombres de elementos no pueden tener espacios o ciertos caracteres.

La hoja de visualización de ficheros YAML/JSON usa este atributo para poner etiquetas en los pares clave/valor de los mapas: YAML es más permisivo en las claves de los mapas que XML, permitiendo espacios, por ejemplo.

```
<!-- Label map values using their keys -->
<xsl:template match="*[local-name(.)='_value']">
  <xsl:attribute name="nodelabel">
    <xsl:value-of select="preceding-sibling::*[local-name(.)='_key'][1]"/>
  </xsl:attribute>
</xsl:template>
```

3.6. Hojas de visualización: particularidades

A la hora de elaborar una hoja de visualización, hemos de tener en cuenta dos cosas. En primer lugar, normalmente especializaremos la hoja `xml`, consiguiendo así

3 Creación de hojas de usuario

ciertas facilidades, como la plantilla `skeleton`, a la cual le pasaremos el argumento `rtf` (Result Tree Fragment) con el código XHTML que queramos que aparezca dentro del cuerpo del documento.

Este esqueleto, por defecto, generará enlaces a todos los antecesores del nodo actual antes del cuerpo, y a todos los hijos inmediatos después del cuerpo.

Por defecto, la hoja de visualización de `xml` mostrará todos los atributos y nodos texto hijos del nodo actual. En este caso no hay tantos requisitos sobre la estructura de la hoja de visualización como antes. De hecho, podemos ignorar por completo la plantilla de generación del esqueleto y usar el nuestro.

Sin embargo, una cosa que sí es importante es la generación de hipervínculos entre distintos nodos. XMLEye añade la funcionalidad necesaria para cualquier hipervínculo que necesitemos: desde direcciones Web normales, de la forma `"http://..."`, pasando por enlaces a partes específicas (`"#nombredelancla"`), e incluso a otros nodos.

Nos pararemos en la parte específica a XMLEye. Podemos crear un enlace entre un nodo y otro usando la sintaxis `"#xpointer(ruta)"`, donde `ruta` es la ruta absoluta e única del nodo anterior desde la raíz. Como esto no es tan fácil de implementar, disponemos en el nombre de espacios `"xalan://es.uca.xmleye.xpath.GestorRutasXPath"` de la función `obtenerRuta`, que hace precisamente eso.

Para usarla, añadimos la declaración de dicho nombre de espacios al elemento `xsl:stylesheet` y ya podemos usarla así, por ejemplo:

```
<xsl:template match="encapsulate|local" mode="ppacl2_printnode">
  <h2 class="section">
    <a href="#xpointer(rutasxpath:obtenerRuta(.,/))">
      <xsl:value-of select="name(.)"/>
    </a>
  </h2>
  <pre>
    <xsl:value-of select="@formula"/>
  </pre>
</xsl:template>
```

Como vemos, la función toma dos argumentos: el primero es el nodo al cual queremos crear el enlace (aquí el nodo actual), y el segundo es el nodo que vamos a tomar como raíz (el nodo raíz, valga la redundancia).

Si tenemos que crear un enlace a un nodo de otro documento, la sintaxis es muy parecida: `rutadelfichero#xpointer(rutaxpath)"`, donde ahora indicamos tanto la ruta del fichero como la ruta XPath que queremos visualizar dentro de él.

Bibliografía

- [CSFP07] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Control de versiones con Subversion. <http://svnbook.red-bean.com/>, 2007.
- [Fre08] Freedesktop.org. Software/shared-mime-info. <http://freedesktop.org/wiki/Software/shared-mime-info>, 2008.