ESCUELA SUPERIOR DE INGENIERÍA

Segundo Ciclo en Ingeniería Informática

XMLEye: transformador y visor genérico de documentos estructurados

Curso 2007-2008

Antonio García Domínguez
Cádiz, 13 de octubre de 2008

ESCUELA SUPERIOR DE INGENIERÍA

Segundo Ciclo en Ingeniería Informática

XMLEye: transformador y visor genérico de documentos estructurados

DEPARTAMENTO: Lenguajes y Sistemas Informáticos.
DIRECTORES DEL PROYECTO: Francisco Palomo Lozano e Inmaculada Medina Bulo.
AUTOR DEL PROYECTO: Antonio García Domínguez.

Cádiz, 13 de octubre de 2008

Fdo.: Antonio García Domínguez

Índice general

Índice de figuras

Índice de cuadros

1 Introducción

1.1. Metalenguajes en el intercambio de información

Comunicar dos o más aplicaciones entre sí (en máquinas o espacios de memoria distintos, o incluso la misma aplicación en ejecuciones separadas en el tiempo) significa, a grandes rasgos, intercambiar secuencias de bits que siguen una serie de reglas que les dan estructura y significado, es decir, que pertenecen a un *lenguaje*.

Evidentemente, en ambos extremos de la comunicación debe de existir la lógica necesaria para recuperar esa estructura y significado a partir de los bits: los analizadores *léxicos* reconocen los lexemas que constituyen los elementos básicos y los analizadores *sintácticos* y *semánticos* se ocupan de crear estructuras de datos más elaboradas a partir de ellos.

Originalmente, tanto la gramática con las reglas que describían la sintaxis de la mayoría de los documentos, como el contenido que debían tener, eran propios de cada lenguaje. Esto originaba muchas complicaciones imprevistas al tener que integrar varias aplicaciones, posiblemente desarrolladas por diferentes organizaciones, y para entornos hardware y software distintos.

Las razones de este enfoque *ad hoc* eran las limitaciones en memoria y capacidad de proceso del hardware de la época: primaba la eficiencia en recursos computacionales sobre los costes de desarrollo, así que los lenguajes usados eran un fiel reflejo (a veces demasiado) de las representaciones internas de los datos. Con el tiempo, el avance en la tecnología alteró este equilibrio para la gran mayoría de aplicaciones que con el auge de Internet necesitaban más que nunca garantizar el intercambio fiable y perdurable de la información con una mejor gestión de errores y comprobaciones más estrictas.

Esto impulsó la popularidad de los *metalenguajes*, que definían las reglas no ya de uno, sino de familias completas de lenguajes. Los desarrolladores podían partir de una base de reglas y herramientas ya existentes para definir y explotar sus lenguajes de intercambio de información.

Las familias de lenguajes definidas por los metalenguajes pueden tener diversos usos, no excluyentes entre sí:

- Lenguajes de *marcado*, que emplean etiquetas para anotar el contenido de los documentos con información acerca del significado de sus elementos. Son lenguajes descriptivos, más que procedimentales. Por ejemplo, en lugar de indicar que se imprima una serie de cadenas separadas por saltos de línea de 16 puntos, al estilo de PostScript, indicaríamos que dicho grupo de cadenas forman un párrafo.

  Algunos de los primeros metalenguajes de marcado fueron GML (Generalized Markup Language) [31] (1973) y su sucesor, SGML (Standard Generalized Markup Language), estandarizado como ISO 8879:1986. HTML (Hyper Text Markup Language), sobre el que se apoya hoy en día toda la World Wide Web, es una derivación de SGML.

  Posteriormente, se empleó SGML para definir XML (eXtensible Markup Language) [8], el metalenguaje por excelencia hoy en día. Sacrifica gran parte de la flexibilidad de SGML para simplificar su manipulación automática. Además de las ventajas de los metalenguajes y de los lenguajes de marcado, XML permite definir vocabularios fácilmente extensibles por terceras partes que acomoden nueva información antes no prevista.

  Uno de los ejemplos más notables del uso de XML es el formato para documentos ofimáticos ODF (Open Document Format), publicado como el estándar ISO 26300:2006, e implementado en diversos paquetes ofimáticos, como OpenOffice, KOffice o Google Docs.

- Lenguajes de *serialización* y *deserialización* de estructuras de datos de memoria a un flujo de bytes y viceversa, respectivamente. Esto permite enviar dichas estructuras a través de una conexión de red o guardarlas en ficheros, por ejemplo.

  Aunque XML se usa ampliamente para este fin, hoy en día YAML (YAML Ain't a Markup Language) está reuniendo cada vez más adeptos, por su sintaxis más concisa y fácil de leer y su facilidad de uso. JSON (JavaScript Object Notation), un subconjunto de YAML 1.1 más fácil de analizar pero menos legible, está cobrando fuerza en la comunidad de desarrolladores Web.

  Algunos ejemplos del uso de YAML y JSON incluyen los volcados de bases de datos del entorno de desarrollo web Django, los descriptores de módulos Perl y los marcadores del navegador web Mozilla Firefox en sus versiones 3.0 y posteriores.

Sin embargo, muchos de estos metalenguajes no fueron diseñados para ser legibles por personas. Este es el caso de aquellos basados en XML o JSON, por ejemplo. Otras veces, incluso para metalenguajes más legibles como YAML, los documentos son simplemente demasiado complejos o grandes.

El único tipo de herramienta que puede resolver estos problemas es un visor, que reúna información agregada en un formato fácil de comprender. Por otro lado, la creación de un visor específico para un lenguaje determinado implica un esfuerzo considerable, que sólo se halla justificado si dispone de la «masa crítica» necesaria. Especializar un

visor genérico ya existente supone el mejor enfoque para los formatos con comunidades de usuarios más pequeñas.

1.2. Formatos de intercambio de información que no se basan en metalenguajes

Los problemas de legibilidad antes mencionados no son específicos de los formatos basados en metalenguajes. Las grandes capacidades de procesamiento automatizado que nos otorgan las computadoras nos permiten tratar problemas mucho más grandes, pero con cada vez mayor frecuencia nos vemos impedidos a la hora de analizar los propios resultados.

También aquí, la solución usual para formatos lo bastante populares es crear visores específicos para ellos. En muchos casos, el documento original se convierte primero a un formato basado en un metalenguaje, para reutilizar la amplia gama de documentación, herramientas y bibliotecas existentes. Esto no debería suponer una pérdida de información: XML, por ejemplo, permite construir lenguajes arbitrariamente complejos. Otras veces, es la propia aplicación la que pasa a basar el formato de su salida en un metalenguaje, como en el caso del sistema de demostraciones automatizadas Mizar [11, 52], que pasó a utilizar XML.

Con la infraestructura básica establecida, se puede pasar directamente a crear el visor. Si ya existiera un visor genérico basado en el metalenguaje usado, podríamos evitar también este paso: sólo habría que especializarlo para nuestro formato.

La reducción de las barreras necesarias para crear un visor nos permitiría concentrarnos más en el análisis del formato origen y en la definición del formato final de presentación, sin tener que molestarnos en los detalles de implementación necesarios para mantener un historial de documentos, habilitar la navegación por hipervínculos y otros aspectos de bajo nivel.

Así, podríamos manejar formatos considerados más difíciles de procesar, como aquellos que carecen de gramáticas explícitamente definidas. Tal es el caso de muchos resultados generados de manera automática y dirigidos al usuario final: en este caso, la gramática está definida por el propio código que la produce. Convertir la salida producida a un metalenguaje puede resultar más eficiente que reescribir el programa original si no se ha hecho una separación entre el procesamiento y la presentación del resultado desde un primer momento.

Uno de estos formatos con gramáticas implícitas es el de las demostraciones producidas por el sistema ACL2 (A Computational Logic for Applicative Common Lisp). ACL2 es un sistema de razonamiento automatizado desarrollado en la Universidad de Texas en Austin y publicado bajo la licencia GNU GPL. Utiliza un subconjunto funcional (*aplicativo* o sin efectos colaterales, es decir, sin bucles, ni asignaciones, ni variables

globales, etc.) de Lisp para realizar demostraciones formales de propiedades de sistemas software y hardware complejos.

En [39] se muestran algunos ejemplos de demostraciones sobre sistemas complejos realizadas mediante ACL2. Entre ellos, se menciona la verificación de que cierto microcódigo de un chip DSP (Digital Signal Processing) de Motorola realmente implementaba ciertos algoritmos conocidos, y que llegó a los 84 megabytes de texto puro. No es sólo eso: la salida de ACL2 se realiza en una mezcla de lenguaje natural y expresiones Lisp que puede hacerse muy difícil y tediosa de manejar en demostraciones de interés práctico.

Puede darse también el caso de que la demostración se halle dividida a lo largo de varios ficheros, y la traza de una demostración fallida requiera así la inspección de varios ficheros enlazados entre sí.

Definiendo un visor para ACL2, se podría navegar de forma mucho más sencilla y cómoda, y partiendo de un marco genérico, podemos ahorrarnos la mayor parte del trabajo implicado en ello. Además, seguir los sutiles cambios producidos en el formato de la salida de ACL2 entre versiones se haría mucho más sencillo, al no estar atado el visor a esos detalles.

1.3. Objetivos

Este proyecto toma como punto de partida el Proyecto Fin de Carrera “Post-procesador y Visor de Demostraciones del Sistema ACL2”, en el cual se definió una conversión preliminar de un subconjunto de las demostraciones sin uso de libros de ACL2 a un lenguaje basado en XML, y un visor específico para este formato.

Los objetivos a cumplir son los siguientes:

- Generalizar el diseño del visor a un visor genérico de documentos basado en XML para su uso en un amplio rango de formatos, tanto basados como no basados en metalenguajes. Se necesitará:
  - Integración personalizable con los editores y los convertidores a XML de los múltiples formatos utilizados.
  - Apertura de múltiples documentos en una interfaz en pestañas o TDI (Tabbed Document Interface), con la posibilidad de mantener distintas configuraciones de visualización en cada documento abierto.
  - Recorrido de enlaces entre múltiples documentos, pudiendo seleccionar el nodo exacto a visualizar en el documento enlazado.
  - Retirada de toda lógica específica a ACL2 del visor, moviéndola al antiguo post-procesador.
- Mejorar la calidad global del producto a través de:

    - Mejorar la cantidad y calidad de la documentación, situándola en una plataforma colaborativa.
    - Facilitar la instalación del visor y sus extensiones.
    - Traducir la interfaz del visor al inglés.
    - Publicar el código fuente y otros ficheros asociados a través de una forja.
    - Ampliar el conjunto de casos de prueba de todos los productos.
- Refinar el antiguo post-procesador de ACL2 y sus hojas de estilo, ejecutando las siguientes acciones:
    - Integración de la lógica específica a ACL2 antes contenida en el visor.
    - Manejo de demostraciones que certifiquen libros y reutilicen sus definiciones a múltiples niveles, teniendo en cuenta el grafo completo de dependencias y evitando enviar a ACL2 el mismo fichero varias veces.
- Crear una nueva extensión para procesar documentos basados en YAML y JSON, con sus hojas de estilo:
    - Conversión de documentos YAML 1.1 y JSON a XML, garantizando que no se producen pérdidas ni cambios de la información original.
    - Manejo de anclas y alias de YAML, con la debida creación de vínculos en el XML resultante.

1.4. Alcance

El proyecto se compone de tres productos:

- El producto principal es XMLEye, un visor genérico de documentos basados en XML, integrable con extensiones externas que convierten formatos no basados en XML y con hojas de estilo XSLT (eXtensible Stylesheet Language Transformations) que definen visualizaciones especializadas para ciertos formatos.
- `ACL2::Procesador`, un conversor integrable en XMLEye que toma la salida de ACL2 en lenguaje humano y su fuente Lisp y procesa ambos para obtener un fichero XML sin pérdida de información que otros programas puedan manejar fácilmente. Incluye sus propias hojas de estilo especializadas, también integrables en XMLEye.
- `YAXML::Reverse`, un conversor a XML de documentos YAML 1.1 y JSON. También incluye sus propias hojas de estilo.

Estos productos dispondrán de su propia documentación para usuarios y desarrolladores. Se habilitará un *wiki* y un espacio en una forja para permitir la futura colaboración de nuevos participantes.

1.4.1. Limitaciones del proyecto

Únicamente se utilizarán visualizaciones en HTML mediante una interfaz Swing. Generar árboles estáticos de ficheros XHTML (eXtensible Hyper Text Markup Language) o utilizar interfaces Swing dinámicas mediante JavaFX se dejan como trabajo futuro.

Por limitaciones de tiempo, `ACL2::Procesador` se restringe a un subconjunto de todos los eventos de ACL2, tratando demostraciones que, aunque complejas, no hacen uso de todo el potencial de ACL2. Su uso sería, pues, eminentemente didáctico.

Se ignoran en la actualidad los mensajes de la implementación de Lisp sobre la que se ejecuta ACL2, como los mensajes del recolector de basura, ya que existe actualmente una gran variedad de ellas y no sería factible ni útil añadir lógica para cada una. Por supuesto, sí se han considerado los mensajes de error durante la demostración propios de ACL2.

No se crearán hojas de estilo para lenguajes derivados de YAML o JSON para este proyecto. En futuras versiones se crearán hojas de estilo para ficheros de marcadores de Firefox y volcados de bases de datos de pruebas de Django.

1.4.2. Licencia

Se decidió publicar `ACL2::Procesador`, `YAXML::Reverse` y XMLEye como software libre bajo la licencia GPL (General Public License) en sus versiones 2 o posteriores. Las licencias de las principales bibliotecas, módulos Perl y demás herramientas utilizadas en este Proyecto son:

- Perl: la versión original dirigida a sistemas UNIX puede usarse bajo o bien la licencia denominada "Artística" (disponible bajo `http://www.perl.com/language/misc/Artistic.html`), o bajo la GNU (GNU is Not Unix) GPL en sus versiones 1 o posteriores.
- La distribución de Perl para Windows usada, Strawberry Perl, y la mayoría de los módulos Perl usados comparten el esquema de licenciado de Perl. Algunos módulos varían ligeramente, como en el caso de `XML::Writer`, que no impone restricción alguna en su uso comercial o no comercial, o `XML::Validate`, que únicamente admite la licencia GPL.
- A través de algunos de los módulos Perl, se emplean de forma indirecta las bibliotecas `libxml2`, `libxslt` y `libyaml`. Todas se hallan bajo la licencia MIT (Massachusetts Institute of Technology), prácticamente equivalente a la licencia BSD (Berkeley Software Distribution) revisada, en la que se retira la necesidad de incluir el aviso del uso de dicho componente en todo el material publicitario del producto final. Al no ser una licencia *copyleft*, los usuarios no se hallan obligados a devolver a la comunidad los cambios que hagan sobre las bibliotecas.

- OpenJDK 6.0: GPL, con la excepción Classpath (véase `http://www.gnu.org/software/classpath/license.html`), que permite enlazar módulos independientes bajo cualquier licencia con módulos escritos bajo la GPL versión 2, sin que la GPL se extienda a dichos módulos. De esta forma, puede usarse para desarrollar aplicaciones comerciales, por ejemplo, y no supone un impedimento para su adopción. En última instancia, es muy similar a la licencia LGPL (Lesser General Public License).

  Un pequeño porcentaje (menor al 5 %) de OpenJDK sigue estando bajo licencias propietarias: Sun está ahora trabajando en sustituirlas por componentes equivalentes.
- IcedTea 6.0: GPL con la excepción Classpath. Reemplaza ese pequeño porcentaje propietario de OpenJDK y añade un proceso de compilación completamente basado en herramientas libres.
- Apache Ant: APL (Apache Public License) 2.0. Esta licencia tampoco es *copyleft*, y es compatible con la GPL en sus versiones 3 o superiores.
- ACL2: GNU GPL, combinando a su vez componentes bajo la GNU GPL (por ejemplo, la biblioteca `readline`) y bajo la GNU LGPL (como el entorno Lisp GCL (GNU Common Lisp)).

1.5. Visión general

Tras una revisión del calendario seguido, detallaremos a lo largo del resto de la memoria el proceso de análisis, diseño, codificación y pruebas que se siguió al realizar el proyecto.

En particular, se describirá el estado del proyecto en su última iteración, que tiene la arquitectura y el diseño definitivos.

Los manuales de usuario y de instalación se incluyen tras un resumen de los aspectos más destacables de proyecto y las conclusiones. En dicho manual, se halla un apartado dirigido a usuarios avanzados que deseen definir sus propias hojas XSLT.

1.6. Glosario

1.6.1. Acrónimos

ACL2 A Computational Logic for Applicative Common Lisp

AMD Advanced Micro Devices

API Application Programming Interface

APL Apache Public License
ASCII American Standard Code for Information Interchange
BNF Backus-Naur Form
BOM Byte Order Mark
BSD Berkeley Software Distribution
C3 Chrysler Comprehensive Compensation System
CASE Computer Assisted Software Engineering
CPAN Comprehensive Perl Archive Network
CPU Central Processing Unit
CSS Cascading Style Sheet(s)
CVS Concurrent Versions System
DOM Document Object Model
DSP Digital Signal Processing
DTD Document Type Definition
EAFP Easier to Ask for Forgiveness than Permission
FDL Free Documentation License
FSF Free Software Foundation
GCL GNU Common Lisp
GIMP GNU Image Manipulation Program
GML Generalized Markup Language
GNU GNU is Not Unix
GPL General Public License
GTK+ GIMP Toolkit
GUI Graphical User Interface
HTML Hyper Text Markup Language
IBM International Business Machines
IDE Integrated Development Environment
IEEE Institute of Electrical and Electronics Engineers
J2SE Java 2 Standard Edition
JAXP Java API for XML Parsing
JDBC Java DataBase Connectivity
JDK Java Development Kit
JIT Just In Time

JRE Java Runtime Environment
JSON JavaScript Object Notation
JSR Java Specification Request
JVM Java Virtual Machine
LGPL Lesser General Public License
LTS Long Time Support
MIT Massachusetts Institute of Technology
MVC Model View Controller
MVP Model View Presenter
ODF Open Document Format
PAR Perl ARchive Toolkit
PDF Portable Document Format
POD Plain Old Documentation
POSIX Portable Operating System Interface, UniX based
Perl Practical Extraction and Report Language
SAX Simple API for XML
SGML Standard Generalized Markup Language
SVG Structured Vector Graphics
SWT Standard Widget Toolkit
TCK Technology Compatibility Kit
TDI Tabbed Document Interface
UML Unified Modelling Language
URI Uniform Resource Identifier
URL Uniform Resource Locator
UTF Universal Transformation Format
W3C World Wide Web Consortium
WWW World Wide Web
XHTML eXtensible Hyper Text Markup Language
XML eXtensible Markup Language
XP eXtreme Programming
XSL-FO eXtensible Stylesheet Language Formatting Objects
XSLT eXtensible Stylesheet Language Transformations
YAML YAML Ain't a Markup Language
YAXML YAML XML binding

1.6.2. Definiciones

Definiciones generales

Analizador léxico Programa o módulo que se ocupa de reconocer los símbolos terminales de un lenguaje dentro de las cadenas de texto recibidas por el analizador sintáctico.

Analizador sintáctico Programa o módulo que se encarga de agrupar las cadenas de símbolos terminales proporcionados por el analizador léxico en símbolos no terminales. Pueden reflejar así restricciones de mayor nivel de abstracción sobre las cadenas válidas del lenguaje. Normalmente le sigue un analizador semántico.

Analizador semántico Programa o módulo cuyo cometido es realizar cualquier procesamiento adicional específico para el lenguaje del texto procesado, y que el analizador sintáctico no sea capaz de implementar a través de su gramática. Excepto por los lenguajes más simples, una gramática estándar no puede representar todas sus restricciones.

Deserialización Proceso inverso a la serialización. En él se recupera una copia con el mismo contenido semántico que la estructura de datos original, a partir de la secuencia de bytes resultante de la serialización.

Forja Sitio web dedicado a hospedar proyectos de desarrollo de software. Normalmente, las forjas son gratuitas para proyectos de software libre y/o código abierto, pero también existen forjas [3] para entornos comerciales. Los servicios proporcionados varían mucho entre forja y forja, pero como mínimo suelen tener espacio en algún sistema de control de versiones, un sistema de control de incidencias y un área en la que alojar ficheros para su descarga por los usuarios.

Lenguaje Subconjunto del conjunto potencia de los caracteres de un alfabeto. Un lenguaje puede definirse a partir de una gramática que especifique restricciones en la sintaxis de las cadenas consideradas válidas. En el caso de los lenguajes artificiales, esta gramática viene definida explícitamente a través de algún mecanismo formal.

Lenguaje de marcado Lenguaje artificial que decora un texto sencillo con anotaciones acerca de la estructura, el significado y/o el formato de sus partes. Estas anotaciones son de tipo declarativo, describiendo qué características tiene cada región del texto, y no cómo se consiguen.

Algunos ejemplos muy conocidos incluyen SGML, HTML, XML, TEX o LATEX.

Lenguaje de serialización Lenguaje artificial que representa estructuras de datos a partir de una serie de símbolos terminales organizados según una serie de reglas. Idealmente, un lenguaje de serialización debería ser conciso, fácilmente legible y eficiente tanto al serializar como deserializar.

Metalenguaje En el ámbito en que nos ocupa, un metalenguaje consiste en una serie de reglas que proporcionan la sintaxis y semántica, dentro de un modelo de datos, como un árbol o un grafo, por ejemplo, de una familia completa de lenguajes. No llega a detallar el contenido exacto de los documentos, quedando por concretar en cada una de sus instancias. Estas que se hallan en un nivel más bajo de abstracción, y ya son lenguajes completamente definidos.

Serialización Proceso en el que se traduce una estructura de datos en memoria a una secuencia de bytes que representan su contenido. Esta puede posteriormente enviarse a través de una conexión de red, guardarse en un fichero, etc.

Símbolo no terminal Símbolo de una gramática que puede ser expandido a una sucesión de símbolos terminales y/o no terminales.

Para ilustrar mejor el concepto, supongamos que tenemos la gramática $G = (N, \Sigma, P, S)$ con los símbolos terminales $\Sigma = \{a, b\}$, los símbolos no terminales $N = \{E\}$, el símbolo inicial $S = E$, y el conjunto de reglas P de producción en BNF (Backus-Naur Form) siguiente:

$$\begin{aligned} E &\rightarrow Eb \\ E &\rightarrow a \end{aligned}$$

Podríamos hacer la siguiente expansión para obtener la secuencia de símbolos terminales abb:

$$\begin{aligned} E &\Rightarrow Eb \\ &\Rightarrow Ebb \\ &\Rightarrow abb \end{aligned}$$

Símbolo terminal Símbolo de una gramática que representa una secuencia de caracteres de un alfabeto con un significado coherente. Algunos ejemplos incluyen "división" (con la cadena "/", por ejemplo) o "entero" ("200"). El significado de un símbolo terminal puede ser el mismo para todas sus ocurrencias (como en "división"), o puede variar en cada instancia (como en el caso de "entero").

Wiki Sitio web cuyas páginas pueden ser editadas a través del propio navegador web, distribuyendo su mantenimiento a una comunidad completa de usuarios. Existe una gran variedad: desde *wikis* de empresa sólo accesibles por una parte de los empleados, hasta *wikis* de acceso y modificación públicos, como la conocida Wikipedia. El primer wiki fue WikiWikiWeb [59].

Definiciones en el dominio de YAML, JSON y YAXML

Alias Identificador único establecido sobre un nodo del documento, para posterior referencia mediante anclas.

Ancla Referencia a un alias definido en otra parte del documento. Evita tener que repetir contenido idéntico en varias partes del documento en la salida serializada.

Documento Estructura de datos nativa al lenguaje de programación utilizado, consistente en un único *nodo* raíz.

Se obtiene a partir del flujo tras los eventos de *análisis*, la *composición* del grafo a partir de estos, y la *construcción* de la estructura nativa de datos a partir del grafo.

En dirección inversa, una estructura de datos nativa se *representa* por un grafo dirigido y con raíz de nodos, se convierte a un árbol ordenado, que se recorre para generar eventos de *serialización* y finalmente *presentar* la estructura original en forma de una sucesión de bytes.

Flujo Sucesión de bytes que cumple las reglas correspondientes de producción de la gramática de YAML, y en la cual se pueden hallar presentados cero, uno o más *documentos* YAML, separados por delimitadores. El primer delimitador es opcional.

Opcionalmente, puede comenzar por un BOM (Byte Order Mark) de UTF (Universal Transformation Format) indicando si la entrada se halla codificada en UTF-8, UTF-16 Little Endian, o UTF-16 Big Endian. Por defecto, se asume que la entrada se halla codificada en UTF-8.

Nodo Cualquier elemento con entidad propia del árbol del documento. Puede tener otros nodos anidados en su interior.

Escalar Categoría de todo nodo con contenido atómico opaco consistente en una serie de cero o más caracteres Unicode. El significado de dichos caracteres se concreta a partir de su etiqueta.

Esquema Combinación de un conjunto de etiquetas y un mecanismo para etiquetar nodos por defecto. A través de estos esquemas, como el JSON Schema o el Core Schema, podemos darles interpretaciones estándar a los nodos secuencia, nodos vector asociativo o nodos escalar.

Por ejemplo, conseguimos que los escalares con el contenido “true” se les dé la etiqueta “tag:yaml.org,2002:bool”, indicando que se trata de un valor de verdad booleano.

Etiqueta Todo nodo tiene una etiqueta, que indica el tipo de información que representan. Por ejemplo, los enteros tienen la etiqueta “int”, y los números de coma flotante “float”.

Secuencia Categoría de los nodos que contienen cero o más nodos, numerables según su posición relativa entre los demás.

Vector asociativo Categoría de todo nodo que contiene un conjunto de cero o más pares clave-valor, en donde las claves han de ser únicas. Tanto las claves como los valores pueden ser nodos arbitrariamente complejos y de cualquier tipo.

Definiciones en el dominio de ACL2

Átomo Lisp Mínima unidad sintáctica de toda expresión Lisp. Puede ser un símbolo, un número, un carácter o una cadena.

Clausura lambda Par formado por un bloque de código a ejecutar y las ligaduras a variables usadas en dicho código locales al bloque en que fue definida la clausura, que persisten tras la ejecución de dicho bloque.

El concepto, creado por Alonzo Church en su λ-cálculo, ha sido llevado a diversos lenguajes de programación con mayor o menor exactitud: Scheme (el primer lenguaje en implementarlo completamente [29, 41, 22, 50]), Perl y sus subrutinas anónimas, Java y sus clases anónimas, etc.

Dependencia inversa Se dice de la relación entre un evento y aquellos eventos que lo mencionan en su demostración.

Evento Orden de ACL2 que modifica su mundo lógico en alguna medida.

Libro Conjunto de funciones y teoremas que expresan propiedades sobre ellas y que forman una teoría computacional en ACL2.

Meta Unidad mínima de toda demostración de ACL2, a la que le corresponde una etiqueta única dentro de la demostración, una conjetura y una serie de acciones que conducen a su demostración, refutación, simplificación o división en otras metas.

Una meta puede contener submetas, generando un árbol de demostración de manera recursiva. Por supuesto, existe una meta raíz en toda demostración de ACL2, que recibe la etiqueta “Goal”.

Mundo lógico A nivel abstracto, el *mundo lógico* o simplemente *mundo* de ACL2 es la base de datos en la que basa sus demostraciones, y que podemos extender con nuevas reglas, funciones y otros elementos.

A nivel de implementación, un *mundo* no es más que una lista propia de Lisp de tripletas propiedad-símbolo-valor. Podemos crear nuestros propios mundos si así lo deseamos y utilizarlos con las funciones `getprop` y `setprop`. El mundo de ACL2 se halla bajo el nombre *’current-acl2-world*, pero no puede ser modificado excepto por un evento.

Paquete Lisp Nombre de espacios que califica a un identificador, reduciendo el riesgo de una colisión. Por defecto, en ACL2 se usa el paquete ACL2, pero podemos definir nuestros propios paquetes y cambiar entre ellos libremente.

Par Lisp Estructura de datos ordenada que engloba a dos elementos.

El formato básico es *(a . b)*, donde *a* o *b* pueden ser átomos o pares Lisp. Así, para crear una lista con los elementos 1, 2 y 3, escribiríamos *(1 . (2 . (3 . nil)))*, o en una notación simplificada, *(1 2 3)*. Listas como ésta donde el último elemento es *nil* se conocen como *listas propias*.

Proyecto Grafo acíclico dirigido con raíz en un fichero Lisp que describe una demostración a realizar mediante ACL2, utilizando una serie de eventos definidos en varios libros, ya se hallen integrados en la distribución de ACL2 o hayan sido creados por el usuario.

Un proyecto depende de una serie de *subproyectos*, cuyas raíces son los nodos directamente adyacentes al nodo raíz del proyecto principal.

S-expresión Su nombre es una abreviatura de "expresión simbólica", y consiste en un átomo Lisp o en un par Lisp. Los eventos de ACL2, por ejemplo, están formados por S-expresiones, utilizando notación prefija para la aplicación de funciones y operadores.

1.6.3. Otras tecnologías

XSLT

XSLT (véase [12]) se trata de un lenguaje XML estandarizado por el W3C (World Wide Web Consortium) que define hojas de estilo capaces de transformar una entrada XML a otros formatos.

El formato de salida no ha de ser necesariamente XML también: se podría generar texto arbitrario, o un documento HTML.

El uso de hojas XSLT permite definir las transformaciones de forma declarativa, simplificando mucho la tarea de transformación frente a otros métodos, como el uso del API DOM (Document Object Model), o el procesado mediante SAX (Simple API for XML).

Permite que la visualización del documento XML en XMLEye se halle dirigida por datos y no por código, y así sea extensible y personalizable por el usuario sin necesidad de recompilación.

XPath

XPath (véase [13]) es un estándar relacionado con XML del W3C que además de poderse usar independientemente, es parte fundamental de otras tecnologías XML, como XSLT, o XQuery, un lenguaje especializado para consultas sobre datos XML de cualquier fuente (incluso una base de datos).

Su propósito es permitir identificar fácilmente conjuntos de nodos de un documento XML, siguiendo un modelo de datos similar al del estándar DOM, con ligeras diferencias.

La sintaxis de una expresión XPath es declarativa, usando condiciones que deben cumplir los nodos resultado de dicha expresión. Se pueden filtrar nodos en función de su tipo, nombre (en caso de ser elemento), atributos, descendientes, etc.

En XMLEye se usa como parte de XSLT en las transformaciones y, por separado, en las búsquedas.

YAXML

YAXML (YAML XML binding) [60] es una correspondencia XML → YAML en borrador desde 2006. Se halla descrita a nivel informal de manera textual, y con mayor rigor a través de una transformación automática implementada mediante una hoja de estilos XSLT 1.0.

`YAXML::Reverse` es un módulo Perl que implementa la correspondencia inversa y al mismo tiempo sirve para depurar y mejorar YAXML, aprovechando el ciclo YAML → XML → YAML resultante.

2 Desarrollo del calendario

No se ha seguido una estricta planificación durante la realización de este proyecto, dado que se ha desarrollado la mayor parte de él de forma simultánea a los estudios del último curso del Segundo Ciclo en Ingeniería Informática, durante mi participación en el II Concurso Universitario de Software Libre [46].

Los contenidos de la forja [17] han sido revisados en paralelo con el proyecto desde la elaboración de los paquetes Debian de cada producto.

2.1. XMLEye

2.1.1. Primera iteración: rediseño

El diseño del visor original se hallaba fundamentado en la premisa de que sólo era necesario visualizar un único documento a la vez. Al cambiar esta premisa, se necesitó revisar el diseño completo de la aplicación. Se cambió de un diseño basado en fachadas a un diseño basado en el patrón arquitectónico Presentación-Abstracción-Control.

Las pruebas de unidad existentes fueron una gran ayuda a la hora de mantener la funcionalidad durante el cambio.

2.1.2. Segunda iteración: interfaz TDI

Con el nuevo diseño, se pudo integrar una interfaz multidocumento TDI o basada en pestañas, con controles para cerrar pestañas individuales y cerrar todas las pestañas. Se implementó la capacidad de mantener el estado y opciones de visualización de cada documento de forma independiente.

2.1.3. Tercera iteración: hipervínculos entre documentos

Se extendió la sintaxis existente para enlaces entre nodos de un documento para poder enlazar con nodos de otros documentos, aprovechando el soporte de demostraciones que usen libros por el lado de `ACL2::Procesador`. Al pulsar en uno de esos enlaces, se abriría o seleccionaría la pestaña del documento en cuestión y se pasaría al nodo especificado.

2.1.4. Cuarta iteración: descriptores de formatos de documentos

Se retiró la lógica restante específica a ACL2 de XMLEye: en particular, se retiraron los diálogos explícitos de importación. Se reemplazó por un sistema de descriptores de formatos de documentos que generalizaban el enfoque anterior a cualquier formato para el que hiciera falta un conversor y/o un editor, y que se pudiera identificar por su extensión. Las órdenes especificadas son personalizables por cada usuario del sistema a partir de unos valores globales por defecto.

2.1.5. Quinta iteración: estabilización

Debido al importante número de cambios realizados en el diseño de la capa de aplicación, se consideró oportuno dedicar un tiempo prudencial a la depuración en detalle y simplificación del diseño en ciertos puntos. Un punto en el que se hizo mucho hincapié fue en el manejo de la concurrencia en XMLEye, que producía ocasionalmente problemas difíciles de reproducir.

2.1.6. Sexta iteración: paquete Debian

En esta iteración se creó un paquete Debian para XMLEye. Se hicieron diversos ajustes en XMLEye debidos a la necesidad de acomodarlo a las prácticas recomendadas por la política de Debian, y se aprovechó para verificar el correcto funcionamiento de XMLEye con OpenJDK, la versión de código abierto del JDK (Java Development Kit). Ahora XMLEye puede funcionar al 100 % sólo con software libre. También se añadió la posibilidad de abrir documentos con XMLEye desde la línea de órdenes.

2.1.7. Séptima iteración: manuales y traducción al inglés

Se escribieron sendos manuales de usuario y desarrollador en DocBook, convirtiéndose a los formatos HTML de una o varias páginas y PDF (Portable Document Format).

La versión PDF fue producida a través de unas hojas XSLT especializadas que producen código LaTeX, puesto que todas las hojas equivalentes existentes daban resultados insatisfactorios.

Además, se tradujeron al inglés todas las cadenas de la interfaz de XMLEye y los nombres de las clases de su código. La arquitectura ya se hallaba internacionalizada, por lo que no hubo que hacer más cambios aparte de indicar el idioma inglés como lenguaje por defecto.

2.1.8. Octava iteración: creación de un wiki

Se instaló, configuró y elaboró una primera versión de los contenidos de un wiki [18]. Detalla tanto las motivaciones detrás de XMLEye, como su arquitectura general, los métodos de instalación y qué tener en cuenta al añadir nuevos formatos de entrada y hojas de estilo a XMLEye.

2.2. ACL2::Procesador

2.2.1. Primera iteración: rediseño

Se revisó la organización del post-procesador para que siguiera las mejores prácticas reconocidas del CPAN (Comprehensive Perl Archive Network). Entre otras cosas, se añadió documentación al propio código en formato POD (Plain Old Documentation), se convirtieron las pruebas de regresión existentes a guiones Perl, y se añadieron más pruebas para asegurar la cobertura de la documentación.

Una importante ventaja de reescribir las pruebas de regresión usando mejores herramientas es la capacidad de calcular las diferencias entre la salida esperada y la obtenida de forma más inteligente, pudiendo ignorar de forma selectiva diferencias que no se consideren importantes a nivel semántico.

2.2.2. Segunda iteración: eventos para libros

Se añadió soporte para algunos eventos relacionados con el manejo de libros: IN-PACKAGE, DEFPKG, INCLUDE-BOOK y CERTIFY-BOOK.

2.2.3. Tercera iteración: grafos de dependencias

Se separaron de forma explícita los pasos de análisis del código fuente de las demostraciones, análisis de la salida resultante y producción de la salida. Esto posibilitó la generación previa al análisis de la salida de una demostración de su grafo de dependencias. Este grafo acíclico dirigido lista todos los eventos obtenidos de cada libro en uso, y detalla la ruta relativa a la definición del libro.

Empleando estos grafos, se pudieron refinar las hojas de estilos para que crearan los enlaces a las definiciones de aquellos elementos que se originaran en otros libros.

2.2.4. Cuarta iteración: integración con ACL2

Aprovechando la información de los grafos de dependencias, se añadió la lógica necesaria a `ACL2::Procesador` para invocar él mismo a ACL2 sobre los ficheros implicados en una demostración, haciendo un recorrido en post-orden del grafo. Los resultados del análisis son almacenados en ficheros, y sólo son recalculados cuando es necesario.

2.2.5. Quinta iteración: ejecutable monolítico y paquete Debian

Se integró la posibilidad de crear ejecutables monolíticos con todas las bibliotecas y módulos necesarios y el propio intérprete de Perl mediante el módulo PAR, y se elaboró un paquete Debian, empaquetando además todas sus dependencias, y revisando el código de `ACL2::Procesador` para que cumpliera la política de Debian.

2.2.6. Sexta iteración: traducción al inglés

Se tradujeron todos los mensajes de error y los POD más importantes a inglés. La traducción del código se deja para posteriores versiones.

2.2.7. Séptima iteración: actualización para ACL2 v3.3

Se revisó `ACL2::Procesador` para tener en cuenta los ligeros cambios producidos en la salida de ACL2 en su versión 3.3.

2.3. YAXML::Reverse

2.3.1. Primera iteración: Procesamiento de YAML 1.0

Se realizó una primera versión consistente en un solo guión Perl que cargaba el fichero YAML 1.0 fuente y hacía la correspondencia inversa a YAXML. Se implementó la conversión del ejemplo de la factura de [60], y de los 5 ejemplos de [56].

2.3.2. Segunda iteración: reorganización y traducción

Pronto se vio que el problema era más complejo de lo esperado, y se reestructuró el guión en forma de un módulo Perl, igual que se había hecho con `ACL2::Procesador`. El código y toda la documentación se tradujo directamente a inglés.

2.3.3. Tercera iteración: ejecutable monolítico y paquete Debian

De forma similar a `ACL2::Procesador`, se creó un paquete Debian siguiendo las directrices de la política de Debian, y se posibilitó la generación de ejecutables monolíticos que incluyeran todos los módulos, bibliotecas y el intérprete Perl necesario.

2.3.4. Cuarta iteración: YAML 1.1/JSON y otras mejoras

La versión hasta el momento utilizaba `YAML::Syck`, que funcionaba correctamente para ficheros YAML 1.0. Sin embargo, cuando hubo que utilizar ficheros JSON, no funcionaba debidamente: aparentemente, JSON es subconjunto de YAML 1.1, no de la versión 1.0.

Se cambió a `YAML::XS`, y se corrigieron otras cuestiones relativas al procesamiento de ficheros de entrada codificados en UTF-8.

2.4. Diagrama Gantt

Se ha elaborado un diagrama Gantt (figura 2.1 en la página 39) para poder visualizar con mayor facilidad la distribución de las iteraciones.

Se pueden ver las fechas exactas de inicio y fin de cada producto y cada iteración de cada producto y sus duraciones D_R y D_I en días reales y días ideales, respectivamente, en el cuadro 2.1 de la página 40. Tomado de [6], un día ideal equivale a la cantidad

de trabajo que puede realizarse en un día por una persona, sin distracción alguna y a máximo rendimiento.

Se ha empleado la herramienta *Gantt Project* para dibujar el diagrama y GNOME Planner para calcular las fechas con mejor precisión. Ambos son código abierto: la primera está hecha en Java y disponible en `http://ganttproject.sourceforge.net`, y la segunda se halla escrita en C++ con la biblioteca GTK+ (GIMP Toolkit) y se puede encontrar bajo la dirección `http://live.gnome.org/Planner`.

2.5. Porcentajes de esfuerzo

Los porcentajes aproximados de esfuerzo se detallan en los cuadros 2.2 al 2.4 (páginas 40 a 41).

La mayoría de los requisitos eran conocidos de antemano desde el final del Proyecto Fin de Carrera sobre el que se basa éste, por lo que hay muy poco énfasis en la fase de análisis. El importante rediseño de XMLEye ha motivado que los porcentajes de esfuerzo dedicados a la prueba de XMLEye y a su propia construcción sean prácticamente equivalentes.

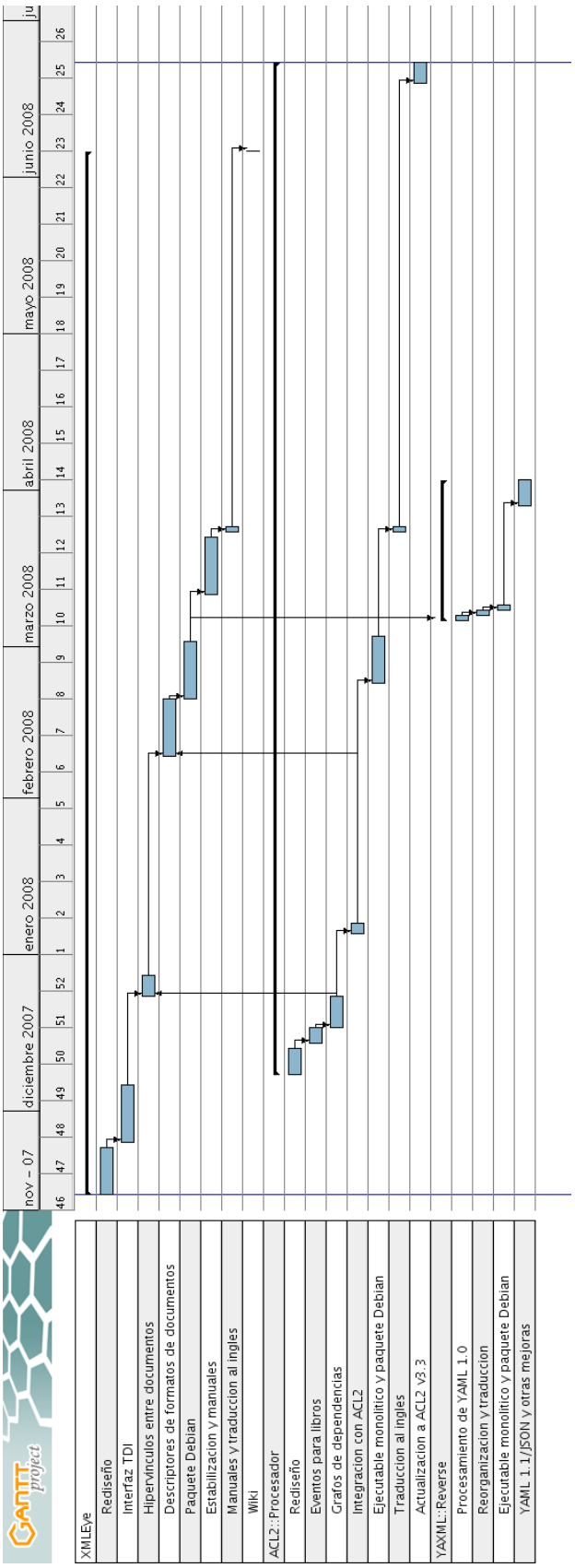


Figura 2.1: Diagrama Gantt de iteraciones

Producto	Inicio	Fin	D_I	D_R
XMLEye	15/11/2007	02/06/2008	34	53
Rediseño	15/11/2007	24/11/2007	6	8
Interfaz TDI	25/11/2007	06/12/2007	4	8
Hipervínculos entre documentos	27/12/2007	31/12/2007	2	5
Descriptores de formatos de documentos	07/02/2008	18/02/2008	6	10
Paquete Debian	21/02/2008	03/03/2008	5	10
Estabilización y manuales	09/03/2008	20/03/2008	8	8
Manuales y traducción al inglés	21/03/2008	22/03/2008	1	2
Wiki	01/06/2008	02/06/2008	2	2
`ACL2::Procesador`	08/12/2007	19/06/2008	16	27,5
Rediseño	08/12/2007	13/12/2007	2	4
Eventos para libros	14/12/2007	17/12/2007	2	3,25
Grafos de dependencias	17/12/2007	23/12/2007	3	4,25
Integración con ACL2	04/01/2008	06/01/2008	1	2
Ejecutable monolítico y paquete Debian	21/02/2008	01/03/2008	4	8
Traducción al inglés	21/03/2008	22/03/2008	1	2
Actualización a ACL2 v3.3	14/06/2008	19/06/2008	3	4
`YAXML::Reverse`	06/03/2008	02/06/2008	5	6
Procesamiento de YAML 1.0	06/03/2008	06/03/2008	1	1
Reorganización y traducción	07/03/2008	07/03/2008	1	1
Ejecutable monolítico y paquete Debian	08/03/2008	08/03/2008	1	1
YAML 1.1/JSON y otras mejoras	30/03/2008	04/04/2008	2	3

Cuadro 2.1: Cuadro de fechas y duraciones de las iteraciones

Fase del proceso	Días ideales	Porcentaje
Análisis	1,5	4,44 %
Diseño	6,25	18,38 %
Construcción	13,25	38,97 %
Pruebas	13	38,24 %
Total	34	100 %

Cuadro 2.2: Porcentajes de esfuerzo: XMLEye

Fase del proceso	Días ideales	Porcentaje
Análisis	0,5	3,125 %
Diseño	3	18,75 %
Construcción	7	43,75 %
Pruebas	5,5	34,375 %
Total	16	100 %

Cuadro 2.3: Porcentajes de esfuerzo: ACL2::Procesador

Fase del proceso	Días ideales	Porcentaje
Análisis	0,25	5 %
Diseño	0,5	10 %
Construcción	2,75	55 %
Pruebas	1,5	30 %
Total	5	100 %

Cuadro 2.4: Porcentajes de esfuerzo: YAXML::Reverse

3 Descripción general del proyecto

3.1. Perspectiva del producto

3.1.1. Entorno de los productos

El proyecto está formado por XMLEye, `ACL2::Procesador` y `YAXML::Reverse`. Los dos primeros se basan sobre las versiones finales del visor y el post-procesador presentados en el Proyecto Fin de Carrera "Post-procesador y Visor de Demostraciones del Sistema ACL2", y el tercero se halla escrito desde cero. Los tres productos tienen entidad propia, pudiendo usarse integrados a través de una sencilla interfaz o por separado.

XMLEye, `YAXML::Reverse` y `ACL2::Procesador` se hallan en continuo desarrollo, y se seguirán mejorando más allá de este Proyecto. En el caso de `YAXML::Reverse` no se añadirá nueva funcionalidad, sino que se seguirá depurando su corrección ante ejemplos más complejos y nuevas versiones de la especificación de YAML.

`ACL2::Procesador` sólo trata un subconjunto, aunque ampliado respecto a su versión original, de las demostraciones del sistema ACL2.

3.1.2. Interfaces software y hardware

La interfaz entre `ACL2::Procesador` y cualquier sistema que haga uso de su salida (por ejemplo XMLEye) está bien definida a través de su POD, que indica la forma de invocación correcta, y de su DTD (Document Type Definition), que permite conocer el formato de la salida sin tener que examinar el programa.

No existe una DTD explícita para `YAXML::Reverse`, puesto que el formato de la entrada resultante depende en gran medida del documento proporcionado: no olvidemos que YAML, el formato convertido, es un metalenguaje, al igual que XML, el formato destino.

Se ha de procurar también reducir el acoplamiento de los conversores y XMLEye con el sistema operativo y hardware al mínimo, para garantizar transportabilidad a lo largo de distintas arquitecturas y sistemas operativos.

3.1.3. Interfaz de usuario

En cuanto a la interfaz de usuario, se ha procurado hacerla lo más sencilla y flexible posible, con una simple vista que permita relacionar la estructura en forma de árbol de la demostración con información acerca del elemento actualmente seleccionado.

La integración con el resto del entorno del usuario y la adaptabilidad a sus necesidades mediante la configuración de su comportamiento permitirán al usuario sacarle el mayor provecho posible a esta aplicación.

Todas las funciones deben estar disponibles por teclado y ratón, por razones de accesibilidad y usabilidad.

3.2. Funciones

3.2.1. XMLEye

- Navegación jerárquica simultánea por múltiples documentos a distintos niveles de detalle, con vínculos entre sus elementos y con elementos de otros documentos.
- Búsqueda a lo largo del documento, con diversas opciones de filtrado.
- Integración con nuevos formatos especificados por el usuario, sin necesidad de modificar el código. Cada formato puede especificar el editor y (opcionalmente) el conversor a XML con el que deberá integrarse XMLEye.

  Los formatos son localizables a distintos idiomas, y personalizables por cada usuario del sistema. Se deben de poder restaurar los ajustes originales del formato en cualquier momento.
- Cambio en tiempo de ejecución del comportamiento de visualización del documento y sus nodos, que debe ser extensible por el usuario sin necesidad de añadir código específico a XMLEye.
- Monitorización en segundo plano del documento abierto, volviéndolo a abrir cuando se produzcan cambios en él. Esto posibilita su edición y visualización en paralelo.

3.2.2. ACL2::Procesador

- Conversión automática de un subconjunto de interés didáctico de la salida de ACL2 a XML, incluyendo demostraciones divididas entre múltiples ficheros.
- Generación de grafos de dependencias de una demostración con todos los libros usados, a cualquier número de niveles de profundidad.

- Invocación automática de ACL2 que tenga en cuenta las dependencias entre los distintos ficheros implicados y repita las demostraciones sólo ante cambios en las fuentes originales.
- Facilidad de mantenimiento y de actualización ante cambios en la salida entre versiones distintas de ACL2.
- Diversas hojas de estilo que produzcan información agregada a múltiples niveles en formato de hipertexto, manteniendo las dependencias directas e inversas entre todos los elementos, incluyendo a aquellos que pertenezcan a otros documentos.

3.2.3. YAXML::Reverse

- Conversión automática sin pérdidas ni cambios en la información de YAML a XML.
- Transformación de anclas y alias YAML a enlaces basados en identificadores únicos y atributos específicos.
- Hojas de estilo para XMLEye que tengan en cuenta los aspectos específicos de la salida producida por la conversión anterior.

3.3. Características del usuario

Podríamos establecer varios tipos de uso para este proyecto:

XMLEye integrado con `YAXML::Reverse` Esta combinación permitirá a XMLEye abrir cualquier documento escrito en un lenguaje basado en el metalenguaje YAML. Esto incluye marcadores de Firefox 3, volcados de bases de datos del entorno de desarrollo web Django, o descriptores de módulos Perl, por ejemplo.

Tiene por lo tanto un amplio abanico de posibilidades dirigido a un gran número de usuarios, que se dividirán entre los *usuarios finales* que emplearán las hojas ya existentes, y que no requerirán más conocimientos que los necesarios para instalar la aplicación y `YAXML::Reverse`, y los *usuarios avanzados*, que podrán personalizar sus visualizaciones a su antojo.

Esta funcionalidad cobrará mayor utilidad cuando se le añada a XMLEye la capacidad de publicar la visualización en efecto como una página WWW. Se podría entonces mostrar en una web de forma muy sencilla nuestros marcadores de Firefox, por ejemplo, y con más información que la exportación por defecto a HTML provee.

XMLEye integrado con `ACL2::Procesador` El usuario sólo emplea el post-procesador mediante su integración con el visor para navegar por demostraciones de ACL2. Dentro de este modelo de uso, distinguimos dos tipos de usuarios:

Estudiante Un estudiante de ciclo superior estudiando métodos formales de desarrollo de software usaría esta herramienta para ayudarle a visualizar el proceso de demostración de ACL2 sobre demostraciones sencillas.

Existen otras herramientas para ACL2 realizadas por motivos pedagógicos para este tipo de usuarios, pero se centran en aspectos distintos.

Un ejemplo es `DrACuLa`, basado en el entorno de Scheme (un dialecto de Lisp) llamado `DrScheme`. Entre algunos de los lenguajes disponibles, se puede hallar un dialecto simplificado de ACL2 extendido con unas librerías de manipulación de gráficos llamadas "teachpacks" para agilizar la enseñanza, ilustrando el hecho de que los métodos formales se pueden aplicar a proyectos software normales, y no sólo a aquellos con un alto contenido teórico.

Investigador Un investigador especializado en ACL2 preferiría utilizar una herramienta que estructurara de alguna forma la demostración en texto plano obtenida.

Podría además aprovechar la información adicional como las dependencias inversas y otros enlaces y estadísticas para realizar su trabajo de forma más eficiente.

Cuando se añadiera la posibilidad de publicar una demostración como una página WWW, podría añadir a su página personal enlaces a algunas de sus demostraciones.

Ambos tipos de usuario tienen un alto nivel de experiencia en el uso de ordenadores, por lo cual se espera que no tengan problemas a la hora de configurar su entorno, si bien no se asumen conocimientos específicos de las herramientas usadas.

Notemos que, en la actualidad, `ACL2::Procesador` sólo trata un subconjunto limitado de la entrada, y así hemos de limitarnos al uso puramente didáctico, es decir, al usuario estudiante.

Sin embargo, la intención de este proyecto es extender su soporte hasta el punto que sea útil también para un investigador. El tratamiento de demostraciones divididas a lo largo de varios ficheros implementado en este Proyecto es un primer paso en esa dirección.

Sólo XMLEye XMLEye no contiene lógica específica para ningún formato en su código Java para la visualización. Toda se halla en las hojas XSLT y los descriptores de formatos de documentos.

Añadiendo nuevas hojas, el usuario podría realizar sus propias visualizaciones de cualquier documento XML, pudiendo establecer enlaces entre los nodos o

cambiar el icono o etiqueta mostrado en el árbol del documento, entre otras cosas.

Si el formato a tratar ya se basa en XML, elaborar un descriptor de formato de documento es completamente opcional: sólo haría falta si quisiéramos utilizar una extensión o un editor distintos a los usados para XML en general.

Sólo `ACL2::Procesador` El usuario no utiliza XMLEye, y se limita a procesar de alguna forma la salida de `ACL2::Procesador`. Este tipo de usuario sería, realmente, un desarrollador que tendría cierta experiencia con ACL2 y deseara darle otra aplicación a la salida. Los autores de Mizar tenían algo similar en mente cuando cambiaron el formato de su salida a XML [52].

Un ejemplo podría ser usar una hoja XSLT para generar una versión HTML de la demostración, que se pudiera publicar en un servidor WWW, por ejemplo.

También se podría generar un documento XSL-FO (eXtensible Stylesheet Language Formatting Objects) y, usando un procesador XSL-FO, obtener una salida en formato PDF o PostScript, entre otros.

Sólo `YAXML::Reverse` Aunque están empezando a surgir validadores para YAML, como Kwalify (disponible en `http://www.kuwata-lab.com/kwalify/`), y en principio se podría en transformar un documento YAML a partir de un algoritmo *ad hoc* sobre los datos deserializados, hay muchas menos herramientas que para XML.

Combinando `YAXML::Reverse` y la versión refinada de YAXML que incluye, un desarrollador podría convertir el documento YAML a XML, aplicarle la herramienta XML en cuestión, y luego devolver el resultado XML a YAML.

La batería de pruebas de `YAXML::Reverse` se ocupará de asegurar que en todo el ciclo YAML → XML → YAML no se produzcan pérdidas ni cambios indeseados en la información del documento original.

3.4. Restricciones generales

3.4.1. Control de versiones

Al seguir un proceso incremental de desarrollo de software, se necesitó un sistema de control de versiones de todas las fuentes del proyecto.

Estos sistemas permiten almacenar todas las versiones de un árbol de ficheros, pudiendo así manipular todas las revisiones de cualquier fichero en cualquier momento.

Además de servir como una medida de seguridad contra la pérdida de información accidental, agilizan los cambios drásticos, ya que no hay que establecer medidas especiales por si fallaran: se pueden revertir los cambios hechos en cualquier momento.

En particular, `Subversion` es un sistema que trata de resolver las insuficiencias del conocido CVS (Concurrent Versions System), pudiendo mantener revisiones de directorios completos, establecer propiedades especiales sobre los elementos del repositorio y enviar nuevas revisiones de forma atómica, entre otras cosas.

Dispone de excelente documentación, con un libro [14] disponible bajo la licencia Creative Commons.

3.4.2. Lenguajes de programación

Con vistas a la transportabilidad, se eligieron dos lenguajes de programación con implementaciones interpretadas para el proyecto: Perl y Java.

Ambos lenguajes reducen significativamente el potencial de error gracias a su uso de recolectores de basura: Perl utiliza un recolector basado en recuento de referencias y la JVM (Java Virtual Machine) de Sun usa un algoritmo de barrido y marcado más avanzado.

Perl (Practical Extraction and Report Language) Este lenguaje es un candidato ideal para la labor de `ACL2::Procesador`, teniendo la implementación con mayor funcionalidad de expresiones regulares integrada dentro del propio lenguaje. En el caso de `YAXML::Reverse`, cualquier otro lenguaje con un sistema de tipos dinámico (como Ruby o Python) habría servido, pero por familiaridad con el entorno y por simplificar la instalación de ambos convertidores, se optó por el mismo lenguaje.

El rendimiento de Perl para el procesado automático avanzado de textos es alto, y se trata de una herramienta muy estable: la primera revisión de la versión de uso generalizado actual, la 5.8, se remota al 2002. La versión estable más reciente actualmente, la 5.10, fue publicada en diciembre de 2007.

Es interpretado en el sentido que el código fuente de un guión Perl es compilado a instrucciones a ser interpretadas por una máquina virtual antes de cada ejecución.

Otra ventaja es la disponibilidad en forma de módulos de todo tipo de extensiones a las capacidades básicas de Perl en el CPAN.

Java Otro requisito era poder crear una interfaz gráfica de cierta complejidad con la mayor transportabilidad posible.

A pesar de la existencia de otros marcos de desarrollo de GUI basados en lenguajes con implementaciones interpretadas como PerlGTK, PyGTK o wxPython, se deseaba además tener una plataforma integrada con soporte para XSLT sin necesidad de dependencias externas, y la capacidad de mantener un aspecto atractivo y uniforme entre plataformas.

Además, el API usado (Swing) es mucho más maduro y estable que el de cualquiera de las tres alternativas anteriores. Otra opción habría sido usar el SWT (Standard Widget Toolkit), alternativa de IBM que utiliza componentes nativos, pero su soporte de sistemas operativos distintos a Windows es inferior al de Swing.

El problema de la velocidad no es tan grave como se podría esperar, pues desde la versión 1.3 del JRE (Java Runtime Environment) la JVM o máquina virtual de Java incluye un compilador en tiempo de ejecución o JIT (Just In Time) llamado HotSpot, que convierte a código nativo y optimiza agresivamente las partes más usadas del programa de forma dinámica.

3.4.3. Sistemas operativos y hardware

Aunque tanto Perl como Java están disponibles para una gran cantidad de plataformas, para simplificar, este proyecto ha sido desarrollado y probado únicamente en Windows XP (con el Service Pack 2) y en GNU/Linux, ejecutando la versión 8.04 (Hardy Heron Long Time Support) de la distribución Ubuntu, basada en Debian, y la versión 10.3 de la distribución openSUSE.

XMLEye puede funcionar sobre cualquier entorno Java que implemente la J2SE (Java 2 Standard Edition) 5.0 como mínimo. Esto incluye evidentemente a los JRE 5.0 y 6.0 de Sun y recientemente a las versiones 6.0 y 7.0 de OpenJDK [45]. OpenJDK es una iniciativa liderada por Sun que, en combinación con los esfuerzos del proyecto IcedTea [33], ha conseguido una versión prácticamente 100 % funcional y basada en software libre de las J2SE 6.0 y la futura 7.0.

Como mínimo, existen ediciones de la J2SE 5.0 o superiores para Solaris, MacOS X 10.4, Windows 32-bits y 64-bits y Linux 32-bits y 64-bits. El soporte de 64-bits depende de la CPU usada: no se pueden usar procesadores Intel Itanium, pero sí todos los demás chips de 64 bits de AMD (Advanced Micro Devices) e Intel.

Por el lado de Perl, la transportabilidad es mucho mejor:

- Apple Mac OS Classic 8.1 en adelante.
- Windows 95/98/ME/NT/2000/XP.
- Cualquier sistema basado en UNIX (los *BSD, Linux, Solaris y Mac OS X, entre otros).
- Otros sistemas menos comunes, como: TiVo, HP-UX, Novell Netware, NonStop, Plan 9, VMS, etc.

3.4.4. Bibliotecas y módulos usados

Perl

Se ha usado la versión 5.8.8 para el desarrollo de este proyecto. La primera revisión de la versión 5.8 de Perl, la 5.8.0, introdujo diversas mejoras, pero sólo nos afecta realmente la introducción en la distribución estándar del módulo Locale::Maketext, usado para localización de los mensajes de error.

Es posible por lo tanto que los convertidores funcionen instalando manualmente dicho módulo y demás dependencias con las versiones 5.6.x, que implementan la imprescindible sintaxis orientada a objetos. No ha habido tiempo para hacer pruebas al respecto, sin embargo.

Ha sido necesario el uso de varios módulos: algunos vienen incluidos en la distribución estándar de Perl, y otros se han de instalar por separado. Estos últimos se instalan automáticamente durante la compilación de los convertidores siempre que las bibliotecas requeridas estén presentes en el sistema. Se tratan de:

File::ShareDir Permite calcular las rutas a ficheros instalados a través del mecanismo de directorios automáticos de Perl en localizaciones accesibles por todos los usuarios del sistema y acceder a ellos. De la instalación se ocupa `Module::Install`.

File::Spec Realiza transformaciones sobre rutas a ficheros de manera transportable, como convertir rutas relativas a absolutas, por ejemplo.

Getopt::Long Mediante este módulo, se pueden recibir opciones de la línea de órdenes en formato POSIX con las extensiones GNU, como la posibilidad de introducir argumentos entre las opciones. Se pueden también usar opciones en formato abreviado y reunirlas, para usuarios expertos.

Module::Install Permite escribir fácilmente instaladores de módulos Perl que sigan las mejores prácticas del CPAN, e instalen de forma automática todas las dependencias, de manera recursiva. Los instaladores resultantes son compatibles con cualquier herramienta que funcione con instaladores generados mediante el módulo estándar *de facto* `ExtUtils::MakeMaker` o con `Module::Build`.

Incorpora funcionalidades para verificar dependencias de ejecución y compilación, crear distribuciones del código, lanzar las pruebas de unidad, e instalar ejecutables y ficheros para su posterior uso con `File::ShareDir`, entre otras.

PAR Permite crear ficheros `.par` de módulos y/o guiones Perl que funcionan de forma muy similar a los `.jar` en Java: son archivos comprimidos que reúnen todo el código necesario, y pueden ejecutarse de forma muy sencilla utilizando la herramienta de línea de órdenes `parl`.

PAR::Packer Este módulo permite crear ejecutables monolíticos específicos a la arquitectura y sistema operativo actual de cualquier guión Perl. Incorpora una herramienta de línea de órdenes llamada `pp` que reúne en el mismo ejecutable los

módulos, bibliotecas y demás ficheros usados, e incluso el propio intérprete de Perl.

A diferencia de `PAR`, los ficheros generados son completamente autosuficientes, y no requieren por lo tanto de ningún proceso de instalación previo.

Pod::Usage Permite documentar fácilmente el uso correcto del programa, pudiendo usar la sección POD incrustada en el guión principal para generar ayuda en línea en formato `man` o texto puro.

Test::More Marco de pruebas de unidad basado completamente en Perl que reemplaza y extiende al módulo `Test::Simple`.

Test::Pod::Coverage Módulo ocupado de implementar pruebas de unidad sobre la cobertura de la documentación incluida en el propio código Perl en formato POD. Se asegura de que todo método público definido por todos los módulos tenga documentación asociada.

XML::LibXML Es uno de los módulos que puede emplear `XML::Validate` para implementar su funcionalidad. Provee de una interfaz Perl a la biblioteca C XML del proyecto GNOME, `libxml2` (véase `http://xmlsoft.org/`), que evidentemente ha de estar ya instalada.

XML::SemanticDiff Permite comprobar las diferencias entre dos documentos XML, notificándolas a través de una serie de manejadores de eventos. Estos manejadores de eventos pueden añadir lógica específica con información acerca de qué diferencias resultan de interés y qué otras diferencias no son tan importantes, y también pueden etiquetar dichos sucesos con información adicional.

XML::Validate Da la posibilidad de validar documentos XML fácilmente usando diversas bibliotecas externas. Esto permite usar ficheros DTD, por ejemplo. Actualmente sólo se usa en las pruebas de unidad, pero en un futuro se podría emplear para la validación de todos los resultados producidos por `ACL2::Procesador`.

XML::Writer Para salida con verificación básica de XML, usando un interfaz sencillo.

Java

Por otro lado, la amplitud de la J2SE 5.0 ha permitido implementar XMLEye limitando las dependencias externas a un *Look & Feel* libre distribuido bajo licencia BSD llamado JGoodies Looks, disponible en `https://looks.dev.java.net/`.

La implementación de JAXP (Java API for XML Parsing) 1.3 de J2SE 5.0 usa dos bibliotecas de Apache:

XSLTC 2.6.0 Implementa los estándares del W3C XSLT 1.0 y XPath 1.0. Basado en Apache Xalan, con una serie de parches de Sun.

Esta nueva versión reemplaza al antiguo motor XSLT Xalan interpretado. En este nuevo motor, toda hoja XSLT es compilada de antemano a instrucciones de la JVM, generando un *translet*, cuya ejecución es notablemente más rápida.

La limitación de XSLTC consiste en que no permite elementos de extensión, parte del estándar XSLT 1.0. Tampoco puede usar fuentes JDBC (Java DataBase Connectivity) en la transformación, ni programas incrustados escritos en otros lenguajes (Java, por ejemplo).

Xerces 2.6.2 Proporciona analizadores W3C DOM nivel 3 y SAX 2.0.2, junto con implementaciones de otras partes del JAXP 1.3. De forma análoga a XSLTC, se halla basado en la versión oficial de Apache Xerces, junto con una serie de parches seleccionados.

Esta biblioteca reemplaza a Crimson, otro analizador XML de Apache, incluido en J2SE 4.0 como parte del JAXP 1.1. Este analizador estaba centrado más en la simplicidad y bajo consumo de memoria, y se halla listado actualmente como "proyecto en hibernación" en `http://xml.apache.org`, la web de proyectos XML de Apache.

Para desarrollar XMLEye, se han usado las siguientes herramientas:

Ant 1.7.0 Sistema de gestión de tareas de compilación, empaquetado, generación de documentación y lanzamiento de pruebas de unidad, entre otras. Es similar al conocido GNU Make, siendo la principal diferencia su soporte multiplataforma, gracias a estar implementado en Java.

Eclipse 3.3.2 IDE (Integrated Development Environment) desarrollado en sus inicios por IBM que es, junto con NetBeans, uno de los más usados hoy en día. Aunque es particularmente popular en la comunidad Java, existen versiones para otros lenguajes, como C o C++. Dispone de herramientas para hacer rápidamente refactorizaciones en el código, facilitando enormemente cualquier cambio al diseño de una aplicación.

JUnit 3.8.2 Popular marco de desarrollo de pruebas de unidad, inspirado en el marco SUnit para Smalltalk, que ha originado otros marcos parecidos, como CppUnit para C++. Se ha usado la versión 3.8 y no las versiones 4.x dado que estas últimas no están aún soportadas por Ant, tras cambios en su API (Application Programming Interface).

3.5. Requisitos para futuras versiones

3.5.1. XMLEye

- Filtrado de las hojas de visualización y preprocesado según el tipo de documento actualmente abierto.

- Búsqueda mediante consultas XPath arbitrarias para usuarios avanzados. Marcado de nodos coincidentes en una búsqueda en la vista de árbol.
- Creación de nuevos tipos de documento en la propia interfaz.
- Crear nuevas hojas para otros formatos basados en XML, como las trazas de ejecución del motor XSLT Saxon, los ficheros de proyecto de GNOME Planner, o las demostraciones de los sistemas Mizar [52] o Prover9 (`http://www.cs.unm.edu/~mccune/mace4/`).
- Cambio de la biblioteca Xalan a Saxon, pasando de XSLT 1.0 a XSLT 2.0, que incorpora muchas nuevas funcionalidades.
- Mejora del motor XHTML/CSS (Cascading Style Sheet(s)). Se evaluarán los motores del proyecto Lobo (ver `http://lobobrowser.org/`), que añade soporte para JavaFX, y del proyecto Flying Saucer (`https://xhtmlrenderer.dev.java.net`), aparentemente más maduro y fácil de integrar. JavaFX podría ser un nuevo formato de visualización que permitiría crear interfaces Swing completas.
- Integración de lenguajes de *scripting* en el interior de las hojas XSLT, para poder implementar funcionalidad adicional de manera más sencilla y sin modificar XMLEye.
- Publicación de documentos con visualizaciones XHTML del árbol y los nodos seleccionados, posiblemente integrando capacidades de búsqueda. Idealmente, publicar el documento debería producir una carga mínima en la máquina que lo sirviera.
- Creación de un ejecutable para Windows mediante Launch4J (`http://launch4j.sourceforge.net/`).
- Creación de una versión lanzable desde el navegador mediante Java WebStart.
- Envío de los paquetes Debian al repositorio oficial.
- Traducción completa del código al inglés.

3.5.2. ACL2::Procesador

- Extracción de información de la salida de mayor variedad de eventos, como `defaxiom` o `verify-guards`, y de órdenes que se expandan a eventos, como `make-event` o cualquier uso de una macro Lisp.
- Mayor cantidad de información generada automáticamente, como por ejemplo sumarios globales.
- En paralelo con el requisito anterior, aumento de la variedad y potencia de las hojas de visualización y preprocesado.
- Integración con la ayuda HTML de ACL2.

- Envío de los paquetes Debian al repositorio oficial.
- Envío de la distribución al repositorio CPAN.
- Traducción completa del código al inglés.

3.5.3. YAXML::Reverse

- Implementación de hojas de visualización y preprocesado para diversos lenguajes basados en YAML.
- Depuración del procesado de anclas, alias y etiquetas.
- Soporte de YAML 1.2 cuando quede finalizado (actualmente se halla en estado de borrador).
- Envío del paquete Debian al repositorio oficial.
- Envío del módulo al repositorio CPAN.

4 Desarrollo del proyecto

Procederemos a describir el análisis, diseño, implementación y realización de pruebas del proyecto como un todo, considerando la separación de `YAXML::Reverse` y `ACL2::Procesador` respecto de XMLEye como un detalle de diseño, motivado por el deseo de aprovechar los puntos fuertes de cada lenguaje y hacer tanto al visor como a los conversores reutilizables.

En particular, las secciones de diseño y análisis describirán la última iteración de cada producto, al contener éstas la arquitectura y diseño definitivos, que también han ido cambiando y han ido siendo refinados a lo largo de las iteraciones.

4.1. Proceso

Se ha seguido la metodología XP (eXtreme Programming) en el desarrollo de este proyecto. Fue creada por Kent Beck, Ward Cunningham y Ron Jeffries durante el desarrollo del C3 (Chrysler Comprehensive Compensation System), un sistema unificado de gestión de nóminas.

La parte “Extreme” de XP consiste en llevar prácticas conocidas y recomendadas de la ingeniería de software a su máxima expresión. Por ejemplo, si las pruebas son buenas, entonces todo el código debería tener pruebas, y además éstas deberían estar automatizadas y ejecutarse continuamente.

4.1.1. Origen

Su popularización se dio a través de la participación conjunta de estos tres autores, con intervenciones del resto de la comunidad, en la web [23], fundada en 1995.

En 1999 su popularidad se vio aumentada en gran medida tras la publicación de la primera edición de “Extreme Programming Explained”.

Para este proyecto, se siguió la segunda edición [5], publicada en 2004, que en respuesta a las críticas recibidas introdujo importantes cambios y reorganizaciones, como la sustitución de las 12 prácticas por un núcleo de prácticas obligatorias apoyado por una serie de prácticas opcionales.

Ya se mencionó anteriormente en §2.5 otra obra relacionada con XP ([6]).

4.1.2. Características

XP es una metodología ágil de desarrollo de software, aplicable por lo general a proyectos de pequeño y medio tamaño, en un equipo de hasta 12 personas.

En los métodos ágiles, se descarta todo documento o procedimiento innecesario para el proyecto, permitiendo al equipo avanzar rápidamente concentrándose en lo importante y sin perder tiempo en formalismos.

En XP, los detalles se comunican en conversaciones directas entre las partes implicadas (incluyendo al cliente), aclarando confusiones y dudas en el momento. Así, XP se dirige más a las personas que al proceso, y más a la obtención de software útil que al seguimiento de un proceso rígido y burocrático.

En cada iteración, se entrega una versión funcional del software que el cliente puede probar para ampliar la información disponible en la siguiente iteración.

Una idea que se suele tener respecto a XP (por aquello de que es "Extreme") es que no se realiza documentación en absoluto, sea lo que sea. El requisito de documentación externa al proyecto es muy común, por ejemplo, y debe ser atendido. Ron Jeffries trata de corregir éste y otros malentendidos en [37].

A diferencia de otras metodologías, donde el cliente es una entidad externa al proyecto con el que se sólo se comunica el analista, en XP es una parte integral del equipo. Se ocupa de formular los requisitos a nivel conceptual, darles su prioridad, y resolver dudas que pueda tener el resto del equipo.

XP nos recuerda así que el software existe para dar un valor añadido al cliente, y no por sí mismo. Así, las decisiones acerca de la funcionalidad deseada son del cliente, y las decisiones técnicas y el calendario lo establecen los desarrolladores.

Valores

Aquí se describen algunos de los valores que todo proyecto basado en XP debe tener en mente:

Simplicidad La solución más sencilla suele ser la mejor. Esto no quiere decir que se use la primera solución que se nos ocurra. La solución escogida debe:

- Hacer todo lo esperado.
- No contener ninguna duplicación.
- Pasar todas las pruebas.
- Tener el mínimo número de elementos.

Lo que este valor intenta evitar es el diseño excesivo al tratar de resolver problemas aún no planteados. Dicho trabajo podría ser inútil o incluso contraproducente si el cliente cambiara de opinión acerca del resto de los requisitos aún no implementados. No se trata de anticipar el cambio, sino de adaptarse a él.

Comunicación Consiste en la comunicación transparente y sin obstáculos entre todas las partes del equipo, incluido el cliente. La falta de comunicación induce a problemas o malentendidos que podrían haberse resuelto fácilmente de lo contrario.

Aprendizaje Más que intentar capturar toda la información de una vez, un proyecto basado en XP debe de concentrarse en el día a día, aprendiendo paso a paso qué es lo que se espera exactamente del programa, dado que ni siquiera el cliente lo sabe con seguridad.

Dicha información puede venir de muchas fuentes: del cliente, de la experiencia propia, de las pruebas automatizadas, etc. Otras veces, cuando no se sabe cómo atacar un problema, se puede aprender realizando experimentos y evaluando varias alternativas, para así poder elegir la mejor y evitar discusiones improductivas.

Principios

Para seguir esos valores, se proponen una serie de principios, que se concretan a través de una serie de prácticas recomendadas. Dichos principios incluyen por ejemplo:

Flujo Se deben de realizar constantemente todas las actividades del desarrollo de software: análisis, diseño, implementación, integración y pruebas.

Esto evita la acumulación de riesgos a la hora de integrar y validar, realizando entregas frecuentes con pequeños riesgos e incrementos de funcionalidad, más que una única entrega con alto riesgo y toda la funcionalidad.

Margen En todo plan, se debe de tener margen para en el caso de no tener suficiente tiempo poder dejar algunas historias para posteriores versiones.

Éste ha sido el caso de sobre todo `ACL2::Procesador`, que sólo procesa un subconjunto limitado de la salida de ACL2, pero es completamente funcional dentro de dicho subconjunto, y mantiene un diseño que facilitará futuras ampliaciones.

Calidad La calidad no es algo opcional, y disminuir el nivel aceptado de calidad no garantiza una reducción del tiempo de desarrollo. De hecho, suele aumentarlo.

El control del tiempo de un proyecto no debe ser así basado en su calidad, sino sobre todo en su alcance. Es mejor hacer una sola cosa bien que varias mal.

Centrado en las personas Son las personas las que realizan el software, no la metodología de desarrollo.

Una metodología de desarrollo de software ha de tener en cuenta las necesidades y limitaciones de las personas. En una labor fundamentalmente creativa como es el desarrollo de software, se necesita que el equipo trabaje a su máximo rendimiento de forma sostenible.

Prácticas

Las prácticas de XP se dividen en dos partes: un núcleo de reglas que todo proyecto XP debe cumplir, y otras prácticas recomendadas pero no obligatorias, que suelen tener como requisito el cumplimiento de todas las reglas principales.

En el caso de este Proyecto, hay tareas que sencillamente no se han podido seguir, como "Programación en Parejas", o "Trabajar Juntos" (reunir al equipo de desarrollo en un lugar), por razones obvias.

Algunas de las prácticas seguidas son:

Historias Las historias de usuario son algo completamente distinto de los casos de uso. Describen una funcionalidad o propiedad deseada del sistema, en palabras del usuario y no del desarrollador. Mediante ellos se pueden expresar todo tipo de requisitos, no sólo los funcionales, y en ellos se basan las pruebas de aceptación.

Por otro lado, un caso de uso representa de forma abstracta la interacción entre el sistema y los demás entes externos, incluyendo el actor principal y los secundarios.

XP no prohíbe el empleo de los casos de uso. De hecho, se pueden usar en combinación, sirviendo el caso de uso como una generalización de una o varias historias de usuario particulares.

Así, puede verse que una historia de usuario es algo mucho más concreto, generalmente pequeño y fácilmente estimable, mientras que un caso de uso se podría ver como una clase de interacciones concretas que el sistema ha de posibilitar. Ambos describen el *qué*, pero lo hacen bajo distintos enfoques.

El equipo de desarrollo puede realizar estimaciones del esfuerzo que requeriría una historia y presentárselas al cliente, dándole más información para decidir qué historias quiere implementadas en cada iteración. Puede que se decida posponer una historia, o dividirla en varias.

En mi caso, los "clientes" han sido mis directores de proyecto, especialistas en ACL2, y por lo tanto usuarios potenciales del proyecto.

Integración Continua La prueba e integración de los cambios se realiza de forma continua y a pequeños pasos. Esto se sigue del hecho de que corregir los fallos introducidos en el software es tanto más complicado cuanto más amplios sean los cambios.

Ciclo Semanal La planificación se ha realizado en ciclos cortos, de duración aproximada de una semana, y se ha mantenido una comunicación semana a semana con los "clientes", en este caso los directores del proyecto.

Compilación en 10 Minutos Aunque para un proyecto de este tamaño no sea una tarea difícil de realizar, se puede considerar un requisito para "Integración Continua".

La idea consiste en mantener siempre el tiempo de compilación y ejecución de todas las pruebas automáticas pertinentes por debajo de 10 minutos, para poder realizar integración continua del código de todo el equipo y evitar tanto problemas de integración de última hora como esperas innecesarias.

Programación Basada en Pruebas Los componentes más importantes de XMLEye han sido desarrollados de esta forma usando JUnit, en cuyo desarrollo participó el propio Kent Beck. Existen otras herramientas para realizar pruebas de unidad: la obra [9] describe las más conocidas en Java.

En este método de programación, se escribe primero una prueba que demuestra que la funcionalidad deseada está implementada. Una vez está escrita la prueba, se comprueba que ésta falle, tras añadir el código estrictamente necesario para que todo compile.

Es entonces cuando se añade el código necesario a la implementación para hacer que se superen dicha prueba y las anteriores.

Además de garantizar mayor corrección, el uso de este estilo obliga al desarrollador a estructurar su código para que sea fácil de probar, haciéndolo modular y cohesivo. Refactorizaciones posteriores en el diseño pueden garantizar inmediatamente la misma funcionalidad que la versión anterior.

En el caso de `ACL2::Procesador`, se ha seguido un proceso similar:

1. Se decide el enunciado Lisp a tratar junto con su salida de ACL2.
2. Se define a grandes rasgos el resultado XML deseado a partir de la salida de ACL2.
3. Se lanza el post-procesador sobre el enunciado y la salida.
   - 3a. Se obtiene el resultado deseado: se continúa con el proceso.
   - 3b. No se obtiene el resultado deseado: se refina el post-procesador y se vuelve al paso 3.
4. Se marca el resultado XML como válido.
5. Se lanzan las pruebas de regresión, comparando automáticamente cada resultado XML obtenido por la versión actual del post-procesador con las últimas versiones validadas:
   - 5a. No hay diferencias: se escoge otro enunciado y su salida y se repite el proceso.
   - 5b. Hay diferencias: si constituyen regresiones, se corrigen y se vuelve al paso 5. En caso de falso positivo, se refina la comparación lo estrictamente necesario para evitarlo si se puede, o de lo contrario se marca la salida XML como válida y se reemplaza a la anterior.

El proceso no difiere mucho de la programación basada en pruebas usual: se definen los resultados deseados y entonces se añade la funcionalidad necesaria. La diferencia radica en que la prueba es de todo el post-procesador, y no de una clase o método, como las usuales pruebas de unidad.

El proceso completo se halla dirigido a través de los mecanismos usuales de pruebas de unidad empleados para cualquier módulo del CPAN, como `Test::More` o `Test::Simple`.

`YAXML::Reverse` hace algo muy parecido a `ACL2::Procesador`: toda mejora se realiza tras añadir un nuevo documento YAML 1.1 o JSON al conjunto de casos de prueba. Cada caso de prueba es ejecutado de la siguiente forma:

1. El documento origen se convierte a XML mediante `YAXML::Reverse`.
2. El documento XML resultante se convierte de nuevo a YAML mediante una versión mejorada de la hoja XSLT de YAXML.
3. Se deserializa el documento original y el documento final y se comprueban que las dos estructuras de datos resultantes son iguales elemento a elemento.

Diseño incremental Las metodologías tradicionales, siguiendo el modelo de ciclo de vida de cascada, intentaban realizar el análisis y diseño de antemano, siguiendo el modelo de la ingeniería tradicional.

Se realizaba la analogía entre la construcción de una casa y la de un proyecto software: una vez estaban implantados los cimientos, cambiar la estructura del edificio era varios órdenes de magnitud más caro.

XP afirma lo contrario: el diseño del programa no es algo que se haga una sola vez y quede fijo por el resto de la vida del programa. Se debe refinar continuamente en función de las necesidades del momento.

Para evitar el aumento del coste de las modificaciones, se dispone de otras prácticas además de ésta, como el uso de pruebas automáticas, la compartición de código, la programación en parejas y la comunicación fluida con el resto del equipo.

Así evitamos tanto el diseño excesivo, que supone una pérdida de tiempo, como la falta de diseño (conocida como el anti-patrón *Bola de Barro* [55]), que dificulta la introducción de cambios.

4.2. Herramientas de modelado usadas

4.2.1. BOUML

Genera código Java/C++/Python/Perl a partir de diagramas UML (Unified Modelling Language), realiza ingeniería inversa de Java/C++, y permite dibujar diversos diagra-

mas UML 2.0, como los diagramas de clases, despliegue, componentes o casos de uso, entre otros.

Está disponible para Windows, GNU/Linux y Mac OS X en la web http://bouml.free.fr. Se trata de un programa escrito en C++ usando la versión 3.3.8 de la biblioteca Qt de la compañía Trolltech, conocida por ser la biblioteca sobre la cual se halla implementado el gestor de escritorio KDE.

4.2.2. UMLet

Esta otra herramienta, a diferencia de BOUML, no trata de ser un CASE, sino una simple herramienta para dibujar diagramas.

Está diseñada para ser flexible y muy fácil de aprender y usar. Dibujar un diagrama consiste en utilizar componentes de una serie de paletas personalizables, pudiendo modificar propiedades editando un pequeño bloque de texto asociado a cada elemento.

Se ha usado para cubrir algunas carencias de BOUML, como por ejemplo en el diagrama de arquitectura del sistema.

El programa se halla disponible como una aplicación Java independiente o como un plug-in para Eclipse en http://www.umlet.com.

4.3. Requisitos

4.3.1. Interfaces externas

En este apartado se describen los requisitos que deben cumplir las interfaces con el hardware, el software y el usuario.

- La visualización permitirá a un usuario acceder directamente a nodos relacionados del árbol del documento.
- La visualización debe de usar texto con formato, para proporcionar mayor información visual que el texto puro.
- Las operaciones de larga duración no deben de afectar al tiempo de respuesta de la interfaz gráfica.
- El estado de la interfaz mostrado al usuario debe de ser siempre coherente con el estado interno de la aplicación.
- El sistema operativo ha de posibilitar el lanzamiento y monitorización de otros procesos en segundo plano.

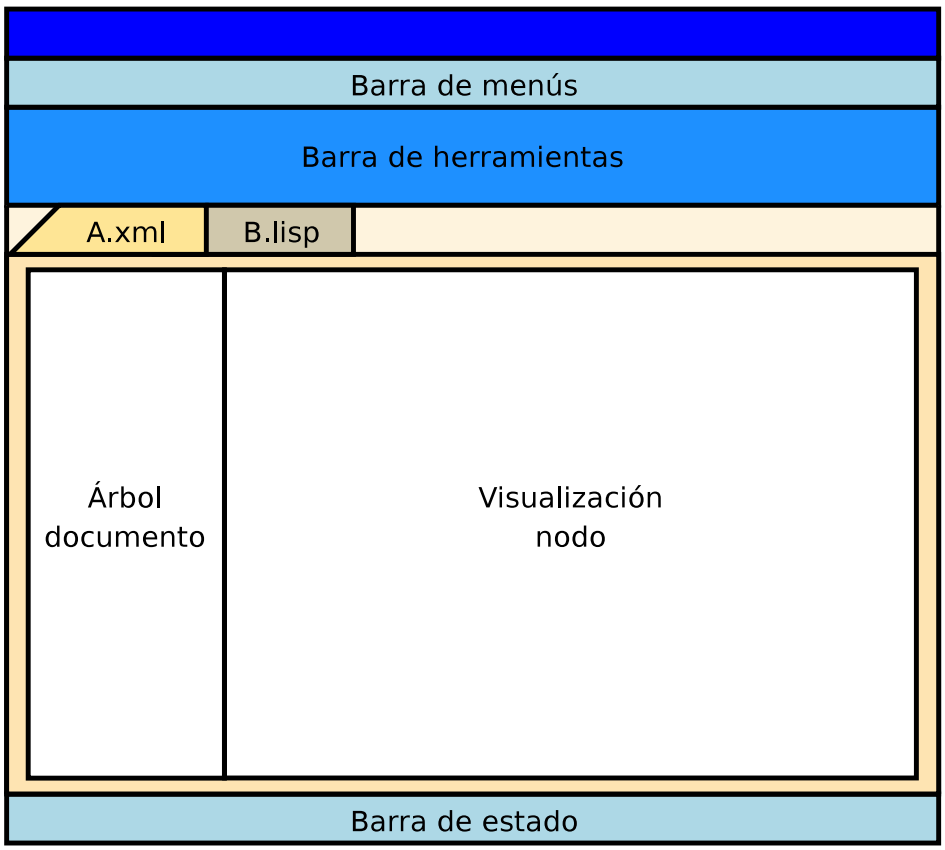


Figura 4.1: Prototipo de la ventana principal

- XMLEye ha de ser capaz de adaptarse a los distintos esquemas de nombrado de ficheros disponibles en cada sistema operativo.
- El formato de salida de `ACL2::Procesador` debe estar bien definido.

Se adjunta un prototipo de la ventana principal en la figura 4.1 de la página 62. Puede verse cómo, en esta nueva versión de XMLEye, se integrará el soporte para visualización simultánea de varios documentos en pestañas.

También hay prototipos para los diálogos:

- Búsqueda: figura 4.2, página 63.
- Gestión de descriptores de formatos de documentos: figura 4.3, página 63.
- Información del producto: figura 4.4, página 64.

4.3.2. Funcionales

- Generar ficheros XML que engloben la información de una fuente Lisp, su demostración por ACL2 asociada y referencias a todas las otras fuentes Lisp (libros

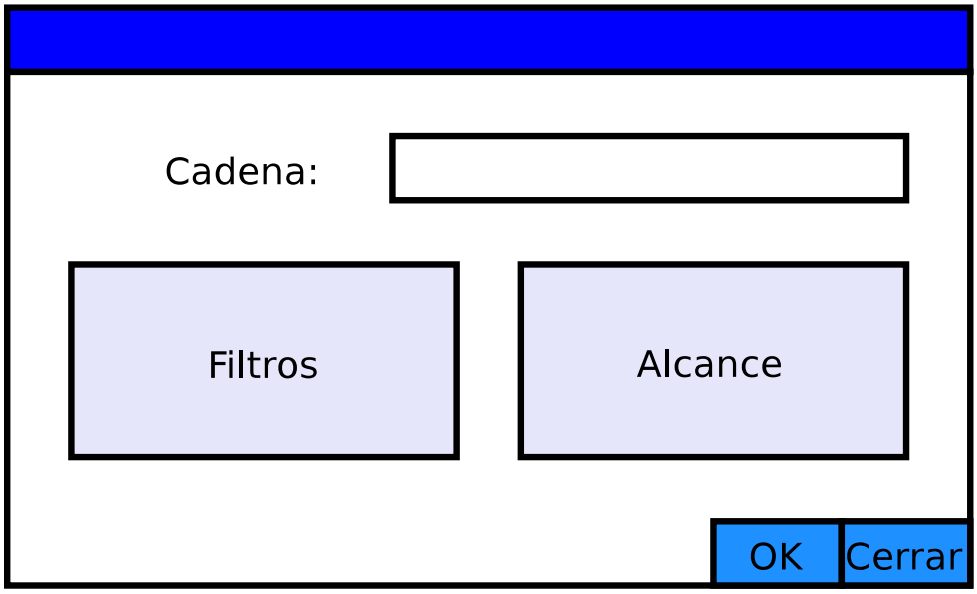


Figura 4.2: Prototipo del diálogo de búsqueda

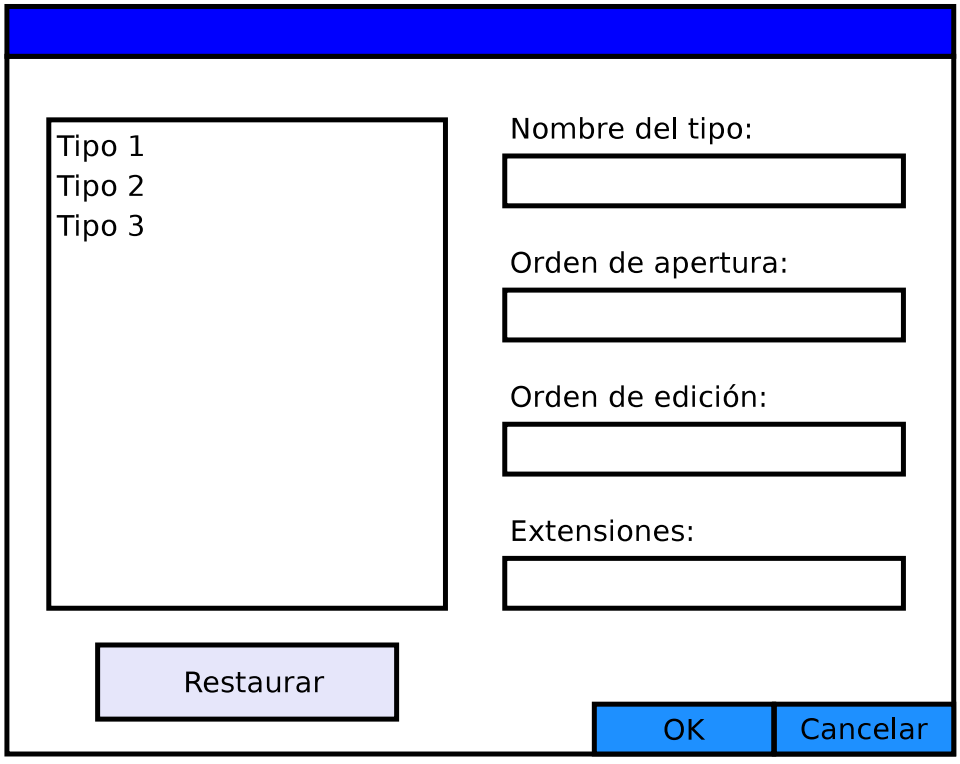


Figura 4.3: Prototipo del diálogo de gestión de descriptores de formatos de documentos

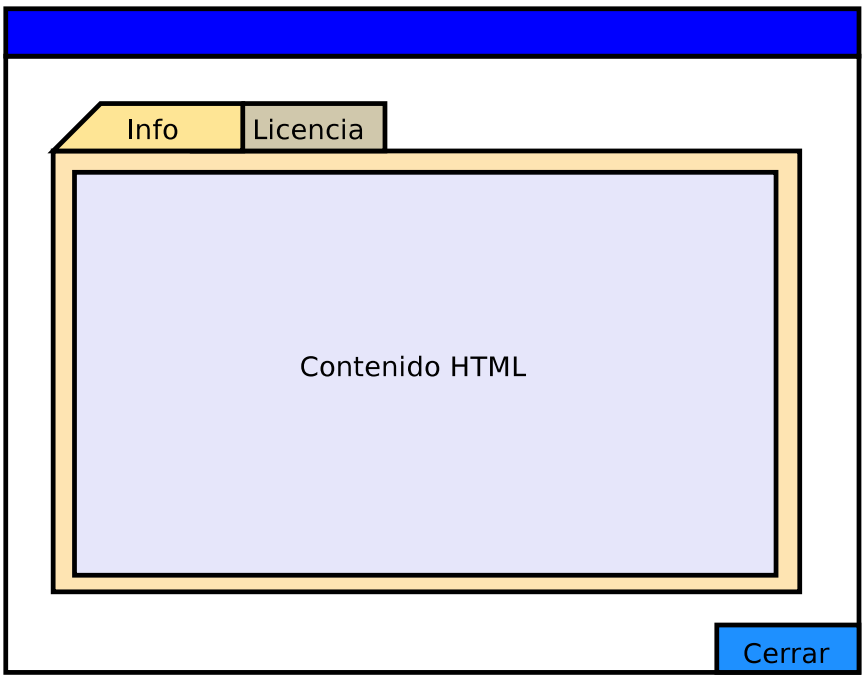


Figura 4.4: Prototipo del diálogo de información del programa

de ACL2) de que dependa.

- Permitir la visualización de documentos pertenecientes a lenguajes basados en YAML 1.1, y la posterior creación de hojas específicas a éstos.
- Navegar por múltiples documentos a la vez, abstrayendo al usuario si éste lo desea de los detalles de la estructura de sus versiones XML.
- Mantener un historial de documentos recientes, fácilmente accesible por el usuario.
- Permitir editar los documentos visualizados con el editor favorito del usuario para el formato correspondiente, si están presentes en el sistema. Puede que en ciertos casos sólo dispongamos del XML final.
- Regenerar cualquier documento abierto si se producen cambios en él en algún momento, pudiendo mantener el editor y el visor abiertos a la vez.
- Relacionar los elementos del documento entre sí, y con elementos específicos de otros documentos.
- Generar automáticamente información adicional que pueda ser de interés para el usuario.
- Realizar búsquedas por el documento, sin pérdidas de información respecto del texto original, usando diversos filtros y pudiendo limitar su alcance.
- Repetir la última búsqueda realizada.

- Personalizar las órdenes usadas para lanzar el editor o el conversor de cualquiera de los formatos aceptados, y volver a los ajustes por defecto en cualquier momento.
- Cambiar el tipo y/o formato de la información mostrada durante la ejecución del programa.

4.3.3. Atributos del sistema software

En este proyecto, la mantenibilidad es fundamental para que posteriores ampliaciones y correcciones se hagan de forma rápida y sencilla, dada la complejidad de la herramienta cuyo uso se pretende facilitar.

También sería deseable la separación entre la lógica de visualización y el resto del sistema por la posibilidad de que los usuarios (véase §3.3) extiendan sus capacidades de visualización sin tener que modificar el código fuente del proyecto, sólo modificando o añadiendo ficheros de datos. Separar la lógica de conversión de XMLEye de cuestiones específicas a `ACL2::Procesador` posibilitará su uso para formatos distintos a demostraciones de ACL2, y que tampoco tengan por qué ser basados en XML.

La transportabilidad, aunque no imprescindible, se corresponde con el deseo de permitir un libre uso de este proyecto como una herramienta más para los usuarios de ACL2 que limite en lo mínimo posible su elección de sistema operativo.

Con vistas a ampliar la base de usuarios potencial de XMLEye y sus extensiones, cobrará importancia no solo la facilidad de uso, sino también la facilidad de instalación, controlando el número de dependencias externas y, cuando esto no sea posible, reduciendo su impacto en la complejidad de la instalación.

4.4. Análisis del sistema

4.4.1. Historias de usuario

Como ya se comentó en §4.1.2 (página 58), los proyectos basados en XP se estructuran en torno a las historias de los usuarios, que describen un comportamiento o propiedad deseada del sistema en sus propias palabras.

Mientras se redactan, el equipo de desarrolladores, tras realizar un análisis superficial del trabajo implicado con ayuda del cliente, asigna a cada historia un tiempo estimado de finalización en días ideales y un riesgo. El cliente puede, en función de esos factores y de sus intereses, marcar la prioridad de una historia.

Una historia de usuario puede ser de muy alto nivel o de muy bajo nivel, pudiendo unirse varias historias de bajo nivel en una sola, o dividirse una de alto nivel en varias más sencillas.

Pasaremos a listar las historias de usuario que se han ido acumulando a lo largo de la comunicación con los clientes durante el desarrollo del PFC, es decir, mis directores de proyecto, usuarios de ACL2 y por lo tanto partes interesadas.

En los casos de `ACL2::Procesador` y `YAXML::Reverse`, hay varias historias pospuestas para futuras iteraciones, sin que ello impida entregar una versión funcional (recordemos el principio "Margen" de XP).

Dividiremos la mayoría de las historias en tareas, dando estimaciones iniciales del tiempo necesario para cada una. Otras serán lo bastante concretas como para no necesitar ninguna subdivisión.

En esta lista de historias de usuario no se incluyen aquellas que se completaron durante el Proyecto Fin de Carrera "Post-procesador y Visor de Demostraciones del Sistema ACL2".

XMLEye

1. "Añadir soporte para visualizar varios ficheros a la vez y enlazarlos entre sí."

   Esta historia de usuario es muy corta para todo el trabajo que implica:

   a) Concentrar el estado de visualización del documento actual en un modelo de presentación, para así poder cambiar entre los estados de cada documento: 3 días ideales. Completada.
   b) Integrar de nuevo y depurar toda la funcionalidad de la aplicación con este nuevo diseño: 3 días ideales. Completada.
   c) Añadir una interfaz TDI a XMLEye, y asegurar su integración con el *Look and Feel* instalado: 4 días ideales. Completada.
   d) Añadir la posibilidad de establecer hipervínculos entre varios documentos: 2 días ideales. Completada.

2. "El visor no debería saber absolutamente nada de ACL2. Nada en su interfaz gráfica ni en su implementación nos debe recordar que podemos trabajar con ACL2. Pero sí podría saber algo de un procesador externo que tradujera ficheros de un formato, del que el visor tampoco sabe nada, a XML. También podría conocer la existencia de hojas de estilo XSLT y de herramientas externas, como un editor, como de hecho ocurre ahora."

a) Mover toda la lógica propia de ACL2 a `ACL2::Procesador`, y retirarla de XMLEye: 0,5 día ideal para retirarla. Véase la historia 5 de `ACL2::Procesador` para la estimación de su recreación. Completada.

b) Definir la estructura y contenidos de un descriptor de formato de documento: 0,5 día ideal. Completada.

c) Implementar y depurar la estructura en 2 niveles del repositorio de descriptores, empleando ciertas variables del entorno si se hallan definidas: 2 días ideales. Completada.

d) Internacionalizar las cadenas contenidas en los descriptores: 0,5 día ideal. Completada.

e) Integrar la información de los descriptores con los controles de apertura y edición de documentos: 2 días ideales. Completada.

f) Crear un diálogo de gestión de los descriptores instalados: 1 día ideal. Completada.

3. "Vigilar directamente el fichero fuente en caso de cambios, en vez de solamente cuando se cierra el editor. Así, si uno abre el editor, hace un par de cambios y guarda (sin cerrar el editor), también se actualizará. En caso de que se desactive temporalmente la reimportación automática, también debería funcionar correctamente si en ese intervalo se hicieron cambios."

Esta historia implica:

- Resolver unas condiciones de carrera conocidas que producían fallos intermitentes en la reimportación automática. Tiempo estimado: 1 día ideal. Completada.
- Refinar el diseño del momento para evitar que al reabrir el documento, apareciera como una nueva pestaña. Tiempo estimado: 1 día ideal. Completada.
- Implementar la detección y notificación con un hilo pausable de baja prioridad en segundo plano, activado de forma intermitente. Tiempo estimado: 1 día ideal. Completada.

ACL2::Procesador

1. "Normalizar la estructura de `ACL2::Procesador` para que siga la de un módulo del CPAN y se porte mejor a la hora de en el futuro hacer un paquete Debian."

Tiempo estimado: 2 días ideales. Completada.

2. "Llevar la cuenta de en qué paquete (espacio de nombres) estamos trabajando. Puede saberse por el indicador (prompt)."

Se requeriría añadir el filtrado de la orden `in-package` y del indicador de ACL2, modificando en consecuencia la salida XML.

Tiempo estimado: 1 día ideal. Completada.

3. "Tratar el ejemplo de las Torres de Hanoi de [61], dividiéndolo en tres partes: `books/hanoi.lisp`, libro que define el algoritmo para resolver el problema y todo lo necesario excepto el propio teorema del número de intercambios; `books/hanoi.acl2`, que se encarga de definir el paquete y certificar el libro; y `hanoi-use.lisp`, que incluye el libro y contiene el teorema con el número de intercambios."

Una vez la capacidad de visualizar y enlazar entre sí varios documentos esté añadida en XMLEye (véase su historia 1), habría que:

- Crear un módulo de detección de dependencias que utilice un simple análisis superficial de sus órdenes, de forma recursiva, y genere un índice de cada libro con los identificadores de los eventos que define y los libros de que depende a su vez. Tiempo estimado: 3 días ideales. Completada.
- Implementar el procesamiento de la salida de los eventos `include-book`, `certify-book`, `defpkg` e `in-package`. Tiempo estimado: 2 días ideales. Completada.

4. "Mejorar los informes de errores de las pruebas de regresión, evitando que fallen con diferencias insignificantes."

Requiere:

- Volver a implementar las pruebas de regresión, esta vez calculando la diferencia entre los documentos no a partir de las líneas de texto que lo componen, sino usando el árbol DOM resultado de analizar los dos documentos XML. Tiempo estimado: 2 días ideales. Completado.
- Implementar el código necesario para ignorar aquellas diferencias que no sean de interés. Tiempo estimado: 1 día ideal. Completado.

5. "Integrar `ACL2::Procesador` con ACL2, haciendo que automáticamente genere las salidas y las sitúe en ficheros `.out` y `.xml` en el mismo directorio, realizando una gestión de dependencias al estilo GNU Make entre ellos y la fuente Lisp. Así evitaríamos importar varias veces el mismo fichero, y se podría simplificar el visor."

Una vez esté terminada la tarea 3, habría que modificar el código que recibe el texto de la demostración de ACL2 y lanza el proceso de análisis para que invoque él mismo a ACL2 en caso de que el fichero `.xml` no exista o se halle desactualizado respecto a la fuente Lisp.

Tiempo estimado: 1 día ideal. Completada.

6. "Manejar los eventos definidos por el usuario con `make-event` y las expansiones de macros creadas con `defmacro`."

No está del todo clara la forma de resolver esta historia de usuario. Un análisis superficial sugiere esta división en tareas:

- Instrumentar el código Lisp con llamadas previas a `:trans1` sobre la orden desconocida, sin que se produzcan modificaciones en el espaciado de cualquiera del resto de las líneas del documento. Tiempo estimado: 2 días ideales. Pospuesta.
- Usar la orden resultante de `:trans1` como el enunciado de la orden para la siguiente salida. Tiempo estimado: 2 días ideales, por posibles cambios importantes a realizar en el diseño. Pospuesta.
- Registrar de alguna forma la expansión producida por `make-event`, tomando como base el fichero fuente `basic.lisp` del libro `make-event` incluido en la distribución de ACL2. Tiempo estimado: 3 días ideales. Pospuesta.

7. "Tratar el caso en que un `include-book` aparezca dentro de un `local` (y entonces lo que se incluye es local al libro, no se ve fuera). Y esto puede aparecer dentro de un `encapsulate`".

   Hay que refinar la sintaxis de los grafos de dependencias para incluir menciones al ámbito de validez de cada entrada: puede que dentro de un `encapsulate` se haga referencia a un evento `P::F`, y en el nivel global u otro `local` o `encapsulate`, se dependa de otro evento distinto con el mismo nombre. Limitando el alcance de las definiciones se podría tratar este problema.

   Tiempo estimado: 3 días ideales. Pospuesta.

8. "Tratar las *forcing rounds*, demostraciones realizadas tras la demostración principal sobre cualquier hipótesis cuya aplicación se hubiera forzado durante la demostración anterior (la demostración inicial, u otra *forcing round*)."

   Hay que refinar la división jerárquica de las metas y procesar la salida específica a las *forcing rounds* estableciendo los enlaces correspondientes.

   Tiempo estimado: 3 días ideales. Pospuesta.

9. "Tratar todas las opciones que pueden acompañar a un `defthm` y a un `defun`."

   En el caso de `defthm`, se han de tratar `:rule-classes`, `:instructions`, `:hints`, `:otf-flg` y `:doc`.

   Para `defun`, se han de tratar las declaraciones y la cadena de documentación (del estilo del argumento de `:doc`).

   Esta historia se habría de dividir entonces en varias tareas independientes:

   a) Soporte para cadenas de documentación, `:rule-classes` y `:otf-flg`: 2 días ideales. Completada parcialmente.

   b) Soporte de `:instructions`: 2 días ideales. Pospuesta.

c) Soporte de `:hints`: 2 días ideales. Pospuesta.

d) Soporte de declaraciones (incluye `:hints` e `:instructions`): 2 días ideales. Pospuesta.

10. "Marcar las salidas de los nuevos apuntes de Moore [44], *Recursion and Induction*."

El trabajo necesario para implementar esta historia de usuario es extensivo, requiriendo implementar todas las historias anteriormente propuestas, y añadir la capacidad a `ACL2::Procesador` de tratar macros Lisp.

Tiempo estimado: 6 días ideales. Pospuesta.

YAXML::Reverse

1. "Abrir documentos YAML en XMLEye sin que suponga una pérdida o alteración de la información disponible."

   - Seleccionar un módulo Perl que permita cargar documentos YAML. Idealmente, debería ser Perl puro, pero la velocidad de carga y la fiabilidad son más importantes. Tiempo estimado: 0,5 día ideal. Completada.
   - Definir un primer prototipo del módulo que verifique la viabilidad del enfoque usado con algunos ejemplos sencillos: 0,5 día ideal. Completada.
   - Convertir el prototipo en un módulo completo al estilo del CPAN, con pruebas de unidad basadas en Perl que verifiquen la conservación de la información: 1 día ideal. Completada.

2. "Procesar el ejemplo de la factura de la web [60] de YAXML."

Hay que establecer un mecanismo para detectar la existencia de anclas y alias en la estructura de datos generada en memoria, evitando tener que volver a analizar el código fuente YAML. Podrían buscarse referencias duplicadas, por ejemplo. Tiempo estimado: 1 día ideal. Completada.

3. "Preparar hojas de estilos para los marcadores de Firefox y volcados del entorno de desarrollo web Django."

   - Comprobar que la biblioteca usada analiza correctamente documentos del subconjunto JSON de YAML 1.1 que emplean los volcados de Django y los marcadores de Firefox, y en caso contrario reemplazar la existente. Tiempo estimado: 1 día ideal. Completada.
   - Comprobar que se procesan debidamente documentos codificados en UTF-8 con caracteres fuera del conjunto ASCII de 7 bits. Tiempo estimado: 1 día ideal. Completada.
   - Implementar hojas de estilos especializadas para los marcadores de Firefox y volcados de Django. Tiempo estimado: 2 días ideales. Pospuesta.

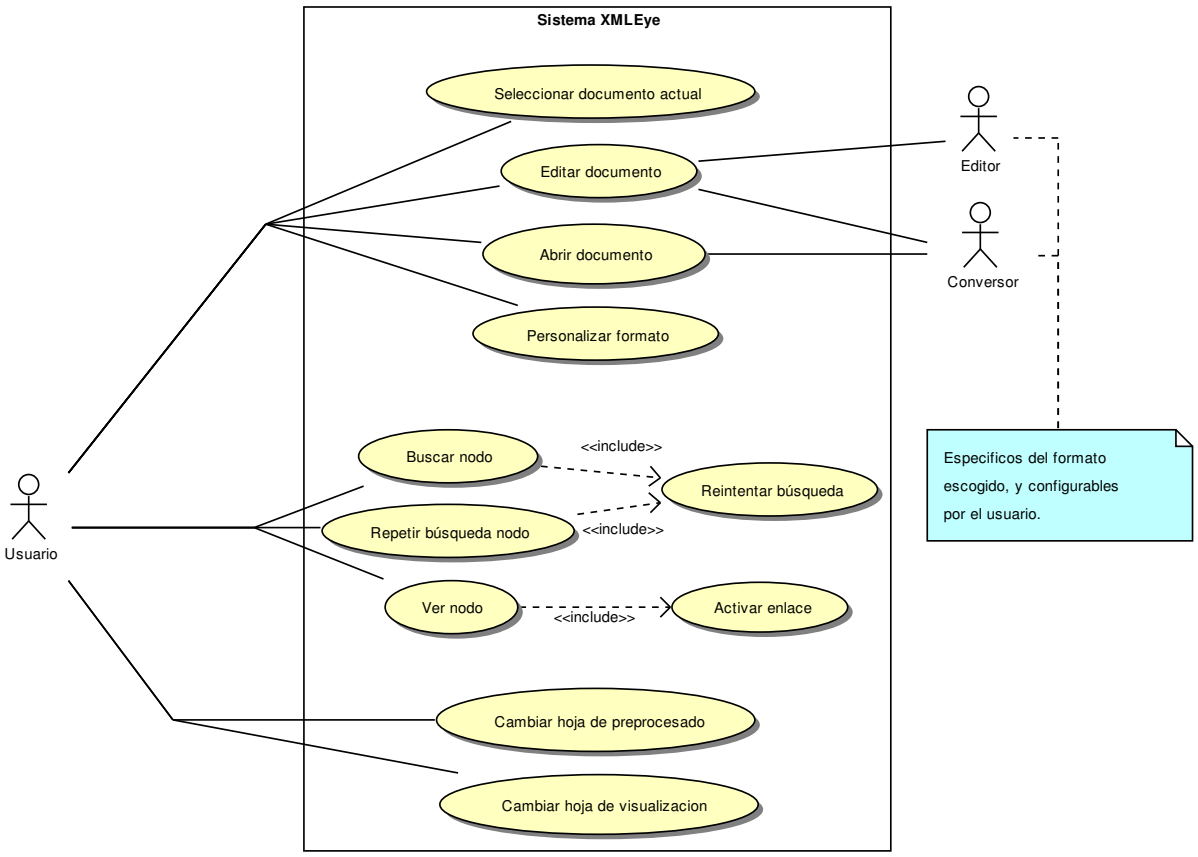


Figura 4.5: Diagrama de casos de uso

4.4.2. Casos de uso

Estos casos de uso nos permiten analizar de manera general y abstracta qué es lo que se requiere en conjunto de las historias de usuario de XMLEye de este Proyecto y la versión del Proyecto Fin de Carrera sobre el que se basa. El diagrama que ilustra las relaciones entre los distintos casos de uso que cumplen las metas del usuario se puede hallar en la figura 4.5.

El límite del sistema establecido engloba únicamente a XMLEye. Los conversores externos, entre los cuales se hallan `ACL2::Procesador` y XMLEye, participan únicamente como un agente externo. El método de comunicación con XMLEye se aclarará más tarde, en la fase de diseño. A menos que se indique lo contrario, los casos de uso aquí mostrados son en general de nivel de meta de usuario.

Seleccionar documento actual

Actor principal Usuario: desea seleccionar el documento sobre el que actuar en el resto de casos de prueba de entre los documentos abiertos.

Precondiciones Existe al menos un documento abierto.

Postcondiciones Se muestra el documento seleccionado.

Escenario principal

1. El usuario indica su intención de cambiar de documento.
2. El sistema proporciona una lista de los documentos disponibles.
3. El usuario selecciona el documento de la lista.
4. El sistema registra la selección, guarda el estado actual de la visualización actual y restaura el estado de la visualización del documento seleccionado, mostrándolo tal y como estaba antes de haber seleccionado otro documento.

Variaciones

3a. El documento es el mismo que el actual:
   1. El sistema cancela el cambio de uso.

Editar documento

Actor principal Usuario: desea editar en paralelo a la visualización el enunciado fuente del documento actual de forma rápida y sencilla usando su editor preferido.

Actores secundarios

Editor Editor preferido del usuario para el formato del documento actual.

Conversor Sistema externo ocupado de convertir el documento a XML, si es necesario.

Precondiciones Existe un documento abierto.

Postcondiciones Se ha abierto el documento con el editor especificado en el descriptor de formato correspondiente.

Escenario principal

1. El usuario indica su intención de editar el enunciado del documento abierto.
2. El sistema lanza el editor, y continúa con su funcionamiento normal, esperando en segundo plano a que se realicen cambios.
3. El usuario guarda algunos cambios en el documento.
4. El sistema detecta los cambios: se regenera la visualización completa del documento.
5. El sistema pasa de nuevo a la espera de más cambios en segundo plano.

Variaciones

2a. La invocación del editor ha fallado:
   1. El sistema indica el error y cancela el caso de uso.

2b. Se ha cerrado el documento:
   1. El sistema deja de monitorizar el documento.

Abrir documento

Actor principal Usuario: desea examinar la versión más reciente del documento en general, para después centrarse en los nodos de su interés.

Actores secundarios Conversor: sistema externo que se ocupa de convertir el documento proporcionado a XML.

Precondiciones Ninguna.

Postcondiciones Se carga la versión más reciente del documento elegido por el usuario.

Escenario principal

1. El usuario indica su intención de abrir un documento ya existente.
2. El sistema muestra los documentos disponibles.
3. El usuario selecciona un documento.
4. El sistema comprueba que el documento existe.
5. El sistema le asocia el primer formato con una extensión coincidente, y aplica su conversor, si tiene uno asociado, al documento. .
6. El sistema muestra el documento elegido, tras añadirlo a la lista de documentos abiertos.

Variaciones

3a. El usuario cancela la selección de documento:
   1. El sistema cancela el caso de uso.

4a. El documento no existe:
   1. El sistema muestra el error y cancela el caso de uso.

5a. No existe un formato con una extensión coincidente:
   1. El sistema utiliza el formato por defecto, XML, que no tiene un conversor asociado.

Personalizar formato

Actor principal Usuario: desea cambiar las opciones de alguno de los formatos instalados.

Precondiciones Existe al menos un formato instalado en el sistema.

Postcondiciones Se cambian los ajustes locales del usuario para el formato indicado, sin cambiar los ajustes globales.

Escenario principal

1. El usuario indica su intención de cambiar las opciones de alguno de los formatos.

2. El sistema muestra la lista de los formatos disponibles.
3. El usuario selecciona un formato de la lista.
4. El sistema muestra los campos modificables del formato:
   - La traducción del nombre del formato que se ajuste mejor a la localización activa por defecto en el entorno del usuario.
   - La orden usada para invocar al editor preferido del usuario para dicho formato.
   - La orden usada para invocar al conversor a XML preferido del usuario para dicho formato.
   - La lista de extensiones, separadas por comas, con las que se asociará a este formato.
5. El usuario modifica el valor de los campos.
6. El sistema solicita confirmación al usuario.
7. El usuario proporciona la confirmación.
8. El sistema registra los cambios a nivel del usuario actual, sin modificar los ajustes globales, y los hace efectivos a partir del mismo momento.

Variaciones

2-7a. El usuario cancela la personalización:
   1. El sistema cancela el caso de uso, sin que se hagan efectivos los campos.

7a. Los campos de nombre del formato, orden de edición o extensiones se hallan vacíos o sólo contienen espacios en blanco, alguna de las extensiones pasadas contiene espacios o está vacía, o la orden de importación contiene solamente uno o más espacios en blanco:
   1. El sistema informa de la situación y solicita corregir dicha situación.

Buscar nodo

Actor principal Usuario: desea localizar rápidamente los nodos que cumplan una serie de condiciones.

Precondiciones Hay un documento abierto.

Postcondiciones Se muestra el resultado de la búsqueda.

Escenario principal

1. El usuario indica su intención de iniciar una nueva búsqueda.
2. El sistema solicita la clave de búsqueda.
3. El usuario introduce la clave de búsqueda.
4. El sistema comprueba que la clave de búsqueda es válida.
5. El sistema presenta al usuario las opciones de filtrado.
6. El usuario elige las opciones de filtrado.

7. El sistema realiza la búsqueda a partir del nodo actual y muestra el primer resultado.

Variaciones

1-6a. El usuario cancela la búsqueda:
   1. El sistema cancela el caso de uso.

4a. La clave no es válida:
   1. El sistema indica el error y pide una nueva clave.

7a. No hay resultados:
   1. *include(Reintentar búsqueda)*

Repetir búsqueda

Actor principal Usuario: desea encontrar otro nodo que cumpla las mismas condiciones que el anterior, sin tener que volver a especificar las opciones de filtrado.

Precondiciones Hay un documento abierto.

Postcondiciones Se muestra el siguiente nodo que cumple las condiciones de la última búsqueda.

Escenario principal

1. El usuario desea repetir la búsqueda anterior:
2. El sistema comprueba que existe una búsqueda anterior.
3. El sistema realiza de nuevo la búsqueda a partir del nodo actual y muestra el primer resultado.

Variaciones

2a. No existen búsquedas anteriores:
   1. El sistema indica el error y cancela el caso de uso.

3a. No hay resultados:
   1. *include(Reintentar búsqueda)*

Reintentar búsqueda

Actor principal Usuario: desea visualizar un nodo que cumpla las condiciones, aunque tenga que relajar el alcance de la búsqueda.

Nivel de caso de uso Subfunción.

Precondiciones Hay un documento abierto.

Postcondiciones Se muestra el primer nodo que cumple las nuevas condiciones relajadas.

Escenario principal

1. El sistema indica la situación y ofrece repetir la búsqueda con parámetros más generales.
2. El usuario acepta.
3. El sistema relanza la búsqueda y muestra el primer resultado.

Variaciones

2a. El usuario rechaza la oferta:
   1. El sistema cancela el caso de uso.

3a. No hay resultados:
   1. El sistema indica la situación y cancela el caso de uso.

Ver nodo

Actor principal Usuario: desea visualizar la información acerca de un nodo del documento.

Precondiciones Hay un documento abierto.

Postcondiciones Se muestra la información del nodo seleccionado.

Escenario principal

1. El usuario indica el nodo del documento que desea ver.
2. El sistema comprueba que existe.
3. El sistema muestra la información del nodo al usuario, dejando que interactúe con él.

Variaciones

2a. El nodo no existe:
   1. El sistema muestra una visualización vacía.

3a. El usuario activa uno de los hipervínculos disponibles:
   1. *include(Activar enlace)*

Activar enlace

Actor principal Usuario: desea visualizar el recurso al que señala el enlace activado.

Nivel de caso de uso Subfunción.

Precondiciones Hay un documento abierto, y se está visualizando un nodo.

Postcondiciones Se muestra el destino del enlace pulsado, si tiene uno definido.

Escenario principal

1. El usuario activa el enlace en cuestión de la visualización del nodo actual.
2. Si el enlace señala a otro nodo, se ejecuta el caso de uso "Ver nodo" sobre él, y se termina el caso de uso.
3. Si el enlace señala a una URL (Uniform Resource Locator), se muestra el documento en uno de los navegadores Web disponibles, y se termina el caso de uso.
4. Si el enlace señala a un ancla de la visualización HTML del nodo actual, se desplaza la posición actual a la del ancla, y se termina el caso de uso.
5. Si el enlace señala a un nodo de otro documento, se visualiza dicho documento mediante "Visualizar documento", posteriormente se ejecuta "Ver nodo" sobre el nodo especificado, y se termina el caso de uso.
6. El sistema cancela el caso de uso.

Cambiar hoja de preprocesado

Actor principal Usuario: desea cambiar la forma en que se muestra la estructura del documento.

Precondiciones Existe al menos un documento abierto y seleccionado.

Postcondiciones Se le muestra al usuario la nueva estructura del documento, procesada desde el texto XML original a través de la hoja seleccionada. Se usará esta hoja para todos los ficheros que se abran a continuación.

Escenario principal

1. El usuario indica su intención de cambiar la hoja de preprocesado en uso.
2. El sistema muestra una lista de las hojas de preprocesado disponibles.
3. El usuario selecciona una hoja de la lista.
4. El sistema recupera el texto XML original del documento y lo procesa mediante la nueva hoja seleccionada.
5. El sistema muestra la nueva estructura del documento, y vacía la visualización del nodo actual, al no haber ninguno seleccionado.
6. El sistema registra la hoja seleccionada como hoja de preprocesado por defecto para todos los documentos abiertos en adelante.

Variaciones

1-3a. El usuario cancela la selección:

1. El sistema cancela la ejecución del caso de uso.

Cambiar hoja de visualización

Actor principal Usuario: desea cambiar la forma en que se muestra la información de los nodos de un documento.

Precondiciones Existe al menos un documento abierto y seleccionado.

Postcondiciones Se visualiza el nodo actual y todos los posteriormente seleccionados en el documento actual y en todos los documentos abiertos posteriormente bajo la nueva hoja.

Escenario principal

1. El usuario indica su intención de cambiar la hoja de visualización para el documento actual.
2. El sistema muestra la lista de las hojas de visualización disponibles.
3. El usuario selecciona una de las hojas de la lista.
4. El sistema actualiza la visualización del nodo actualmente seleccionado, si lo hay.
5. El sistema registra la hoja seleccionada como hoja de visualización por defecto para los documentos abiertos de ahora en adelante.

Variaciones

1-3a. El usuario cancela la selección:
   1. El sistema cancela la ejecución del caso de uso.

4.4.3. Modelo conceptual de datos del dominio de ACL2

Notación

Se ha usado UML 2.0, con las siguientes modificaciones:

- Por claridad, algunas clases se hallan duplicadas en los diagramas. Para evitar confusiones, las duplicaciones ocultan los atributos y se hallan en color azul.
- Igualmente, se han omitido las multiplicidades unitarias en las asociaciones y composiciones.

Descripción general

Para `ACL2::Procesador`, los conceptos a tratar son entes abstractos relacionados con el proceso de demostración de ACL2.

Así, partimos de un *Enunciado*, colección de *Orden*es de ACL2 que produce una *Demostración*. Esta *Demostración* asocia a cada *Orden* un *Resultado* con la salida obtenida de ACL2. Esta contendrá diversos tipos de información según la *Orden* usada.

Puede que nuestro *Enunciado* sea la raíz de un proyecto de ACL2, y se halle definido en base a una serie de definiciones ya demostradas en *Libro*s existentes. Es importante ver que un *Libro* puede requerir de una serie de definiciones previas a su vez: según la terminología de ACL2, sería su *mundo inicial*. Todo *Evento* pertenece a un *Paquete*: si no se indica su nombre previamente con una orden `in-package`, será "ACL2".

Adicionalmente, pueden aparecer avisos, errores y observaciones de ACL2 al inicio del resultado de la ejecución de una *Orden* dada.

A su vez, cada *Resultado*, en caso de necesitar demostrar alguna S-expresión, usará una *Meta* principal, que se podrá dividir recursivamente en submetas. Cada *Meta* usa un determinado *Proceso* de demostración.

Se ha dividido el diagrama de clases conceptuales en tres:

1. Visión general del dominio del problema: figura 4.6 de la página 80.
2. Tipos de Orden: figura 4.7 de la página 81.

   Es interesante ver cómo algunas órdenes utilizan a otras. Por ejemplo, como paso de verificación tras la certificación de un libro con `certify-book`, se intentan cargar sus contenidos en el mundo de ACL2, justo como si se hubiera ejecutado una orden `include-book`.
3. Tipos de Proceso: figura 4.8 de la página 82.

Restricciones textuales

1. *Clave externa*: "nombre" de *Documento*.
2. *Clave externa*: "nombre" de *Paquete*.
3. *Clave externa*: "nombre" de cualquier descendiente de *Evento*, si lo tiene.
4. *Clave externa*: "clavedoc" de cualquier descendiente de *Evento*, si la tiene y está especificada.
5. Todo *Resultado* pertenece o a otro *Resultado* o a una *Demostración*.
6. Todo *Meta* pertenece o a otra *Meta* o a un *Resultado*.
7. Toda *Meta* tiene una etiqueta única dentro de una *Demostración*.
8. *Resultado*.éxito $\in$ {verdadero, falso}.
9. *Documento*.actualizado $\in$ {verdadero, falso}.
10. *Demostración*.actualizada $\in$ {verdadero, falso}.

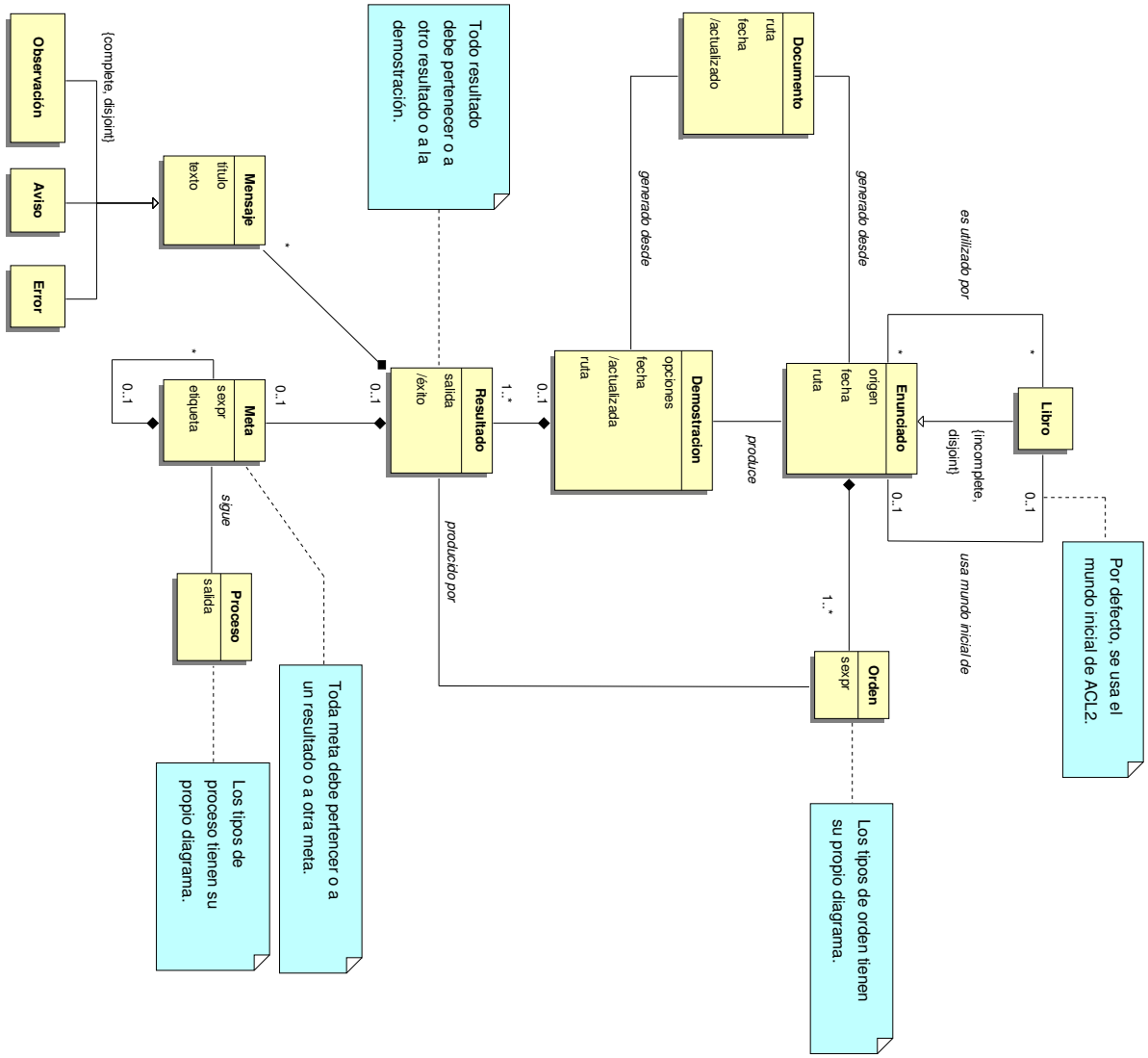


Figura 4.6: Diagrama de clases conceptuales general de ACL2

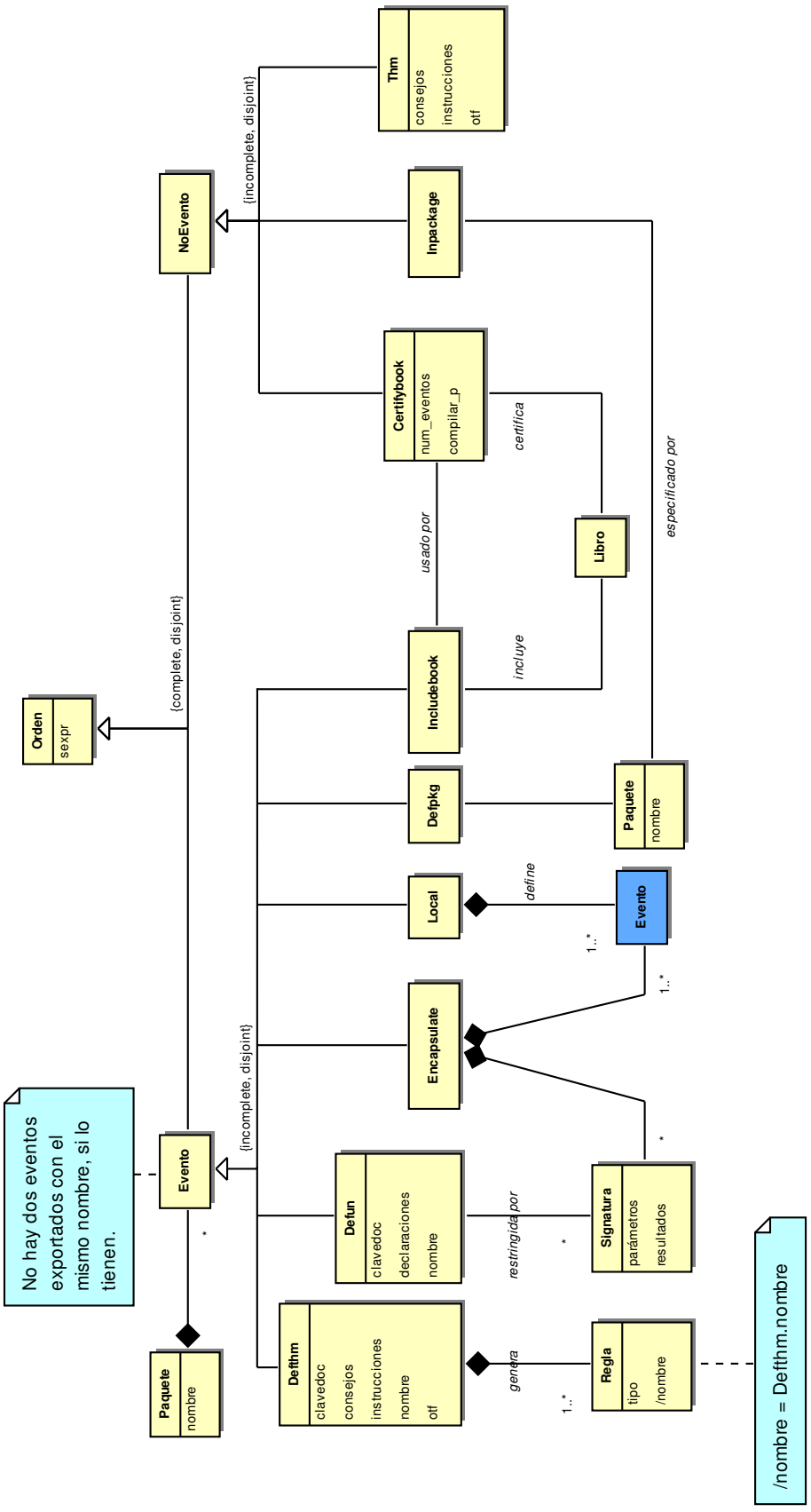


Figura 4.7: Diagrama de clases conceptuales de órdenes de ACL2

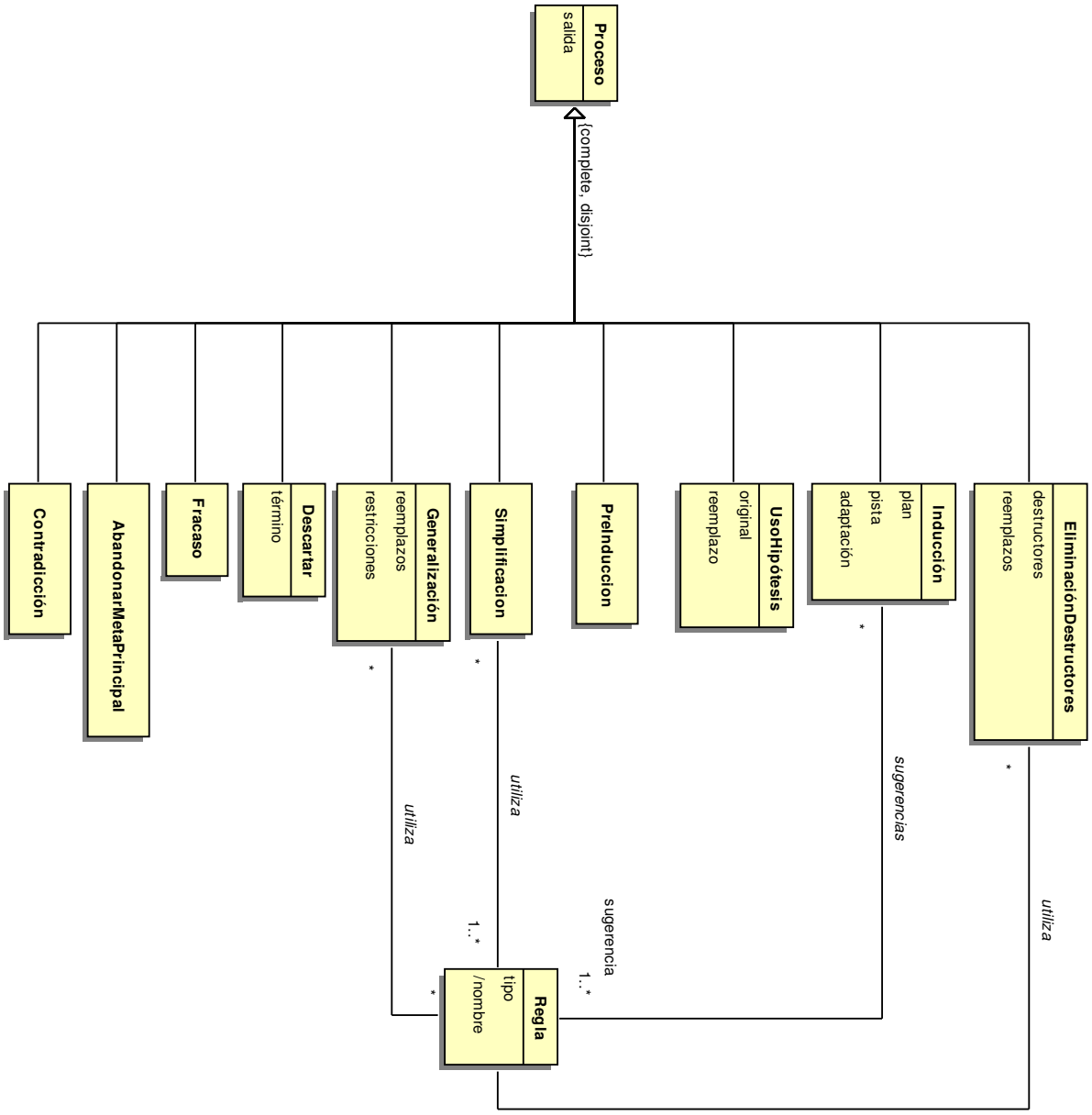


Figura 4.8: Diagrama de clases conceptuales de procesos de ACL2

Derivaciones

1. *Demostración*.actualizada = “verdadero” si *Demostración*.fecha es una fecha y hora posterior a *Enunciado*.fecha. “falso” de lo contrario.
2. *Documento*.actualizado = “verdadero” si *Demostración*.actualizada y *Documento*.fecha es una fecha y hora posterior a *Demostración*.fecha. “falso” de lo contrario.
3. *Resultado*.éxito = no contiene *Meta* con el *Proceso Fracaso* y sus *Resultados* de inferior nivel (si los hay) tienen éxito.
4. *Regla*.nombre = *Defthm*.nombre.

4.4.4. Modelo conceptual de datos del dominio de YAML

Notación

Se seguirá la misma notación que en §4.4.3 (página 78).

Secuencia de procesado

Toda implementación de YAML establece una correspondencia entre las estructuras de datos nativas y los flujos de caracteres Unicode a través de una transformación compuesta por tres pasos en cada dirección, tal y como puede verse en la figura 4.9 de la página 85. Los detalles de cada paso son invisibles a los demás. Según en qué dirección nos desplacemos, hablaremos de:

Volcado Tomaremos una estructura de datos nativa del lenguaje de programación y la presentaremos en forma de una sucesión de bytes dentro de un flujo. En la mayoría de las implementaciones, esta operación recibe el nombre de `Dump`. Seguiremos estos pasos:

1. *Representación* de la estructura de datos nativa en forma de un grafo dirigido con raíz, siguiendo el modelo de información de representación de YAML que describiremos posteriormente.
2. *Serialización* del grafo a un árbol de serialización. Habrá que imponer un orden sobre los nodos del grafo, y evitar los problemas asociados con representar nodos con múltiples arcos entrantes y caminos con ciclos en un árbol. Parte de estos detalles también aparecen en nuestro modelo conceptual de datos.

3. *Presentación* en forma de una sucesión de bytes de la información del árbol, recorriéndolo para lanzar y después capturar una serie de eventos de serialización. Se emplearán diversos estilos y notaciones para generar resultados fácilmente legibles por humanos.

Carga Este es el proceso inverso al volcado, y se suele encontrar bajo el nombre de `Load`. Cada paso va descartando los detalles específicos al paso correspondiente del volcado. Los pasos a seguir son:

1. *Análisis* de los bytes del flujo, lanzando y capturando los eventos de deserialización correspondientes para producir el árbol de serialización que refleje la información original.
2. *Composición* del grafo de nodos a partir del árbol, restableciendo los enlaces originales entre los nodos.
3. *Construcción* de las estructuras de datos nativas del lenguaje a partir del grafo de nodos.

Modelos de información

Normalmente, las aplicaciones sólo necesitan las estructuras de datos finales, que se corresponden en muchos lenguajes (Perl, por ejemplo, o Java, en menor grado) de manera bastante fiel a los grafos del modelo de información de representación. `YAXML::Reverse`, sin embargo, tiene unos requisitos algo especiales: ha de asegurar una correspondencia entre los grafos de YAML y los árboles de XML, por lo que ha de operar a un nivel menor de abstracción, utilizando el modelo de información de serialización descrito en [7]. Una adaptación a UML de dicho modelo se halla en la figura 4.10 de la página 86.

Explicaremos primero los elementos del modelo de información de representación de YAML, y luego describiremos las modificaciones que introduce el modelo de información de serialización.

Modelo de información de representación Un documento YAML se halla representado por el *Nodo* raíz del grafo. Existen varias categorías (*kind* en el original) de *Nodos*: *Escalar*es que contienen una cadena opaca de caracteres Unicode, *Secuencia*s ordenadas de nodos, y vectores asociativos (objetos de la clase *VectorAsociativo*) que contienen un conjunto no ordenado de pares clave-valor (*ParClaveValor*). Es interesante ver que tanto la clave como el valor pueden ser nodos arbitrariamente complejos, pudiendo referirse a cualquier otro nodo del grafo.

Todo nodo dispone de una etiqueta (*tag* en el original) que identifica el tipo abstracto de su contenido. Las etiquetas sirven también en el caso de los escalares para unificar cadenas con el mismo significado a una serie de formas canónicas. Por ejemplo, las reglas de la etiqueta **`tag:yaml.org,2002:int`** utilizan base 10 y un único signo

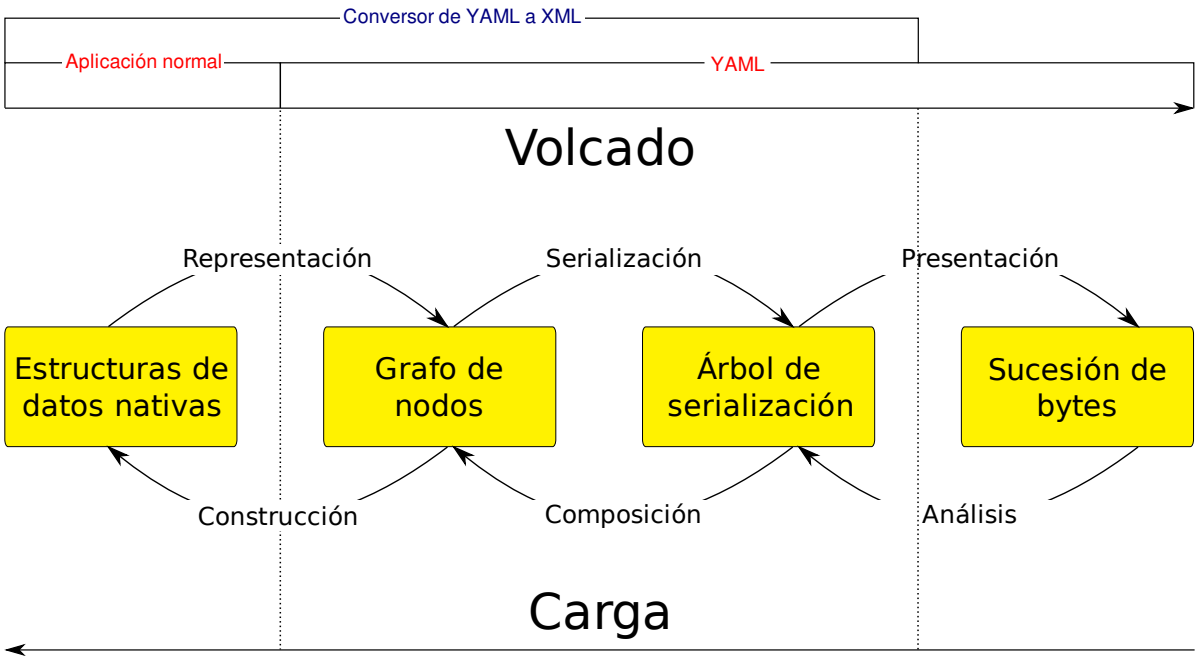


Figura 4.9: Secuencia de procesado de YAML

negativo opcional en sus formas canónicas: un valor hexadecimal como `0xA` sería convertido por cualquier analizador al valor decimal 10.

Modelo de información de serialización Modifica el modelo de representación en dos aspectos:

1. Añade un orden a los pares clave-valor de los vectores asociativos. Este orden es un detalle de serialización, y no tiene impacto alguno en los grafos y estructuras de datos nativas generados.
2. Permite asociar a cualquier nodo un identificador o *ancla*, y usar nodos *alias* señalando a esa ancla en posteriores referencias a ese nodo. Esto permite crear presentaciones compactas de grafos complejos, que podrían incluso contener ciclos en su interior.

   Un ancla no tiene por qué ser única: el nodo al que hace referencia un *alias* no es más que el último que definió esa ancla en el orden establecido.

Restricciones textuales

- *Clave externa*: “nombre” de Etiqueta.

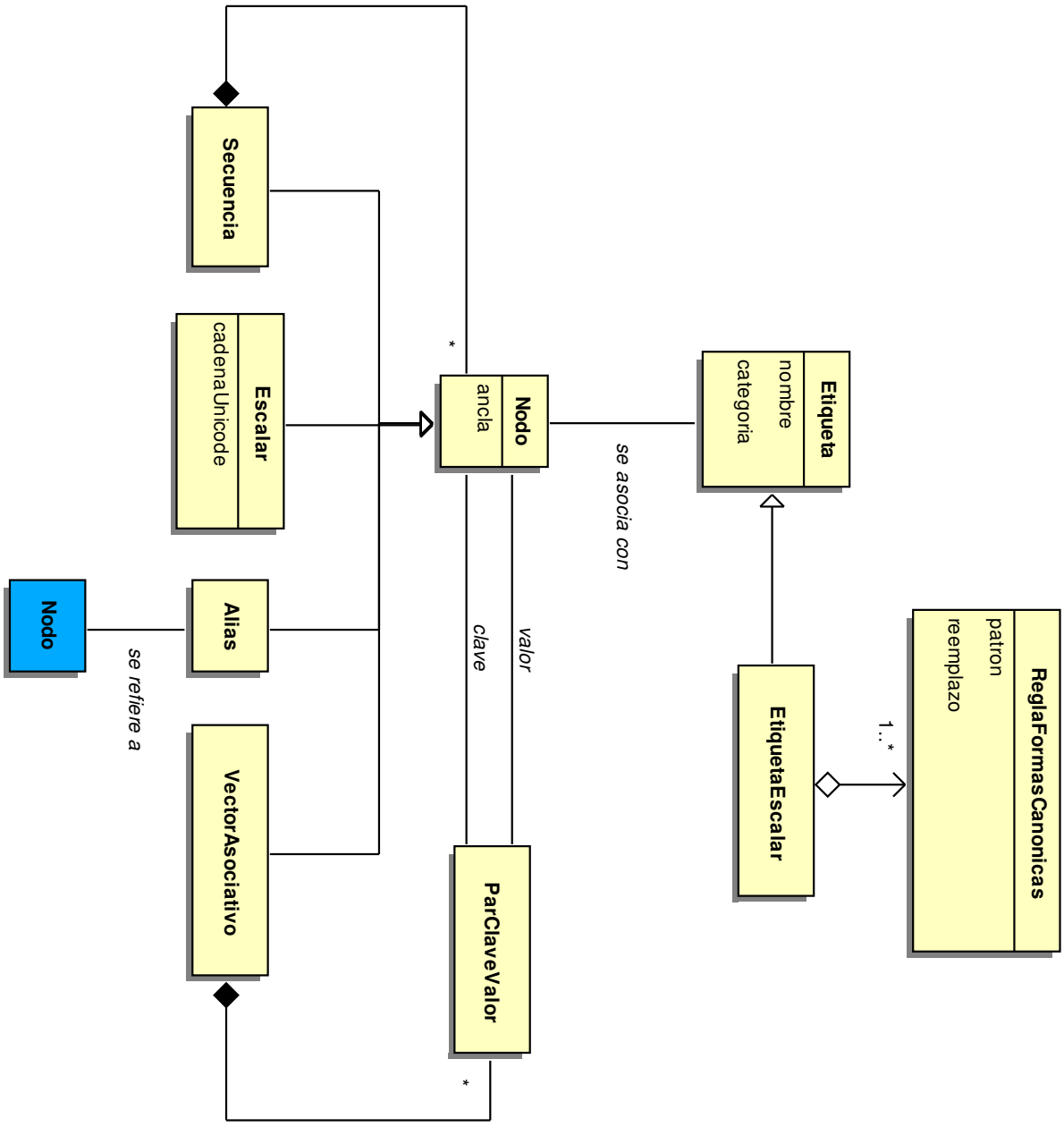


Figura 4.10: Diagrama de clases conceptuales para YAML

4.5. Diseño del sistema

4.5.1. Arquitectura del sistema

Para poder decidir la arquitectura del sistema, primero habría que definir cuáles son las condiciones que afectarán a su estructura.

Dichas condiciones se basan tanto en la funcionalidad esperada del sistema, como en los requisitos no funcionales antes mencionados y otros factores de calidad del software, como la flexibilidad, mantenibilidad, reusabilidad o fiabilidad, entre otros.

La mantenibilidad requiere una arquitectura que evite la propagación de cambios realizados sobre una parte del sistema, definiendo interfaces estables entre estas partes. Dicha arquitectura debe ser además lo más sencilla posible, para facilitar cambios en su estructura y elementos.

La división en partes con límites e interfaces bien definidas mejora también la reusabilidad y facilita las pruebas, evitando acoplamientos innecesariamente complejos que dificulten el desarrollo de una aplicación fiable y con la funcionalidad deseada.

Separación de responsabilidades: división en capas

En el caso de este proyecto, se deseaba separar XMLEye de los detalles de los conversores externos, como `ACL2::Procesador` y `YAXML::Reverse`, y viceversa. En un futuro, se implementarán nuevas interfaces de usuario para XMLEye, por lo que también habrá que separar la lógica de aplicación y de presentación de XMLEye.

Bajo este contexto, se puede ver que la aplicación del patrón arquitectónico Capas (descrito con mayor detalle en [10]) es natural. Se ha dividido la aplicación en cuatro capas:

Presentación Se ocupa de la interacción persona-máquina en general y en particular de la visualización de la información y la navegación a través de ésta. Define las transformaciones a realizar sobre las representaciones creadas en la capa de filtrado.

Aplicación Responde a las peticiones de la capa superior, ejecutándolas de forma independiente a la interfaz escogida. Implementa la infraestructura necesaria para la ejecución de las transformaciones de la capa de usuario, posibilitando su selección en tiempo de ejecución, enlazado de nodos y localización de cadenas. Incorpora la lógica necesaria para la integración con los conversores de la capa de filtrado.

Filtrado A diferencia de la usual capa de dominio en un modelo de tres capas, no manipula instancias de los objetos del dominio, sino que los transforma a una representación manejable por las capas superiores, de ahí su nombre.

Se halla constituida por una serie de conversores independientes entre sí y de XMLEye, como `ACL2::Procesador` o `YAXML::Reverse`. XMLEye sólo ha de conocer la orden a ejecutar, que ha de seguir unas pautas muy sencillas:

- El código de retorno de la orden debe indicar el éxito o fracaso en la conversión.
- El resultado XML ha de volcarse por la salida estándar, y el estado de la conversión por la salida estándar de errores.

Servicios técnicos En esta capa hemos situado todas las bibliotecas, módulos externos y demás que proporcionan parte de la funcionalidad común del sistema, y son internos a éste.

Se ha detallado la visión en capas (modeladas por paquetes UML) a lo largo del eje vertical y en particiones a lo largo del eje horizontal en la figura 4.11 de la página 89.

Más tarde examinaremos cada una de estas capas por separado.

Transformaciones configurables: tuberías y filtros

Descripción Otro patrón arquitectónico usado (con ciertas licencias) en el diseño de este proyecto fue Tuberías y Filtros. Dicho patrón será familiar a todo usuario de un sistema UNIX, de donde se inspiró precisamente.

Consiste en el encadenamiento de una serie de Filtros, partiendo de una Fuente de información, y llegando hasta un Sumidero, destino de la información transformada. A cada paso, la información llega a un Filtro a través de una Tubería, saliendo por otra Tubería hacia el siguiente Filtro o el Sumidero final.

El patrón original (ver [10]) tiene las ventajas de ser fácilmente paralelizable en filtros incrementales que no necesiten leer toda la entrada antes de poder operar, y permitir una gran flexibilidad cambiando el filtro exacto usado a cada paso, dado que están diseñados para ser intercambiables entre sí, con ciertas limitaciones.

Sus desventajas consisten en la dificultad de mantener un estado compartido y de realizar un control detallado de errores, al ser usualmente tratada toda la transformación como una unidad.

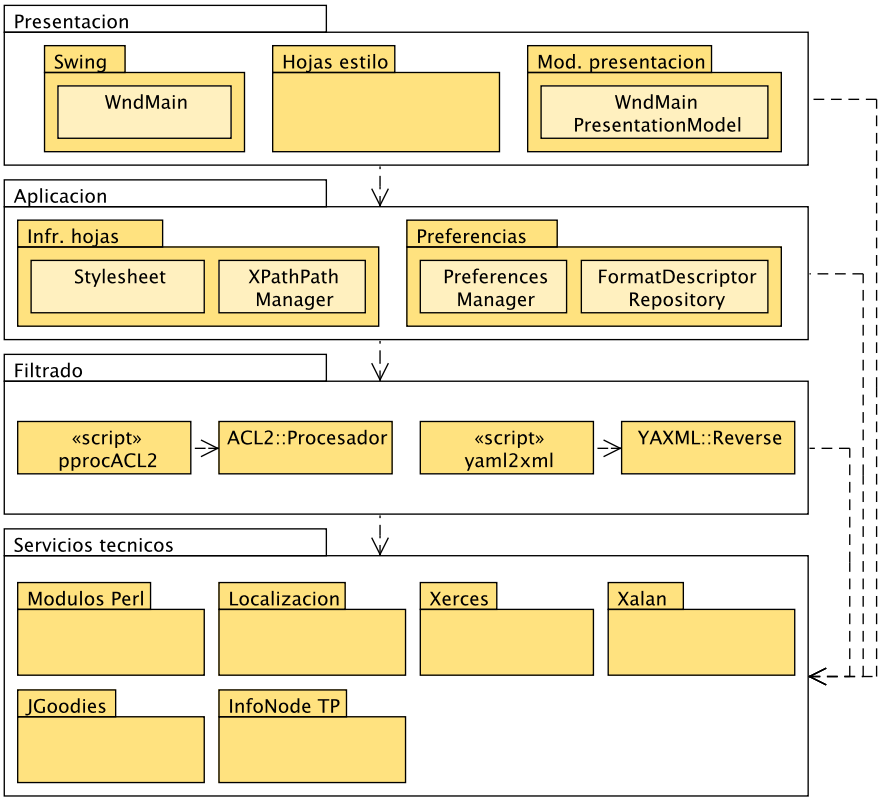


Figura 4.11: Diagrama arquitectónico del sistema

Uso El uso de este patrón a lo largo de este proyecto se fundamentó en la necesidad de permitir la extensibilidad de la visualización por parte del usuario sin necesidad de una recompilación, usando un enfoque puramente dirigido por datos.

Así, en el caso de una demostración de ACL2, si consideramos al sistema externo ACL2 como la fuente de la información y a la visualización por parte de la biblioteca Swing como el sumidero, tendríamos cuatro filtros intermedios:

1. `ACL2::Procesador`, con el enunciado Lisp para ACL2 como entrada, genera una representación en XML del contenido de la demostración realizada. El formato de salida se conoce a través de una DTD, incrustada en el propio documento de salida.
2. El segundo filtro, definido en la capa de presentación, realiza una transformación previa, potencialmente costosa, que afecta a todo el documento, y que sólo se realiza una vez.

   Aquí el usuario puede definir el aspecto que tendrá el árbol del documento, ocultando nodos o cambiando la etiqueta o icono a mostrar. La descripción de la hoja define las entradas esperadas y el formato de la salida.
3. El tercer filtro, también en la capa de presentación, transforma un nodo a su visualización XHTML utilizando una hoja XSLT para el formato de la salida. La transformación se repite cada vez que el usuario cambia de nodo.

   En este caso, se sabe que la salida es XHTML, y que sigue por lo tanto una DTD conocida. También podemos deducir a partir del nombre de la hoja qué entrada espera y qué salida produce.

   Así, sabemos que la hoja `ppACL2` de visualización tomará la salida de la hoja de preprocesado `ppACL2` o una derivada suya y mostrará la información relevante a demostraciones de ACL2 contenida en dicho documento.

   Una hoja de visualización es diseñada suponiendo que la salida del preprocesado sigue el formato de salida de ciertas hojas de usuario de preprocesado. En el caso contrario, la transformación no fallará, pero no se obtendrán los resultados esperados.
4. El cuarto filtro, controlado por la capa de aplicación, marca las coincidencias de la clave de la búsqueda en curso en el texto, si se da el caso. También se aplica una hoja de estilos CSS, que no modifica la estructura pero sí el aspecto del resultado final.

Como vemos, los dos filtros intermedios se ocupan de la mayor parte de la visualización, y según los requisitos funcionales, éstos deberían ser modificables por el usuario sin necesidad de recompilación.

Implementación Se eligió XSLT para implementar estos dos filtros por la flexibilidad que ofrece al estar basado en ficheros de texto fácilmente intercambiables y por su capacidad de describir declarativamente las transformaciones sobre la fuente XML a otros formatos.

Se optó por una visualización basada en XHTML para evitar el uso de numerosos componentes discretos Swing, complicando el diseño de la interfaz y las extensiones por parte del usuario. El uso de hipertexto permite, además de ofrecer mayor información visual al usuario, establecer enlaces entre los distintos elementos de la demostración.

Existen varias diferencias con el patrón original. Los filtros aquí usados no son incrementales (el uso de XML no lo permite), perdiéndose la capacidad de paralelización.

Por otro lado, la pérdida de capacidad de detección detallada de errores no aparece en nuestro caso, al basarse las tuberías en lógica de la capa de aplicación y no en otros mecanismos de menor nivel de abstracción, como memoria compartida.

No ha sido necesario ningún estado compartido, por lo que esta otra desventaja tampoco ha resultado ser ninguna limitación.

En cuanto a rendimiento, hay que considerar el uso de *translets* por el procesador XSLT Xalan XSLTC. Éstos son representaciones compiladas a bytecode Java de hojas XSLT cuya ejecución es notablemente más rápida que la ejecución interpretada clásica, a costa de cierta funcionalidad innecesaria para este proyecto.

La disposición e interconexión de los componentes que intervienen en la transformación se ha recogido en el diagrama UML de componentes de la figura 4.12 de la página 92.

La capa de aplicación no aparece en él, pero se halla implícita, como capa de control. Tampoco han de confundirse las hojas XSLT usadas, que pertenecen a la capa de presentación, con su lógica de control, parte de la capa de aplicación.

Integración de las capas de aplicación y presentación: patrones MVP

En el visor del Proyecto que precede a éste, se había establecido una comunicación entre las capas de aplicación y de presentación mediante Fachadas. Estas Fachadas constituían el único punto de acceso a la funcionalidad de cada capa, y evitaban que la capa superior tuviera que conocer los detalles de la capa inferior.

Aunque para sistemas con interfaces más sencillos se podría haber continuado su uso, en el caso de XMLEye se vio cómo las Fachadas iban tomando demasiadas responsabilidades al mismo tiempo. También se dio el caso de que la capa inferior forzaba a la capa superior a una serie de restricciones, siendo la más grave la de tratar un único documento.

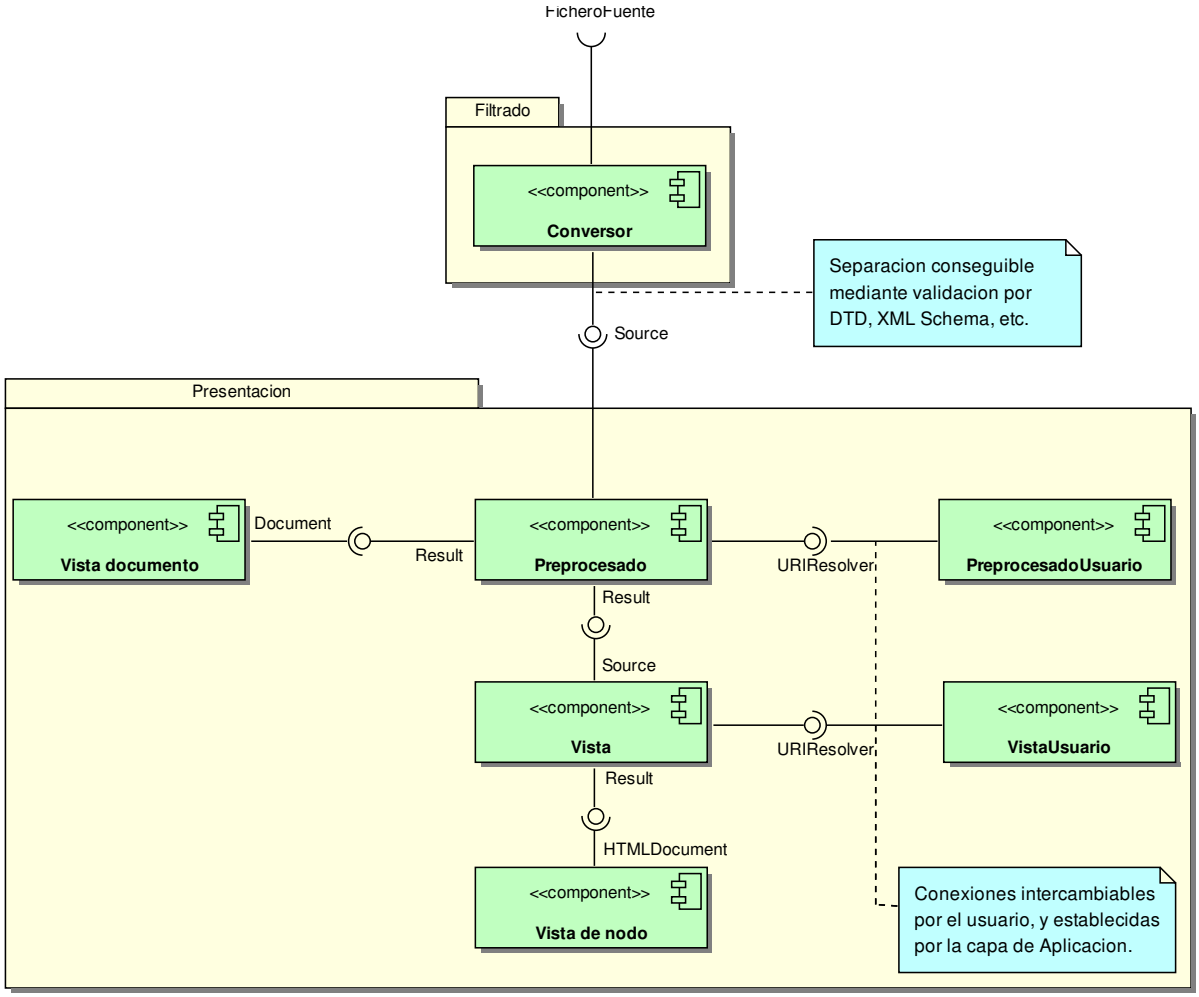


Figura 4.12: Diagrama de componentes para visualización

Otro problema que aquejaba al antiguo diseño es la prácticamente nula capacidad para poder realizar pruebas de unidad de forma cómoda en la aplicación. Existen herramientas que permiten automatizar pulsaciones y entrada de teclado, pero las pruebas de este tipo son extremadamente frágiles: cualquier cambio en la disposición de los controles de la interfaz las invalidaría.

Para resolver ambos problemas, se estudiaron las diversas alternativas ofrecidas en [25], y en particular las derivaciones del patrón MVP (Model View Presenter). Este patrón es una derivación del MVC (Model View Controller) [30] original en el que en vez de tener un trío modelo-vista-controlador para cada control de la interfaz (como un campo de texto, por ejemplo), tenemos un trío para el formulario completo. Ahora el modelo del formulario es el conjunto de clases del dominio usadas, la vista es la estructura formada por todos los controles gráficos del formulario, y el presentador es el componente al que se notifican todas las acciones de interés realizadas por el usuario.

El presentador es el ocupado de modificar el modelo, y posteriormente la vista se actualizará con los cambios hechos por el presentador al modelo. Hay diversos mecanismos para garantizar esta sincronización, constituyendo variantes distintas del patrón MVP básico:

Controlador Supervisor Para los casos simples, la sincronización modelo-vista se establece a partir del patrón Observador o algún otro enfoque declarativo. En casos más complejos, el presentador interviene directamente sobre los controles de la interfaz.

Vista Pasiva La vista en este caso carece de cualquier lógica de sincronización, y es el presentador el que rellena todos los campos. Una posibilidad para evitar acoplar demasiado el presentador a los controles usados es definir una interfaz en el formulario para su relleno: de esta forma, durante las pruebas de unidad podemos sustituir la vista por un Doble para Pruebas [43] y depurar prácticamente toda la funcionalidad.

Modelo de Presentación El presentador contiene el estado abstracto que debería presentar el formulario. Puede que sea el modelo de presentación el que manipule la vista, o que sea la vista la que se actualice a partir del modelo de presentación. En el primer caso, ambos elementos están más acoplados pero podemos probar la sincronización. En el segundo caso, la sincronización no se puede depurar tan fácilmente, pero el modelo de presentación es completamente independiente de la interfaz usada.

De entre estos enfoques, se escogió el Modelo de Presentación, ya que en un futuro se desean crear nuevas interfaces para XMLEye: el presentador no debería de saber nada acerca de la vista. Por la misma razón, es la vista la que se actualiza al estado encapsulado por el modelo de presentación.

El uso de un modelo de presentación también facilita la implementación de una interfaz multidocumento: cada documento puede imponer su propio estado de presentación sin afectar a los demás.

En cuanto a la división en modelo-vista-presentador, tendríamos a los modelos y otros servicios de alto nivel en la capa de aplicación, y a las vistas y presentadores en la capa de presentación.

4.5.2. Capa de filtrado: ACL2::Procesador

Volviendo al diagrama de capas de la figura 4.11 de la página 89, vemos que este conversor ha de:

- Convertir la salida de ACL2 a XML sin pérdidas.
- Definir claramente el formato de dicha salida.

Por la necesidad de manipular el texto de la forma más sencilla y potente posible, se escogió usar un guión escrito en Perl, un lenguaje diseñado a este efecto.

A raíz de la complejidad inherente en el manejo de la salida en lenguaje humano de un programa como ACL2, se ha seguido el paradigma orientado a objetos a la hora de desarrollar `ACL2::Procesador`.

Procesado general de proyectos de ACL2

El diseño básico se fundamenta en el modelo conceptual de datos, con ciertas modificaciones. El guión lanzado por la capa de aplicación emplea la función *convertir_a_xml* del módulo *Procesador*, pasándole la ruta al *Enunciado* raíz del proyecto completo, y vuelca sus resultados por la salida estándar.

Este método es realmente un envoltorio que crea un *Proyecto* con raíz en el *Enunciado* indicado, actualiza por completo de forma recursiva todas las salidas de ACL2 y sus versiones en XML y devuelve la versión XML de la salida de la raíz del proyecto.

Las *Dependencias* del *Proyecto* son obtenidas a partir de un análisis estático del código Lisp, empezando por el *Enunciado* Lisp raíz y continuando de forma recursiva. Esto genera un árbol anidado de *Proyecto*s que dependen unos de otros. Decimos árbol y no grafo pues un subproyecto puede aparecer varias veces en el mismo árbol. Esto supone un coste mayor en espacio, pero no en tiempo: si tuviera que ser actualizado, sólo sería procesado una vez. Se están considerando representaciones más eficientes en espacio para versiones posteriores.

El proceso de actualización consiste en recorrer el árbol e ir actualizando de forma recursiva:

1. Se solicita la actualización de las *Dependencias* del proyecto actual, que forman *Proyecto*s a su vez.
2. Si algún subproyecto fue actualizado o el fichero temporal con la salida de ACL2 no existe o tiene una fecha de modificación más antigua que la del *Enunciado*, se invocará a ACL2 para crear o actualizar dicho fichero temporal.
3. Si el fichero temporal con la salida fue actualizado o el fichero temporal con el documento XML resultante no existe o tiene una fecha de modificación más antigua que la de la salida, se actualizará empleando una instancia de la clase *Demostración*, que encapsula el proceso de análisis de una demostración de ACL2.

El diagrama de secuencia correspondiente a esta lógica se halla en el cuadro 4.14 de la página 97. El diagrama con las clases participantes, integradas con las ocupadas de procesar las *Demostracione*s en sí, forma el cuadro 4.13 de la página 96.

Generación de modelos en memoria

Un concepto importante para comprender el diseño de `ACL2::Procesador` en esta versión es la necesidad cada vez mayor de atrasar el momento de la generación de la salida XML. Originalmente, ambos estaban intercalados. Sin embargo, en muchos casos el orden en que se procesa la información no coincide con el orden en que se debe producir la salida, y esto suponía un impedimento para el procesamiento de demostraciones cada vez más complejas.

A lo largo de las iteraciones, se ha hecho una división mayor y mayor, hasta finalmente separar el procesado y la salida a nivel de una *Demostración* completa, en vez de *Resultado*s individuales. Esto permite dar nuevos usos al análisis ya implementado. Puede verse esta separación en el diagrama de secuencia de la figura 4.15 de la página 98: la inicialización se hace sólo durante la ejecución del constructor `new`, y la salida se produce exclusivamente dentro del método `escribir_xml`.

Por ejemplo, se puede analizar una *Demostración* sin contar con su salida, realizando un análisis estático sobre el código Lisp: toda *Orden* dispone de métodos separados para recuperar información de la S-expresión y de la salida de ACL2, y el último sólo es usado cuando se proporciona dicha salida en el constructor. Este análisis estático proporciona información de todas las dependencias de un proyecto.

Relación con capa de servicios técnicos

Viendo el diagrama general de clases de diseño para la capa de filtrado, se puede observar que la mayoría de las clases tienen dependencias con *XML::Writer* y *Localización*.

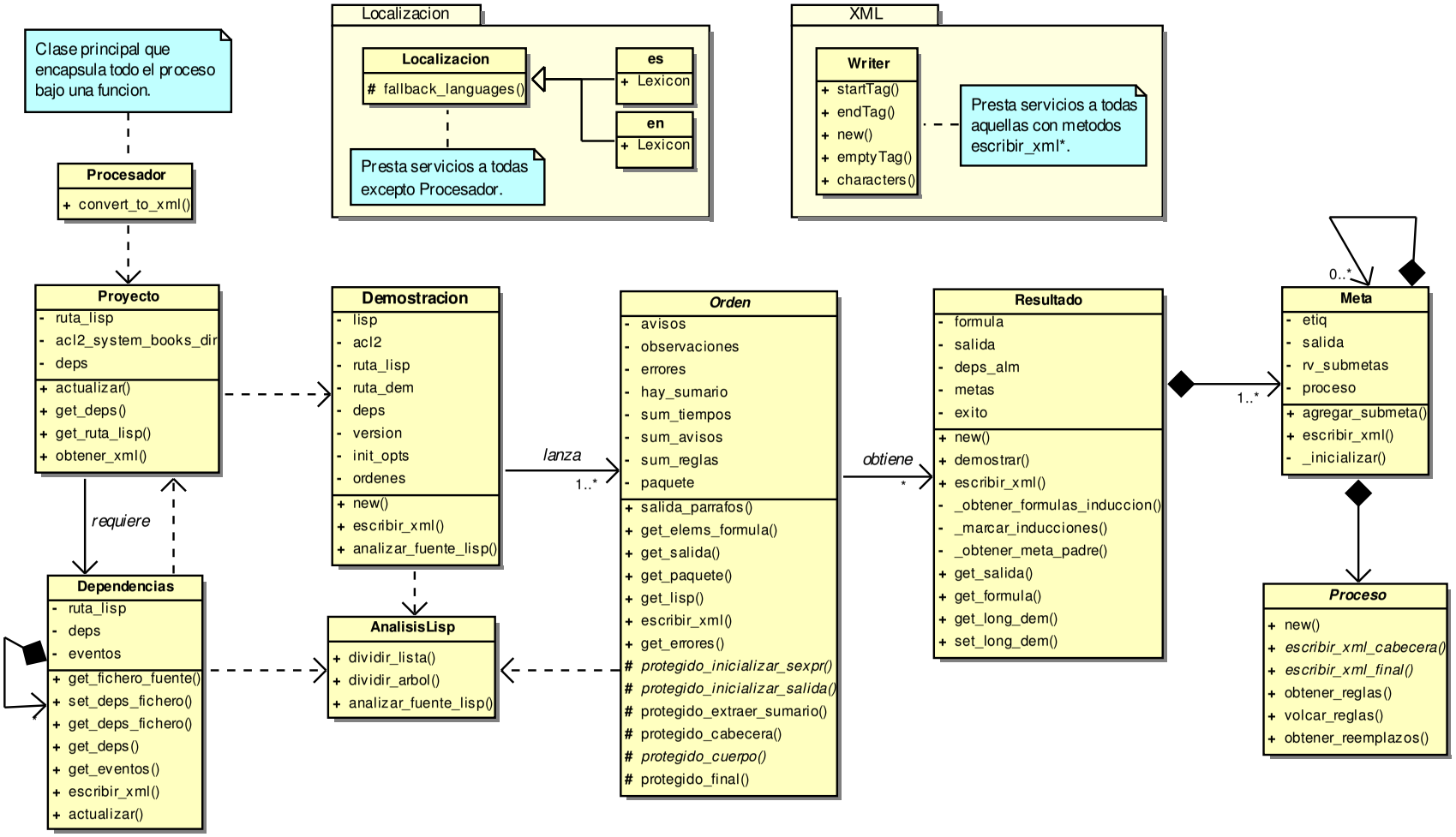


Figura 4.13: Diagrama de clases de `ACL2::Procesador` (general)

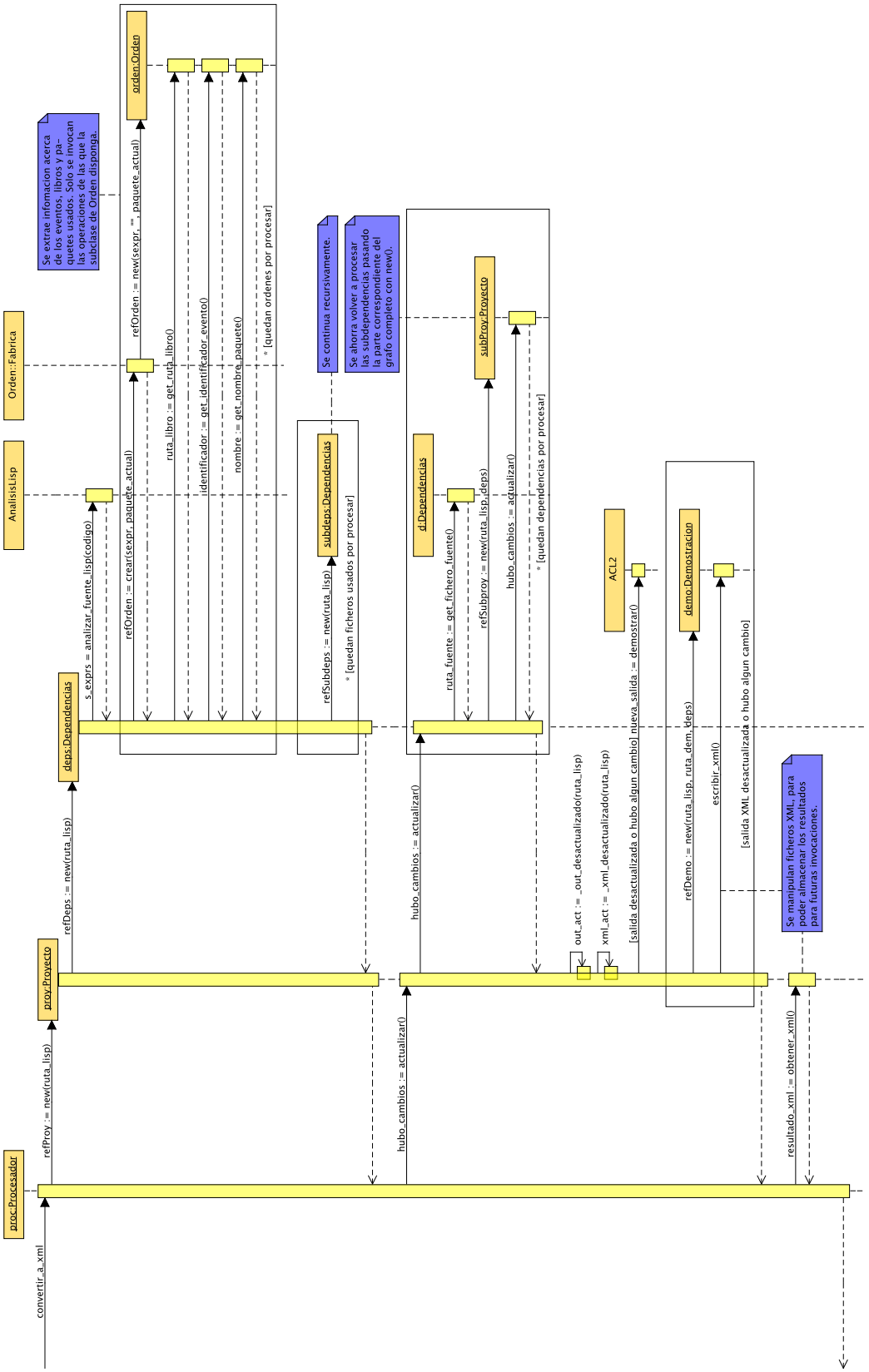


Figura 4.14: Diagrama de secuencia general de filtrado

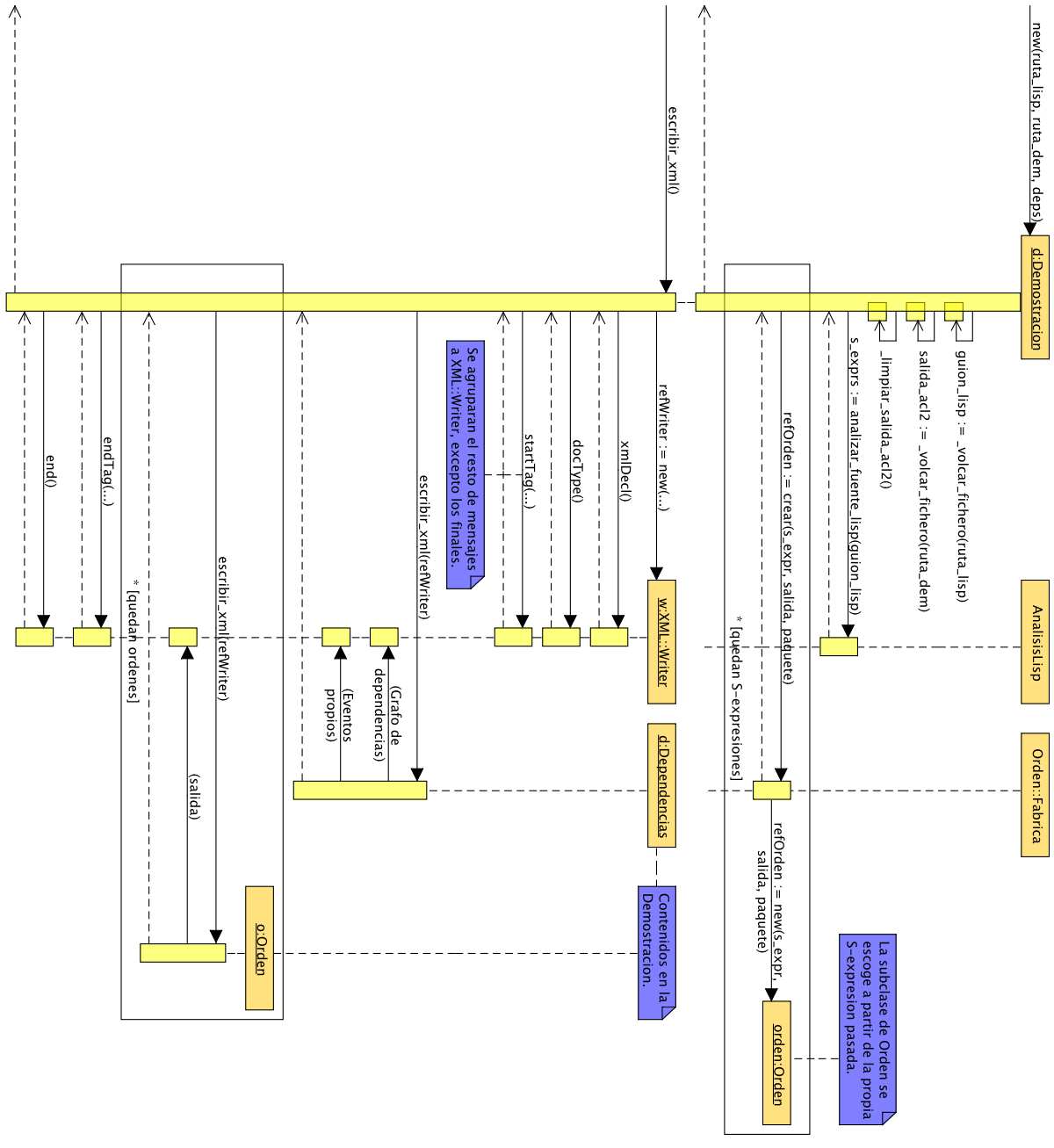


Figura 4.15: Diagrama de secuencia de filtrado de una *Demostración*

Esto es lo usual con los elementos de la capa de servicios técnicos: *XML::Writer* es un módulo usado para la salida XML, que incorpora ciertas comprobaciones automáticas para garantizar la ejecución de generación de documentos bien formados, y *Localización* es el módulo usado para localización de mensajes de error o avisos.

No hay que confundir dichos servicios técnicos con clases de utilidad como *AnálisisLisp*, que aísla la lógica necesaria para convertir listas y árboles Lisp a estructuras de datos equivalentes de Perl.

Reflexión y tipos basados en comportamiento: simplificación del análisis estático

Al implementar el análisis estático necesario para conseguir el grafo de dependencias de un proyecto, surge la necesidad de conocer cuál sería el identificador final y paquete de cada evento, y de las rutas de todos los libros. Evidentemente, hay que distinguir entre los *Eventos* con nombre, los *Eventos* sin nombre y los *NoEventos*, que no producen cambios en el mundo lógico de ACL2.

Sin embargo, se desea conservar toda la lógica propia a cada orden en su propia subclase de *Orden*. La forma más lógica sería añadir niveles a la estructura de herencia, distinguiendo así entre los *Eventos* con nombre, los *Eventos* sin nombre y los *NoEventos*, pero esto sólo resolvería parte del problema: seguiríamos teniendo que preguntar si la *Orden* usada pertenece a la lista de aquellas que afectan al paquete actual, o de la lista de *Ordenes* ocupadas del manejo de libros. Tendríamos entonces que utilizar herencia múltiple. Como vemos, preguntar por la subclase exacta de la *Orden* según el criterio que necesitemos en un momento determinado añade una complejidad al código y crea un acoplamiento mucho mayor de lo esperado.

La alternativa se halla en el concepto conocido como “duck typing” [42], muy popular en lenguajes con sistemas de tipos dinámicos como Python, y defendido (aunque no bajo ese mismo nombre) por autores como Bruce Eckel [21]. La definición del glosario de Python [47] es la siguiente:

> Pythonic programming style that determines an object’s type by inspection of its method or attribute signature rather than by explicit relationship to some type object ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using type() or isinstance(). Instead, it typically employs hasattr() tests or EAFP programming.

Es decir, no distinguimos a los tipos por “quiénes” son, con `instanceof` en Java o `typeid` en C++, por ejemplo, y luego hacemos *upcasting* a una clase o interfaz común. En su lugar, los distinguimos por cómo se comportan: si `anda()` como un *Pato* y `hace_cuac()` como un *Pato*, entonces es un *Pato* para nuestro código.

Se podría ver el "duck typing" como una versión de la programación genérica de C++ en que la comprobación de tipos se hace en tiempo de ejecución y sólo se requieren aquellos métodos que son realmente usados, en vez de todos las que aparecen en el código.

Existen dos estilos para su implementación, como sugiere la definición anterior. El segundo estilo hace referencia al acrónimo EAFP (Easier to Ask for Forgiveness than Permission): asumimos que los métodos necesarios se implementan y capturamos los errores si nos equivocamos. Dado que en el análisis estático se mezclan las *Orden*es que cumplen o no cada uno de los criterios libremente, este método no nos sirve: el manejo de excepciones no es un mecanismo de control de flujo, al fin y al cabo.

El primer método sí es útil, pero notemos que hace uso de algo que no todo lenguaje tiene: la capacidad de consultar y operar sobre la propia estructura del programa, en tiempo de ejecución. Dicho proceso es conocido como *reflexión*, y se halla integrado dentro de muchos lenguajes, como Java, C#, Perl o Python.

Así, el análisis estático asume que si una clase implementa el método `get_nombre_paquete` cambia el paquete actual, si implementa `get_identificador_evento` representa a un *Evento* con un nombre referenciable posteriormente, y si implementa el método `get_ruta_libro`, que de alguna forma ha incluido los contenidos del libro bajo la ruta especificada bajo el mundo lógico de ACL2.

Patrón Método Fábrica: creación de Órdenes y Procesos

Se usó el patrón *Método Fábrica* de [30] para aislar al resto del post-procesador de la lógica necesaria para identificar qué tipo de *Orden* o *Proceso* hay que crear exactamente.

Podemos ver en el diagrama de clases de diseño general de la figura 4.13 en la página 96 dos clases abstractas, *Orden* y *Proceso*. Las instancias de sus subclases se crean mediante los métodos `crear` de *ACL2::Orden::Fabrica* y *ACL2::Proceso::Fabrica*.

En ambos casos, son los valores de los argumentos proporcionados los que determinan qué subclase concreta de la clase abstracta base se va a crear. En el caso de *Orden*, se envía la S-expresión, el paquete Lisp actual y la salida. En el caso de *Proceso*, se envía la salida.

ACL2::Proceso::Fabrica requiere algo más de participación por parte de las subclases de Proceso. Todas deben registrar una subrutina anónima que determine si la salida enviada por *Meta* se corresponde con ella, y que devuelva los parámetros requeridos al constructor en dicho caso. Esto podría considerarse como un uso del patrón *Orden*, descrito en más detalle en la capa de presentación.

Dicha fábrica queda así como un punto central fácilmente accesible por *Meta* de creación de *Procesos*, y la lógica específica a cada proceso se halla concentrada y separada del resto en un único fichero.

Patrón Método Plantilla

Este patrón, también proveniente de [30], se usa en la jerarquía de *Orden* para reunir todo el comportamiento común en el filtrado de las órdenes.

Toda orden en ACL2 produce una serie de mensajes de error, aviso u observación. También se añade usualmente (pero no siempre) un sumario, cuyo formato es fijo en toda orden. Se da además que la lógica de manejo de errores de ACL2 es siempre la misma.

Toda esta lógica común es reunida en un método de una clase base, que emplea una serie de "operaciones de enganche" protegidas, que permiten especializar mediante herencia partes de ese método con la lógica específica para cada orden, facilitando añadir nuevas órdenes o modificar las existentes. El nombre de "método plantilla" viene de esa capacidad de rellenar los "huecos" de su lógica.

Orden dispone de dos métodos plantilla:

1. `inicializar` dispone de los enganches:
   - `protegido_inicializar_sexpr`: operación abstracta.
   - `protegido_inicializar_salida`: operación abstracta.
   - `protegido_extraer_sumario`: tiene una implementación por defecto.
2. `escribir_xml` se puede especializar a partir de:
   - `protegido_cabecera`: tiene una implementación por defecto.
   - `protegido_cuerpo`: operación abstracta.
   - `protegido_final`: tiene una implementación por defecto.

El uso del prefijo "`protegido_`" en los nombres se trata de un esquema de nombrado, siguiendo la práctica común en muchos módulos del CPAN, y no implica la existencia de un mecanismo de control de acceso real, por sencillez.

Jerarquía de Órdenes

El diagrama de clases de diseño UML puede verse en la figura 4.16 de la página 102.

Puede verse como cada clase redefine los métodos de enganche necesarios, añadiendo algunas operaciones privadas para su propia funcionalidad.

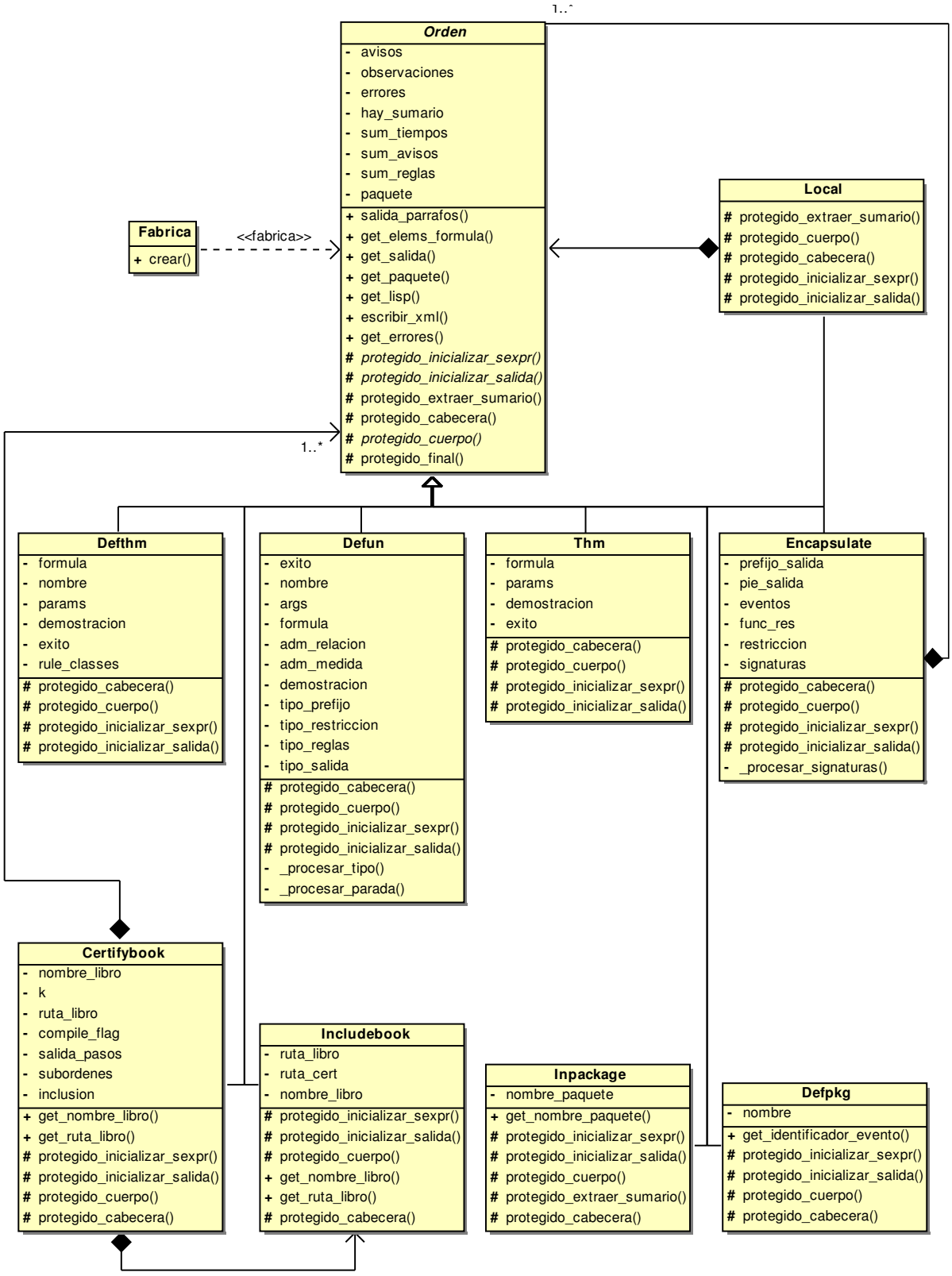


Figura 4.16: Diagrama de clases de Órdenes de `ACL2::Procesador`

Jerarquía de Procesos

El diagrama de esta jerarquía se halla en la figura 4.17 de la página 104.

Los métodos privados estáticos `_identificar` contienen el código que se registra en la *Fábrica*, utilizando una subrutina anónima (parecida a una clausura lambda). Para ocultar los detalles, *Fábrica* emula en su método `registrar` control de acceso por paquete.

Un ejemplo: definición de teorema

Para explicar en más detalle qué es lo que se requiere para el filtrado de una orden de ACL2, usaremos como ejemplo el procesado de un supuesto evento de definición de un teorema, `defthm`.

Tras crearse la *Orden* a través de la *Fábrica Abstracta Orden::Fabrica*, se generará la versión XML de la salida de ACL2 en dos pasos. Cada paso se corresponde con uno de los métodos plantilla antes mencionados.

En primer lugar, se captura toda la información disponible en la salida de ACL2 (si se proporciona) y la S-expresión original mediante `inicializar`:

1. `_init_desde_salida` definido por *Orden* retira toda la información de la salida no específica a la *Orden Defthm*, simplificando así su inicialización.

   Ello incluye retirar los mensajes de ACL2, los mensajes del colector de basura del intérprete Lisp y el sumario de la ejecución de la orden.

2. `protegido_inicializar_sexpr`, propio de *Defthm*, obtiene información específica a *Defthm* a partir de su S-expresión:

   - Fórmula a demostrar.
   - Nombre del teorema.
   - Clases de regla a generar, con el valor por defecto ":REWRITE", indicando que se usará una regla de reescritura de miembros.
   - Activación o desactivación de la bandera OTF ("on through the fog"), obligando a ACL2 a que, si con sólo simplificar no basta, no intente inducir sobre la fórmula original, sino sobre las múltiples fórmulas resultantes de las simplificaciones.
   - Captura de otros argumentos de interés.

3. `protegido_inicializar_salida`, definido por *Defthm*, se ocupa de recabar toda la información de interés del texto de la salida de ACL2. Nos interesan las dependencias de almacenamiento de las reglas generadas, y los pasos de la demostración realizada.

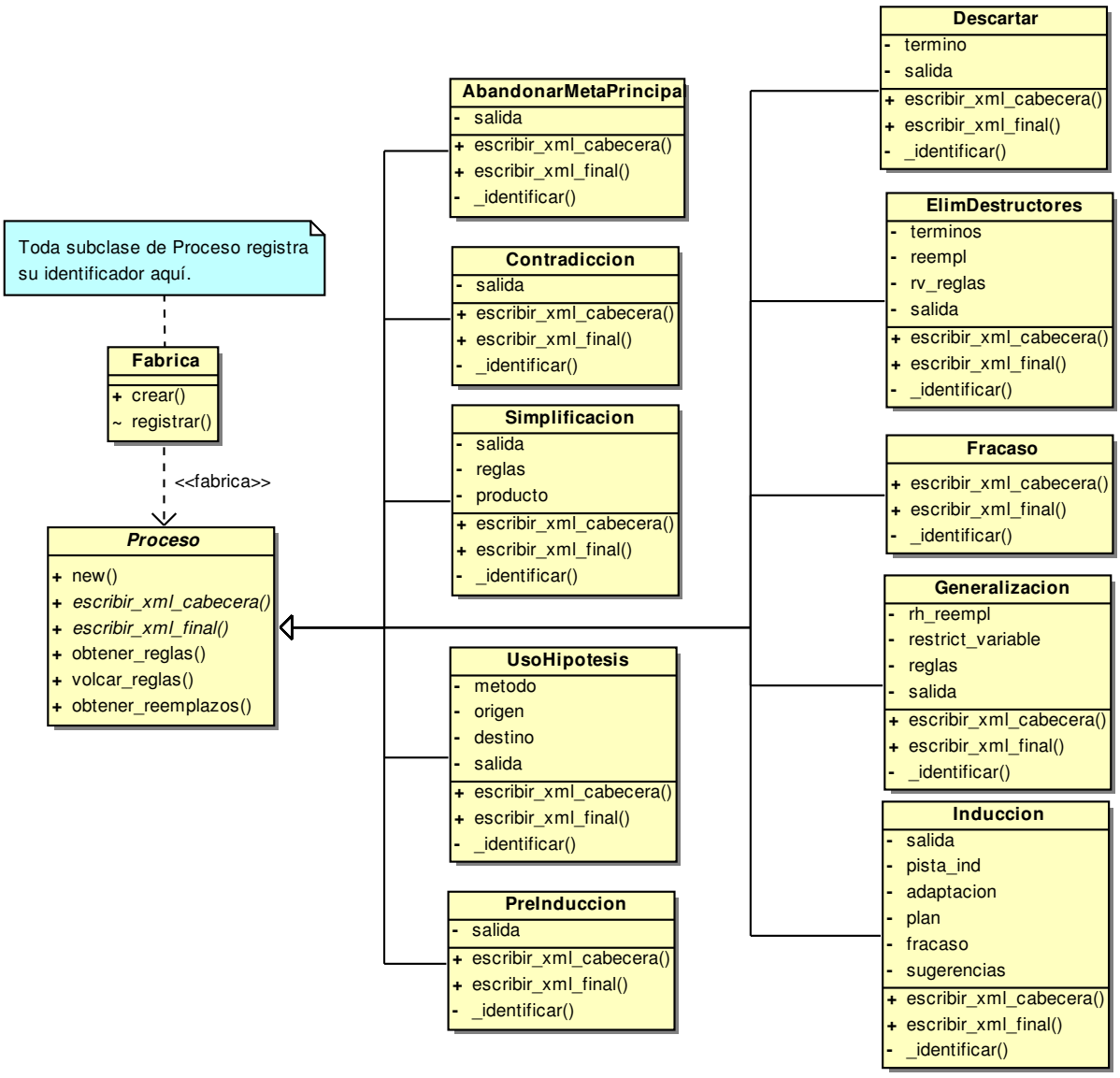


Figura 4.17: Diagrama de clases de Procesos de `ACL2::Procesador`

En primer lugar, recuperaremos la información mediante el método plantilla `inicializar`:

a) Obtener cada una de las metas e inicializarlas. Tras dividir la salida completa en la salida de cada meta, se empleará el *Método Fábrica* `crear` de *Proceso::Fabrica*, que invocará a las subrutinas anónimas registradas para reconocer e instanciar el *Proceso* exacto seguido.

Cada meta introducida se guardará en un vector asociativo para el siguiente paso, utilizando su etiqueta, como "Subgoal *1".

b) Organizar jerárquicamente las metas.

Esto se puede hacer fácilmente a partir de las etiquetas: una meta con la etiqueta "Subgoal *1.1" tiene como padre la meta con la etiqueta "Subgoal *1", por ejemplo.

Gracias al vector asociativo anterior, sólo hay que calcular la etiqueta del padre, recuperarlo y añadirle la submeta.

A continuación, invocaremos al otro método plantilla, `escribir_xml`, para obtener el documento XML que lista toda la información antes recogida:

1. `protegido_cabecera`: escritura de la cabecera de la salida XML correspondiente al `defthm`. Se abre un nuevo elemento `defthm` mediante `startTag` de *XML::Writer*.
2. `_escribir_mensajes`: este método es parte de la funcionalidad implementada por *Orden*, y vuelca los mensajes de ACL2: avisos, observaciones y errores.
3. `protegido_cuerpo`: escritura del cuerpo de la salida XML del `defthm`. Se vuelca la información conseguida antes en `protegido_inicializar`, incluyendo la demostración, delegando en el método `escribir_xml` de *Resultado*.
4. `_escribir_sumario`: también parte de la funcionalidad predefinida por *Orden*, vuelca el sumario de la ejecución del evento, incluyendo tiempos, reglas usadas y avisos dados.
5. `protegido_final`: en este caso, `defthm` no redefine la funcionalidad predefinida por *Orden*, cerrándose el elemento inicial escrito en `protegido_cabecera` con un elemento del mismo nombre que la orden.

Se ha descrito también de manera gráfica este proceso utilizando diagramas de secuencia UML 1.4. Se necesitó dividir el diagrama en dos: el primero, dedicado al análisis, es la figura 4.18 de la página 107, y el segundo, dedicado a la salida, es la figura 4.19 de la página 108.

Por razones de espacio, algunas de las líneas de vida no se extienden hasta el final de la hoja, sin que esto implique su destrucción. El nivel de detalle escogido no describe las operaciones internas de *Orden* o *Defthm* que no requieran interacción por parte de otras clases, o los mensajes de poco interés enviados a *XML::Writer*.

Los vectores asociativos de Perl han sido modelados como objetos de la clase *Hash*, por uniformidad.

4.5.3. Capa de filtrado: YAXML::Reverse

Descripción general

Este conversor implementa la correspondencia XML → YAML conocida como YAXML [60] en sentido inverso, con dos objetivos en general:

- En primer lugar, para permitir a XMLEye abrir cualquier documento basado en el metalenguaje YAML, o su subconjunto (desde YAML 1.1) JSON. Ello permitirá tratar una familia completa nueva de formatos.
- En segundo lugar, para todos aquellos que necesiten aplicar alguna herramienta a sus documentos que no esté disponible para YAML, como transformaciones definidas de forma funcional mediante XSLT.

El proceso es relativamente sencillo:

1. Se analiza el documento origen YAML, deserializándose a una estructura de datos Perl. Este primer paso se consigue mediante el módulo Perl `YAML::XS`, un *binding* que permite aprovechar la funcionalidad de la biblioteca `libyaml` (disponible en `http://pyyaml.org/wiki/LibYAML`), y analizar documentos YAML 1.1 y JSON. Existen una serie de restricciones, que veremos posteriormente en el apartado 4.5.3 de la página 111.
2. Se representa dicha estructura como un árbol XML, con una versión mejorada del formato sugerido por YAXML para tratar ciertos casos especiales.

Sin embargo, hay una serie de detalles de interés en la inversión de YAXML que consideramos merece la pena mencionar. También se han encontrado algunas limitaciones debidas en primer lugar al módulo `YAML::XS`, y en última instancia al propio lenguaje Perl, pero se estima que tendrán realmente poco impacto en su uso.

Regeneración de anclas y alias en el fichero XML

Hay una diferencia muy importante en los modelos de información comúnmente usados en YAML y XML: el modelo de información de presentación de YAML, tal y como se comentó en §4.4.4 (página 84), utiliza grafos dirigidos con raíz, y el modelo DOM comúnmente usado en XML emplea árboles. Esto implica dos problemas, fundamentalmente:

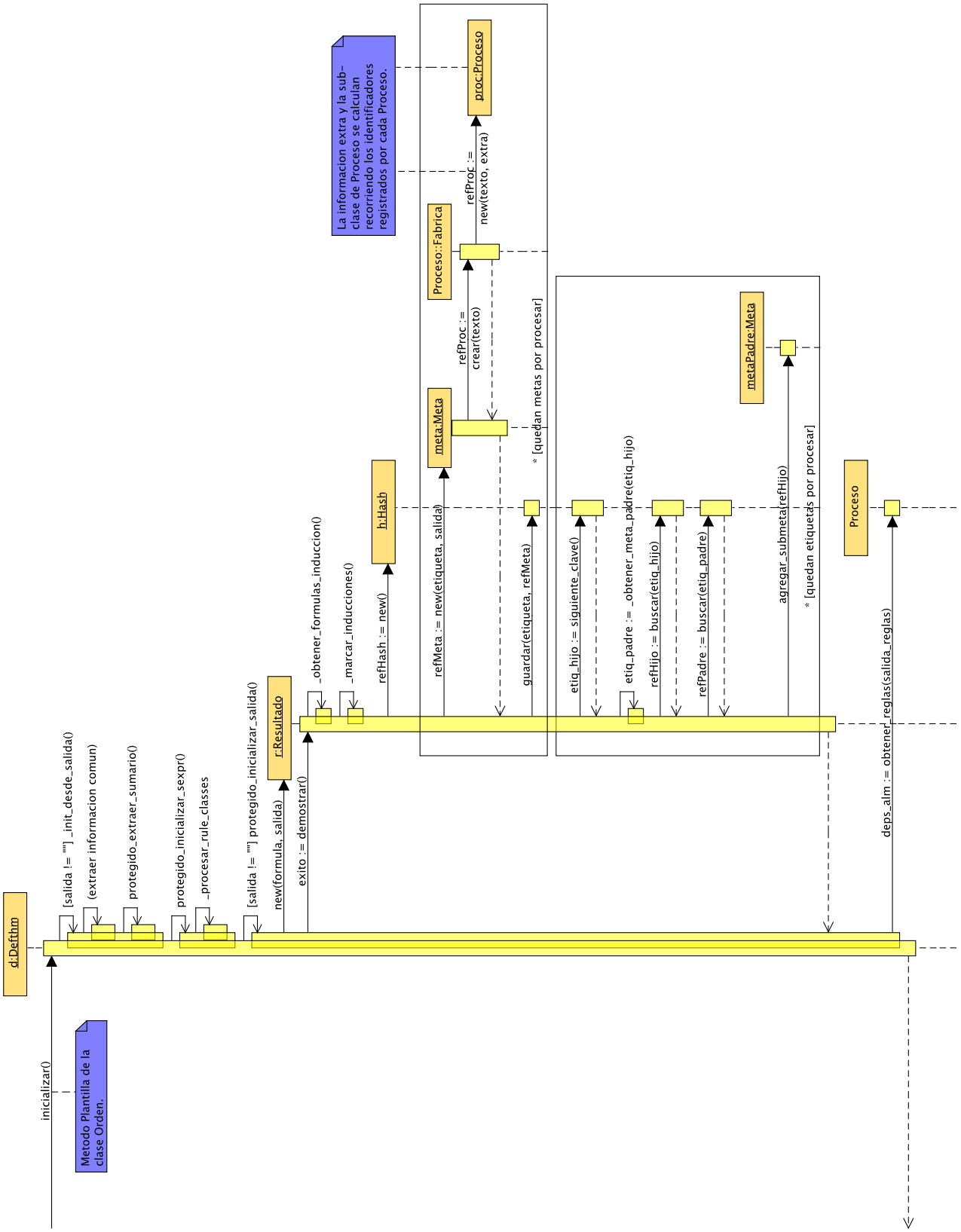


Figura 4.18: Diagrama de secuencia de filtrado de `defthm` (análisis)

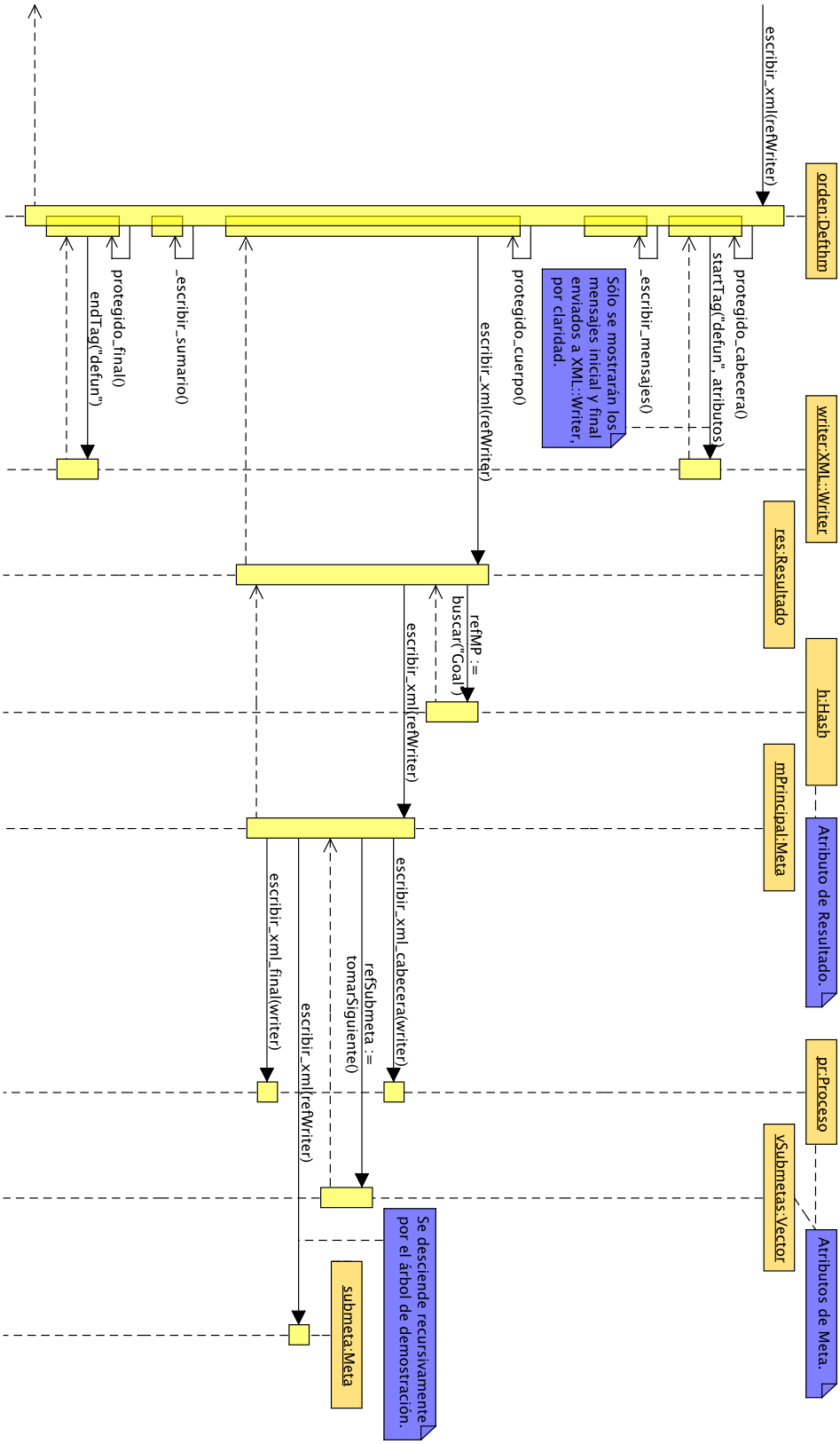


Figura 4.19: Diagrama de secuencia de filtrado de `defthm` (generación de salida)

- Un primer problema, de menor importancia, es el hecho de que si un nodo del grafo tiene varios arcos entrantes, tendrá que ser repetido varias veces en el árbol final. Este problema se agrava cuando el nodo a repetir no es un nodo hoja, sino un nodo raíz de un subgrafo: tendremos que repetir el subgrafo completo. De todas formas, este problema es más una cuestión de eficiencia, y en ciertos casos se podría incluso aceptar el coste adicional.
- El problema mayor es que en los grafos de YAML, se admiten ciclos: no existe forma de representar directamente estas estructuras cíclicas en un árbol XML.

La forma en que YAXML resuelve estos problemas es utilizando no el modelo de información de representación, sino el modelo de información de serialización, que sí está basado en árboles. En dicho modelo, se definen los conceptos de *ancla* y de *alias*, pudiendo así establecer enlaces entre los nodos del árbol, y resolver los dos problemas anteriores. En YAXML, se representan como atributos de un elemento sin contenido (como `<a/>`). Pueden compararse los resultados para un caso sencillo en la figura 4.20 de la página 110.

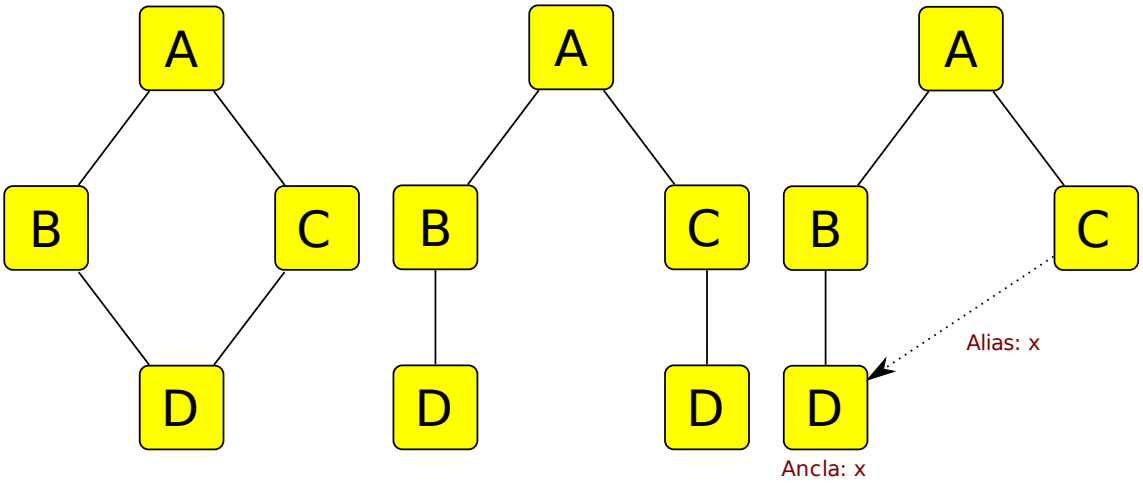
Existe un problema, sin embargo: utilizando `YAML::XS`, lo que obtenemos es el grafo final. Para reconstruir el árbol con anclas y alias a partir del grafo, hemos de distinguir qué elementos del grafo constituyen anclas y qué elementos constituyen alias a esas anclas. Podemos aprovechar el hecho de que la estructura creada por `YAML::XS` se basa en referencias a direcciones de memoria con vectores asociativos, listas y objetos: si en dos o más puntos del árbol aparecen referencias a la misma dirección de memoria, sabremos que la primera aparición será el ancla, y todas las demás serán sus alias.

El mayor problema es que no sabremos si un nodo constituye un ancla hasta que sea usado como tal por primera vez en cualquier otro punto del documento. Por ello, tendremos entonces que dividir la generación de la salida en dos pasos:

1. En una primera pasada, se recopila en un vector asociativo qué referencias constituyen anclas, junto con su posición en orden de uso, a partir de la cual se generará el identificador final.
2. En la segunda pasada, se genera el código XML, cuidando de generar los atributos `anchor` y `alias` en los nodos apropiados.

Pruebas de unidad automáticas basadas en equivalencia

Un problema fundamental en la realización de pruebas de software es la elaboración de los *oráculos* [4] necesarios para comprobar si el comportamiento del software es el esperado. Existe una gran variedad de tipos, según la naturaleza del software bajo prueba.



(a) Grafo original (b) Árbol sin enlaces (c) Árbol con enlaces

Figura 4.20: Representación de grafos mediante árboles XML

En lo que respecta a `YAXML::Reverse`, necesitamos un oráculo que nos diga si un documento YAML ha sido convertido a XML sin pérdidas ni cambios en su información a nivel semántico. Lo ideal es que sea lo más automático posible: no es factible programar un caso de prueba para cada entrada. No podemos hacer comparaciones directas del texto, pues un documento YAML puede ser escrito en muchos estilos distintos, y los documentos XML también disponen de ciertos mecanismos para el manejo del espaciado, por ejemplo.

Una posibilidad es generar un analizador de los documentos XML obtenidos, que construyera representaciones del estilo de las de `YAML::XS` y las comparara con los grafos originales. Hay una forma de conseguir algo similar con un esfuerzo mucho menor: podemos aprovechar el hecho de que combinando YAXML y `YAXML::Reverse` cerramos el ciclo en torno a YAML y XML: podemos así hacer una transformación YAML → XML → YAML, y comparar los grafos resultantes de procesar los documentos YAML original y final. Si son equivalentes, dado que YAXML opera únicamente sobre el documento XML, sabremos que éste contiene toda la información del YAML original sin cambios. Se puede ver un esquema del proceso completo en la figura 4.21 de la página 111.

El principal inconveniente de este enfoque es que YAXML también introduce sus propios fallos, y éstos son vistos como fallos de `YAXML::Reverse` por el conjunto de pruebas. Esto no tiene por qué ser malo: de hecho, puede verse como una forma de depurar los dos programas al mismo tiempo, usando `YAXML::Reverse` como parte del oráculo de YAXML y viceversa.

Tiene la ventaja de ser completamente automático: añadir un nuevo caso de prueba es simplemente añadir un fichero `.yaml` bajo el directorio `t/testInputs`.

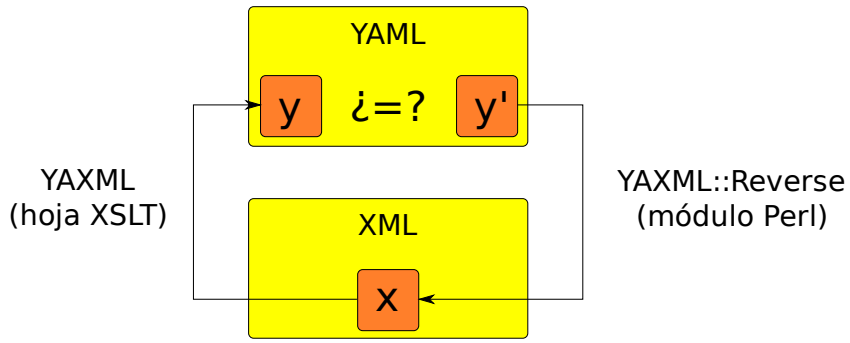


Figura 4.21: Oráculo de pruebas para YAXML::Reverse

Limitaciones de Perl frente a YAML

YAML se caracteriza por ser una especificación muy flexible, pudiendo combinar los tipos básicos de nodos y etiquetas de muchas formas. Este fragmento de la especificación [7], que describe a los nodos de vector asociativo, es particularmente interesante:

> The content of a mapping node is an unordered set of key: value node pairs, with the restriction that each of the keys is unique. YAML places no further restrictions on the nodes. In particular, keys may be arbitrary nodes, the same node may be used as the value of several key: value pairs, and a mapping could even contain itself as a key or a value (directly or indirectly).

Como vemos, podemos tener claves no escalares e incluso referencias cíclicas de un mapa como clave de uno de sus elementos. El problema de los ciclos ya ha sido superado gracias a las anclas y alias, pero usar claves no escalares es más difícil de resolver: los vectores asociativos de Perl, usados como estructura de datos nativa por `YAML::XS`, no admiten claves no escalares.

Existen módulos como `Tie::RefHash` que permiten usar clases que implementan vectores asociativos con claves escalares y no escalares como si fueran vectores asociativos normales, pero no los reemplaza directamente, sino que modifica la forma en que Perl evalúa el código que usa el identificador de la variable "sobrecargada". Es una técnica de bajo nivel: no podemos pasar ese vector asociativo y usarlo en una subrutina como de costumbre, por ejemplo. Además, y lo que es más grave, no nos sirve una vez `YAML::XS` ha terminado su procesado: tendría que usarse antes de asignarle cualquier elemento.

Si usáramos los diccionarios de Python, por ejemplo, uno podría imaginarse emular claves basadas en nodos secuencia mediante tuplas, y claves basadas en nodos de vector asociativo usando tuplas de tuplas. Python no permite usar listas ni diccionarios

como claves. Sin embargo, al igual que en el caso de Perl, todo dependería del *binding* que estuviéramos usando en ese momento.

De todas formas, examinando los lenguajes basados en YAML disponibles actualmente, no parece ser una restricción importante: aún no he podido encontrar un solo formato basado en YAML que tenga claves no escalares. JSON sencillamente no tiene este problema.

Por ello, por lo pronto esta restricción se considera de baja prioridad: solventarla requeriría la reescritura completa de `YAXML::Reverse` bajo otro entorno (seguramente una aplicación escrita en C, C++ o Python), y no supondría realmente una ventaja para su uso general.

4.5.4. Capa de aplicación

La capa de aplicación se ocupa de implementar las clases del modelo de la arquitectura MVP, y de proveer servicios de alto nivel independientes de la interfaz gráfica. Esto permite tener separados todos los aspectos de la interfaz gráfica respecto de aquellos relacionados con la lógica básica de XMLEye.

Veremos a continuación cuáles son esas clases del modelo y servicios ofrecidos, y qué diseño siguen.

Modelos de documentos

El concepto fundamental en XMLEye es el de un *Documento* XML abierto por el usuario y preprocesado. Esta clase se ocupa de encapsular toda esa información y gestionar los cambios en la hoja de preprocesado utilizada y la realización de búsquedas.

Nótese que no se ocupa de la hoja de visualización: como detalle de presentación de que se trata, este aspecto se gestiona en su modelo de presentación, que veremos en §4.5.5 (página 122).

Este modelo de documento integra los predicados de búsquedas mediante palabra completa e ignorando minúsculas definidos mediante extensiones XPath, dejando a las capas superiores únicamente la responsabilidad de definir las *PeticionBusqueda* necesarias.

El conjunto de todas las clases empleadas, tanto del J2SE como de XMLEye, se halla en el cuadro 4.22 (página 113).

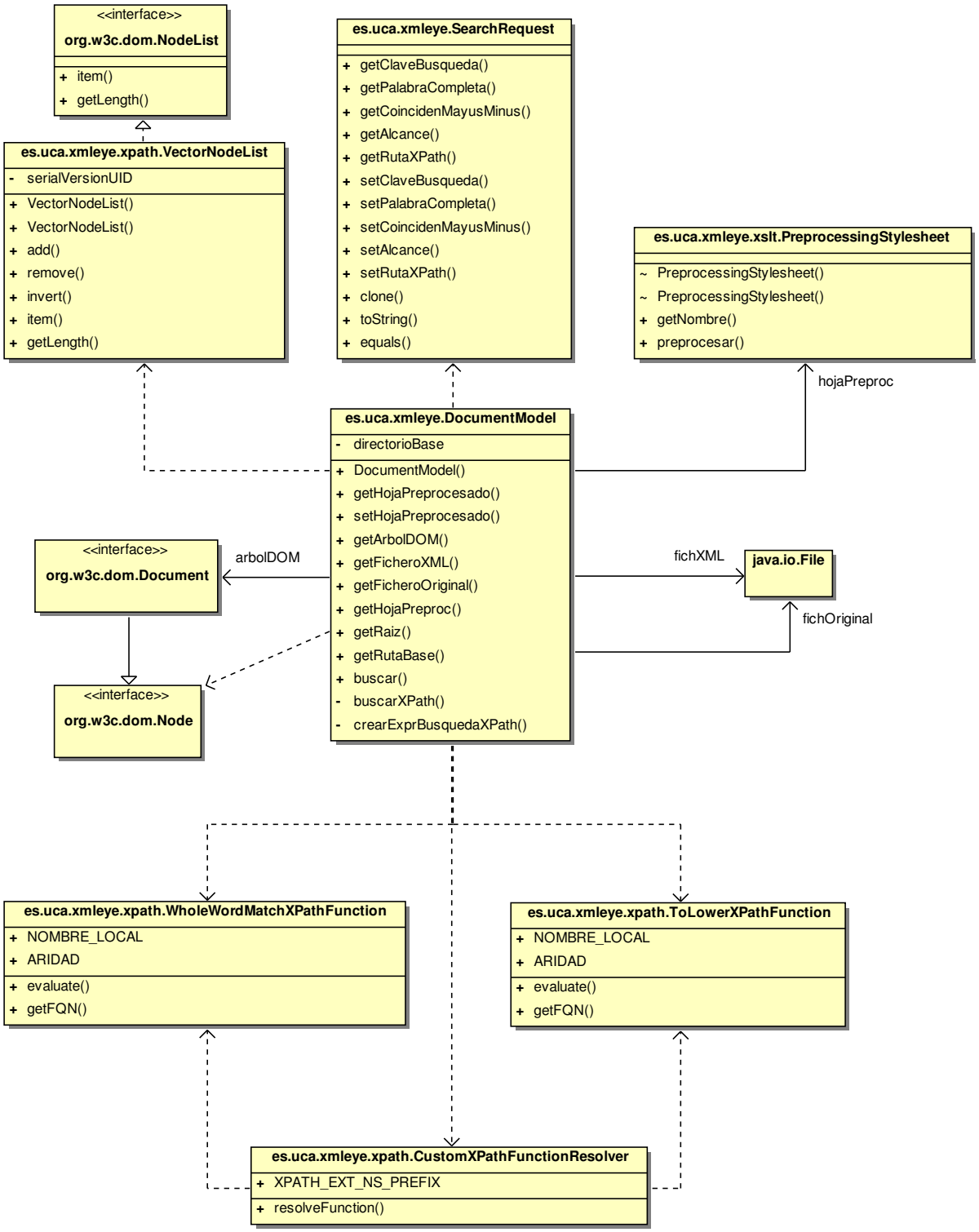


Figura 4.22: Diagrama de clases de diseño para modelado de documentos

Hojas de estilos

Las hojas de estilos participan tanto en la capa de presentación como en la capa de aplicación. En la capa de aplicación se halla toda la infraestructura necesaria para su ejecución, atendiendo a su estructuración en repositorios y soporte para localización, en combinación con el uso de herencia entre distintas hojas. Las propias hojas son parte de la capa de presentación.

Distinguiremos entre:

Hoja de usuario Conjunto cohesivo de una o más hojas XSLT individuales seleccionable por el usuario que conforma un modo de preprocesar el documento o visualizar un nodo.

Se distinguen dos tipos:

Preprocesado Contienen transformaciones que afectan al documento XML completo tras su apertura. Se usan para modificar el árbol del documento o para realizar operaciones costosas como el establecimiento de enlaces.

Visualización Contienen transformaciones a ejecutar sobre el nodo DOM seleccionado por el usuario en cada momento.

Toda hoja de usuario contiene una o varias hojas XSLT especiales cuyo nombre de fichero sigue el patrón `(nombre hoja)[idioma[_país]].xsl`, donde [] indica una parte opcional y () una parte obligatoria. Llamamos a dichas hojas sus *puntos de acceso*. Se intentará antes acceder a los puntos de acceso más específicos respecto del país e idioma de la localización actual.

Hoja XSLT Fichero individual que contiene un documento XML definido sobre el vocabulario W3C XSLT.

Las hojas de usuario han de organizarse cumpliendo una serie de requisitos:

- Han de poder ser fácilmente localizables según el país y el idioma de la localización establecida por el usuario por defecto en su sistema.
- Han de ser fáciles de instalar, sin requerir configuración alguna.
- Han de ser fáciles de elaborar, pudiendo basarnos en hojas de usuario ya existentes.

Los detalles de implementación de cada tipo de hoja de usuario y aquellos comunes a toda hoja de usuario han sido encapsulados en clases, permitiendo a la capa de presentación realizar transformaciones sin conocer los detalles de la biblioteca XSLT usada.

En cuanto a los requisitos sobre la organización de las hojas de usuario, se ha mantenido la separación de responsabilidades moviendo estos requisitos a una clase que actúa de *repositorio*, que se ocupa de localizar las hojas de usuario y configurarlas para

conseguir al mismo tiempo la capacidad de reutilizar código y localizar las hojas. La carga del repositorio se realiza al iniciar XMLEye.

La organización general de las clases utilizadas se halla en la figura 4.23 de la página 117.

Organización del repositorio El esquema de almacenamiento del repositorio no es configurable, siendo así posiblemente menos flexible, pero muy sencillo de utilizar. La organización de las hojas de usuario se explica mejor con un ejemplo. Una posible distribución de hojas XSLT de una hipotética hoja de usuario de visualización llamada `perso` bajo el directorio base con la ruta relativa `xslt` sería:

- `xslt/view/perso/perso.xsl`: punto de acceso inicial para la localización por omisión.
- `xslt/view/perso/perso_en.xsl`: punto de acceso inicial para la localización inglesa, sin especificar el país.
- `xslt/view/perso/perso_en_GB.xsl`: punto de acceso inicial para la localización inglesa en el Reino Unido.
- `xslt/view/perso/logica.xsl`: hoja complementaria que permite que los tres idiomas compartan la misma lógica.

Si fuera una hoja de preprocesamiento, se usaría `preproc` en vez de `view`.

Extensibilidad Las hojas de usuario son importadas por la hoja principal no modificable de su tipo, en la raíz del repositorio, que define la mínima funcionalidad común: `view.xsl` para las hojas de visualización y `preproc.xsl` para las de preprocesado.

Además, una hoja de usuario puede importar a otras y así especializarlas. Es el caso de `ppACL2`, que importa a `xml`.

Para abstraer a las hojas de la lógica de internacionalización, una hoja de usuario que desee importar a otra puede usar URI (Uniform Resource Identifier) especiales como `view_ppACL2`.

Mediante esta URI, se accederá a la hoja de usuario de visualización `ppACL2` a través del punto de acceso que mejor se ajuste a la localización actual del usuario.

Internacionalización El ejemplo de `perso` anterior ilustra el mecanismo de localización establecido por la lógica de la capa de aplicación. Dicho mecanismo imita al usado por los *ResourceBundle*s de la biblioteca estándar de Java.

Si la localización actual por defecto es "es_ES" (español, España) y se busca la hoja de visualización `ppACL2`, *ControlHojasXSL* intentará localizar un punto de acceso en el siguiente orden:

1. `xslt/view/ppACL2/ppACL2_es_ES.xsl`
2. `xslt/view/ppACL2/ppACL2_es.xsl`
3. `xslt/view/ppACL2/ppACL2.xsl`

Descriptores de formatos

Otro servicio también a prestar por la capa de aplicación es el de la gestión de los descriptores de formatos de documento. Al igual que en el caso de las hojas, se necesitaba abstraer a las capas superiores de los detalles de un descriptor individual y de la forma en que se hallan organizados.

Volvemos a tener otro repositorio, esta vez de descriptores, en el que se siguen una serie de convenciones para evitar la necesidad de realizar cualquier tipo de configuración. A diferencia del repositorio de hojas, no tratamos con un único árbol de directorios: se pueden encadenar varios, y así poder tener una serie de ajustes a nivel global para todos los usuarios, y otros locales para el usuario actual. Para restaurar los ajustes locales a los globales, sólo hemos de borrar el descriptor a nivel local y actualizarlos.

Este repositorio abstrae también la lógica que hace corresponder un descriptor a un fichero determinado, y se ocupa de validar contra un XML Schema todos los descriptores. El repositorio es un *AbstractListModel* de Swing, por lo que puede integrarse fácilmente como modelo de un formulario e incluso registrar *Observador*es sobre él si se estima necesario, pero no es obligatorio.

Por otro lado, las clases para los descriptores individuales encapsulan el formato de estos ficheros y proporcionan toda la validación necesaria de su contenido, lanzando excepciones cuando se intentan establecer valores no válidos en algún atributo.

Las clases implicadas se hallan en el cuadro 4.24 de la página 118.

Notificación de cambios sobre ficheros

Un requisito muy interesante era el de poder vigilar los cambios realizados sobre los documentos durante su visualización, pudiendo mantener el editor y XMLEye abiertos en paralelo. Para ello se ha implementado una clase que ofrece una interfaz basada

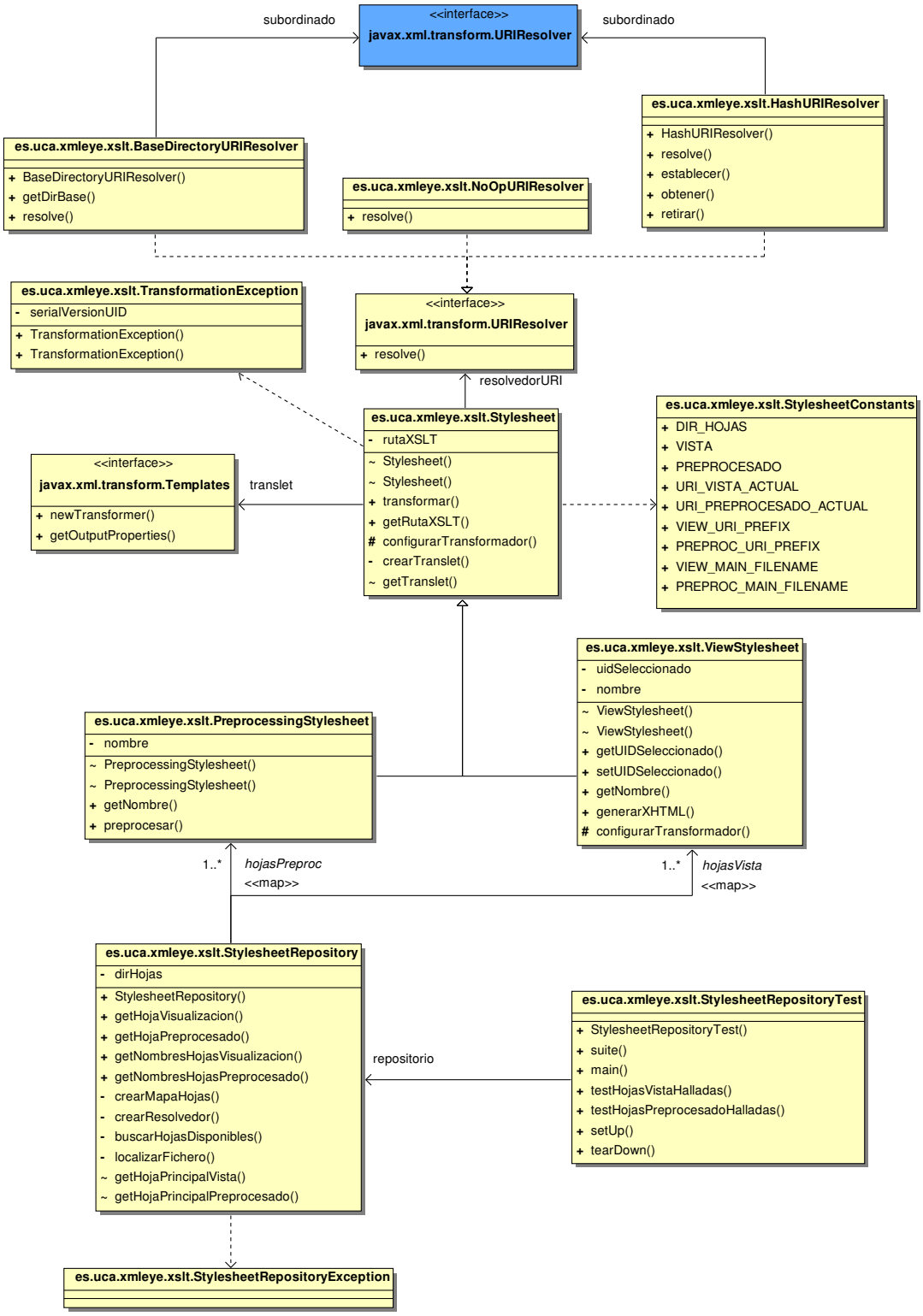


Figura 4.23: Diagrama de clases de diseño para las hojas de usuario

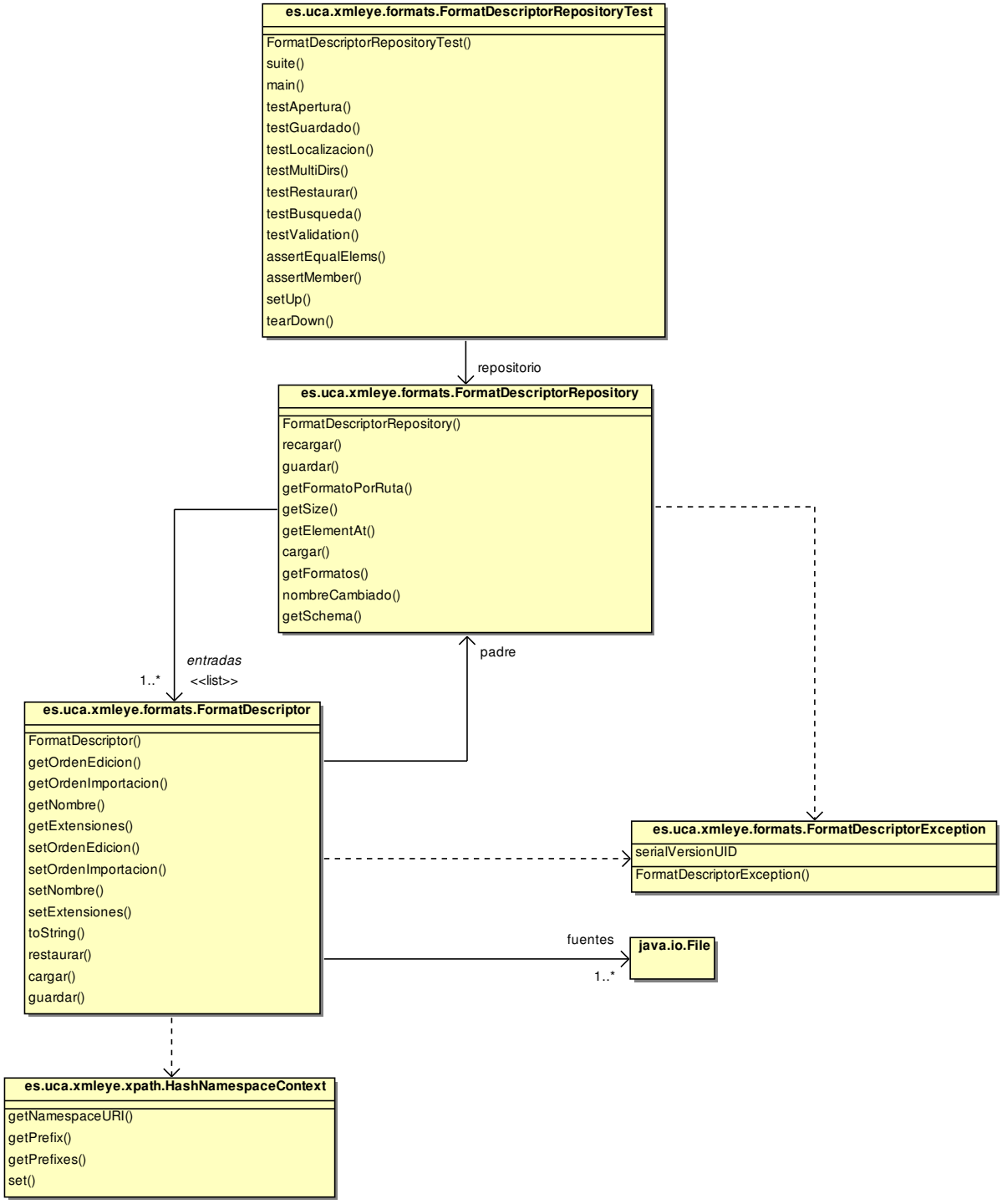


Figura 4.24: Diagrama de diseño de clases de descriptores de formatos

en el patrón Observador, que se basa en la fecha de modificación para notificar de cambios en los contenidos del fichero.

Para ello, se usa un hilo en segundo plano de baja prioridad, que se despierta cada cierto tiempo, realiza la comprobación y vuelve a dormir. Esto evita la pérdida de rendimiento que supondría un bucle de espera activa. Además, las actualizaciones pueden pausarse y reactivarse a voluntad.

Persistencia de preferencias

En este caso es fundamental tener un punto de acceso central a todas las preferencias de los usuarios. Por estas razones, en la versión original del visor se utilizó el patrón *Singleton*, asegurando la existencia de una única instancia de una clase que implementara una interfaz. Se utilizó una versión mejorada de la implementación usual tomada de [30], simplificando sobre la sugerida en [54].

Sin embargo, durante el desarrollo de esta nueva versión, se vio que el uso del Singleton dificultaría la posterior realización de pruebas con sustitutos, e introducía una dependencia innecesaria.

Para retirarla, se aplicó el patrón *Inyección de Dependencias*: ahora no son las clases las que solicitan el gestor de preferencias, sino las que lo reciben a través de su constructor. De esta forma, no se necesita el punto único de acceso como antes, siendo muy fácilmente sustituible durante pruebas, por ejemplo, y manteniendo la independencia respecto a la implementación a través del uso de una interfaz.

El diagrama de las clases involucradas en la gestión de preferencias se halla en el cuadro 4.25 de la página 120. Podemos ver que el *Singleton* ha desaparecido, al ser ahora completamente innecesario.

Se dispone de un valor booleano almacenado directamente en dependencias, de tal forma que una clase que requiera un atributo con persistencia entre las preferencias sólo tendrá que instanciar apropiadamente esta clase y luego utilizar get() y set() sin más complicaciones.

En el diagrama de secuencia 4.26 de la página 121 puede verse cómo se gestionan las preferencias a lo largo del programa:

- Al iniciarse XMLEye, se crea una instancia de la clase principal que representa a toda la aplicación. Esta clase es la ocupada de ensamblar a todas las demás clases de mayor nivel, inyectando las dependencias consideradas oportunas y evitando que dependan demasiado entre sí.
- La clase principal ensambladora crea una instancia de la implementación de gestión de preferencias a usar, y solicita que se carguen las preferencias.

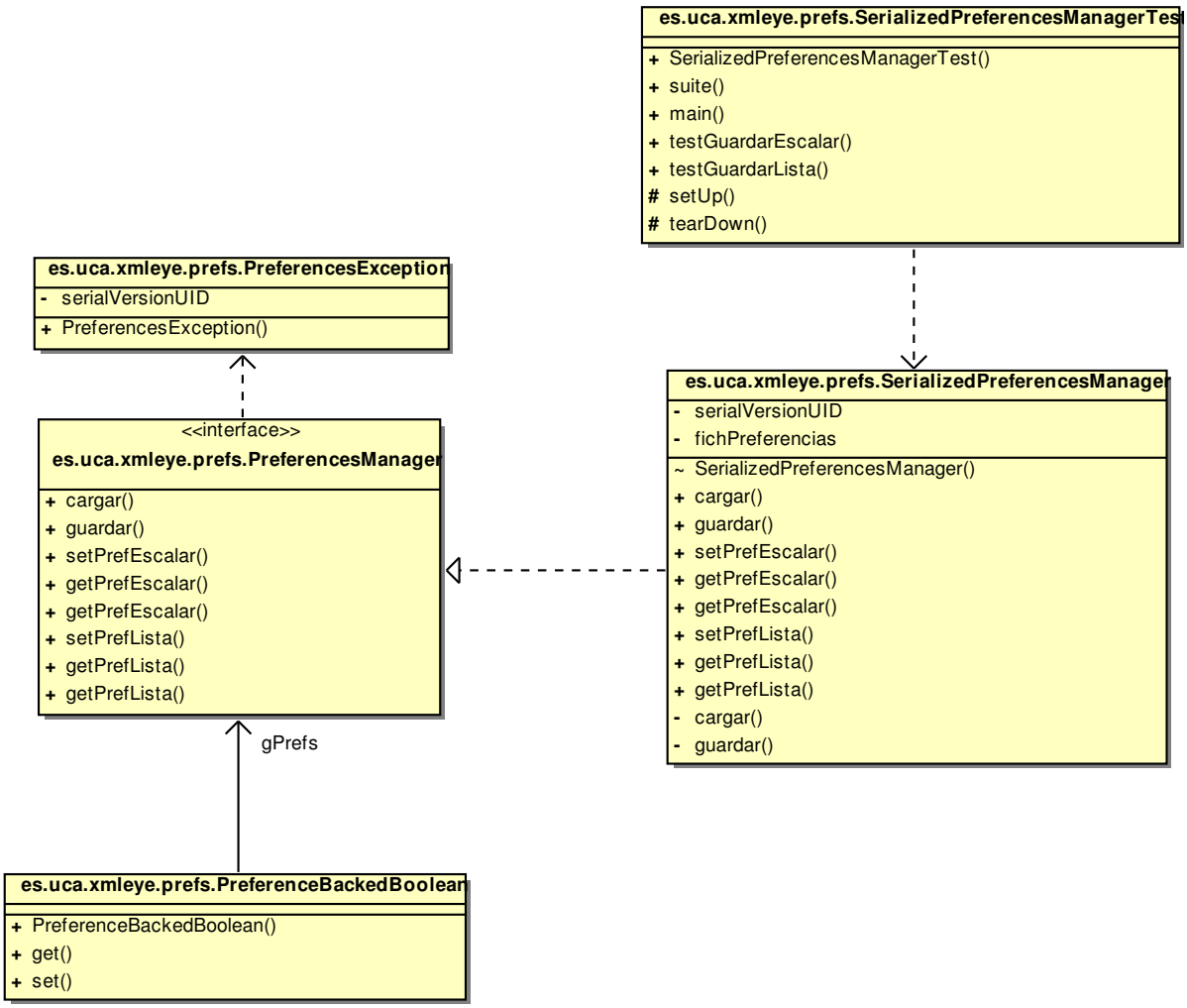


Figura 4.25: Diagrama de clases de diseño de gestión de preferencias

- La clase principal ensambla el modelo de presentación de la ventana principal con el gestor de preferencias (visto a través de su interfaz) y otros parámetros. También ensambla y lanza la ventana principal.
- Al cerrar la aplicación, la misma clase principal solicita el guardado de las preferencias antes de terminar la ejecución del programa.

4.5.5. Capa de presentación

La capa de presentación se encarga de la comunicación persona-máquina, realizando peticiones sobre la capa de aplicación y mostrando los resultados.

Durante el diseño de la interfaz, se usaron las obras [34, 35] como guía, como en el caso del diseño del diálogo “Buscar”, donde el reparto de los componentes sigue

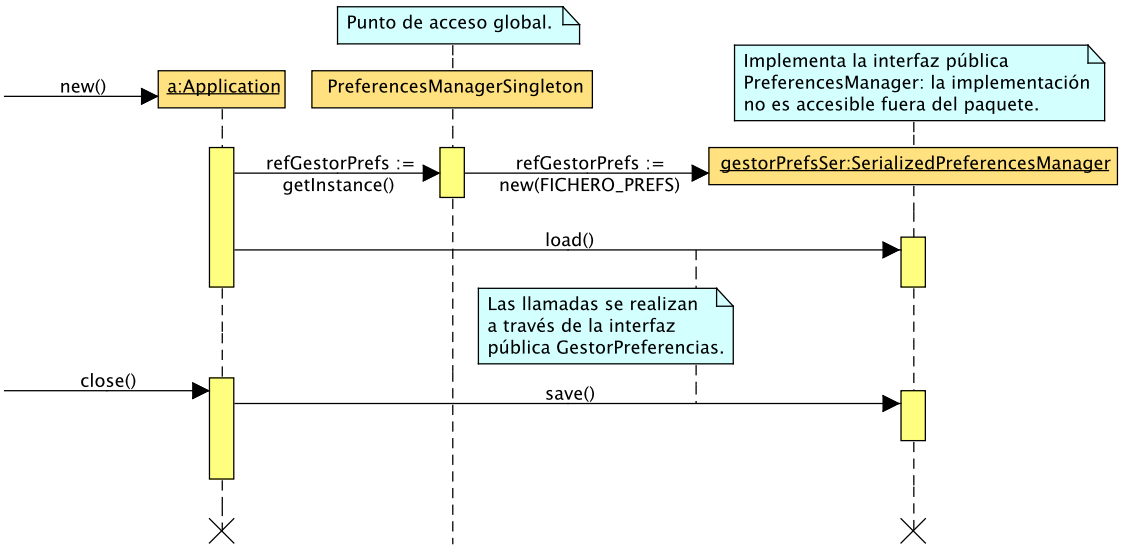


Figura 4.26: Diagrama de secuencia de gestión de preferencias

sus recomendaciones. Otras recomendaciones seguidas son la ejecución de las tareas largas en segundo plano, los elementos de los que deben constar los menús, o las combinaciones de teclas recomendadas para cada opción.

Estructura general

Como ya se comentó en § 4.5.1 (página 91), la capa de presentación se ocuparía de los elementos Vista y Presentación del trío Modelo-Vista-Presentación. En particular, se utilizaría la variante *Modelo de Presentación*, que nos permite definir la lógica de una forma completamente separada de la interfaz, y así poder crear nuevas interfaces de manera mucho más libre.

XMLEye dispone básicamente de 4 formularios, tal y como se estableció durante el análisis (apartado 4.3.1 en la página 61): el formulario principal, el formulario de gestión de descriptores de formatos, el formulario de realización de búsquedas, y por último el formulario con información acerca de XMLEye.

Descartando el último de este apartado por su sencillez, podemos ver que el formulario principal y el de gestión de formatos son los más complejos. Por ello, modelaremos su estado con *Modelos de Presentación*, permitiendo una mejor comprensión y depuración de su código. El formulario de búsqueda es más sencillo, y por lo tanto no requiere dicho enfoque, como veremos posteriormente.

Además de los modelos de presentación, hablaremos en el resto de esta sección de otros aspectos interesantes del diseño de la capa de presentación.

Modelos de presentación y su ensamblaje

En los formularios más complejos, se pudo ver rápidamente que el uso del patrón *Observador* para aspectos como el control de hojas usadas y demás hacía el código cada vez más difícil de estructurar y depurar. Además, este código impedía poder aislar de manera efectiva el estado de múltiples documentos.

Por ello, se cambió a un enfoque donde unas clases encapsulan el estado de la interfaz, de forma independiente a los controles Swing. Estas clases ofrecen una serie de métodos `get` públicos para acceder al estado de la interfaz, y disponen también de un método por cada gesto de interés que pueda provenir del usuario.

Las vistas se reducen, por lo tanto, a informar de los gestos del usuario al modelo de presentación y posteriormente sincronizarse con los modelos. Al quedarse sin la mayor parte de su lógica, el hecho de no poder realizar pruebas sobre los propios controles Swing no es tan importante: podemos hacer pruebas sobre el modelo de presentación, enviando los gestos y estudiando su estado posterior.

Los modelos de presentación pueden anidarse: así, el modelo de presentación del formulario principal contiene a los modelos de presentación de cada documento, que contienen a su vez a los modelos (ya no de presentación) de los documentos correspondiente, junto con otros elementos como una referencia al repositorio de hojas o al repositorio de descriptores de formato. Entre los campos de todo modelo de presentación se encuentra el nodo seleccionado actualmente, la hoja de visualización que está siendo usada o la última búsqueda realizada, por ejemplo.

Los modelos de presentación dependen de otros objetos, como ya hemos visto, pero no deberían quedarse acoplados a una implementación o instancia particular de ellos. En su lugar, estas dependencias se proporcionan a través del constructor, constituyendo una *Inyección de Dependencias*. Los modelos de presentación y sus dependencias y la vista del formulario principal son creados y configurados por un ensamblador central: la clase principal de la aplicación. Esto asegura que toda la lógica relevante se halle en un único sitio, y el resto de las clases se mantengan lo más flexibles posible.

El diagrama de clases que describe las relaciones entre las distintas vistas principales (formularios, listas de pestañas y pestañas) y los modelos de presentación se halla en el cuadro 4.27 de la página 123.

Diseño e integración del formulario de búsqueda

El formulario de búsqueda es una excepción al uso del patrón MVP. Este diálogo carece prácticamente de lógica propia, siendo únicamente una interfaz a partir de la cual el usuario puede construir la petición de búsqueda y enviarla a sus suscriptores, entre los cuales se halla el modelo de presentación del formulario principal.

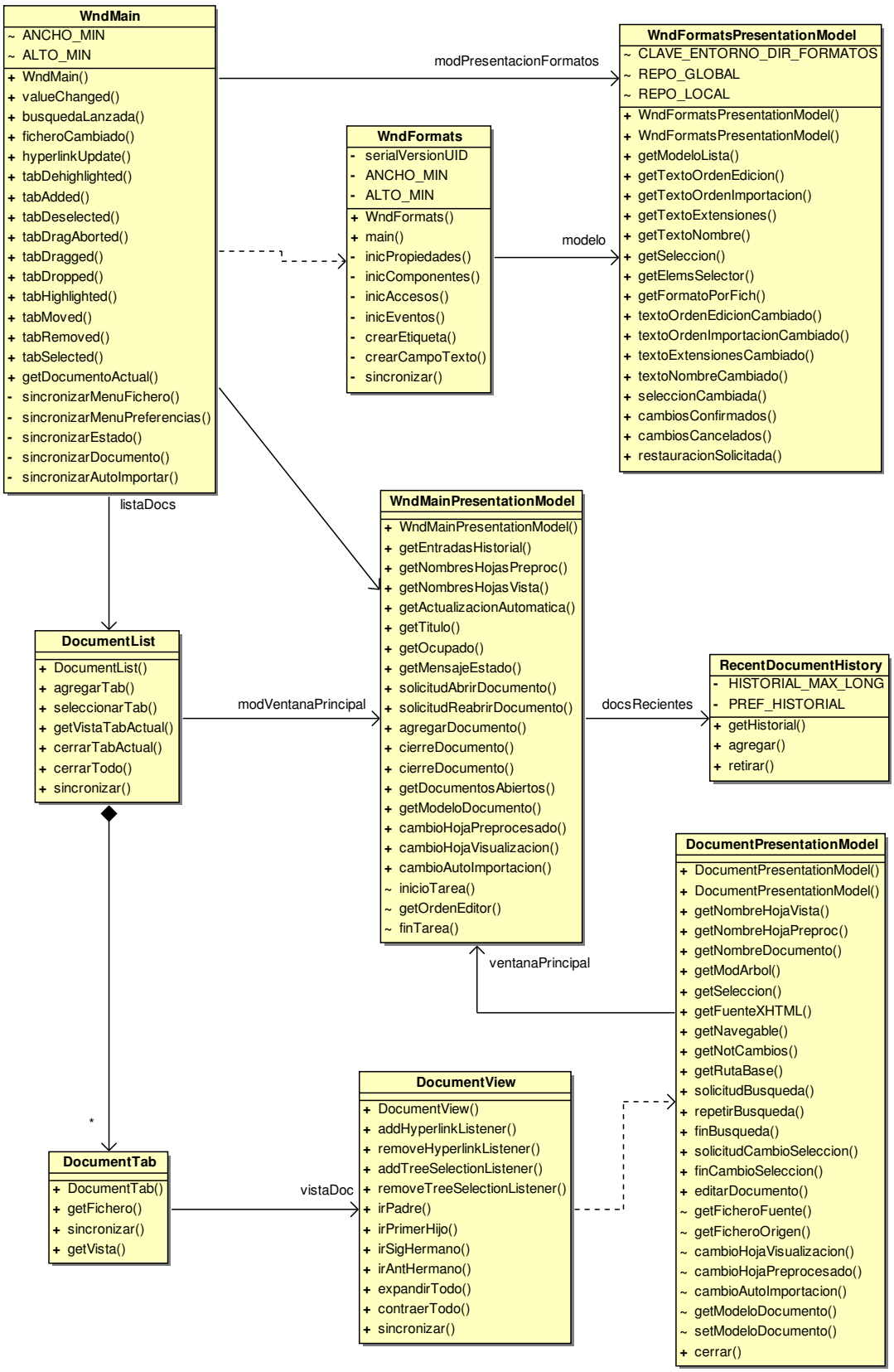


Figura 4.27: Diagrama de clases de diseño de los modelos de presentación

Aplicando de esta forma el patrón *Observador*, se evita el acoplamiento del formulario de búsqueda al formulario principal, y se puede guardar el evento para posteriormente repetir la búsqueda sin necesidad de volver a mostrarlo, por ejemplo.

El formulario principal, como es común, se limita a notificar del gesto al modelo de presentación, que utiliza la funcionalidad del modelo del documento para implementar la búsqueda, añadiendo algo más de inteligencia en lo que respecta a búsquedas repetidas.

Puede verse el diagrama de clases de diseño correspondiente en el cuadro 4.28 de la página 125.

Gestión de diálogos de mensajes

A diferencia del caso de §4.5.4, se ha mantenido el uso del patrón *Singleton* para la gestión de diálogos de mensaje: el punto de acceso global en este caso es realmente necesario, y la lógica que provee es muy concreta.

En particular, la clase a la que da acceso el *Singleton* permite mostrar esos diálogos de forma independiente al hilo en que nos hallemos: normalmente, para poder lanzar un diálogo debemos hacerlo en el hilo de Swing, o se crearán condiciones de carrera indeseadas. Esta clase hace uso de las *SwingUtilities* para enviar las tareas apropiadas al hilo de Swing desde cualquier otro de los hilos, como cuando estamos convirtiendo un documento de entrada, por ejemplo.

Sin embargo, en este caso, no se ha dividido en una interfaz y una implementación, ya que a diferencia de la gestión de preferencias, no se están considerando implementaciones alternativas.

Presentación de árboles XML dirigidos por datos

Los componentes usados para visualizar árboles DOM XML emplean, como todo componente Swing, una modificación [24] del patrón *Modelo-Vista-Controlador* original de [30].

En esta versión modificada, llamada “arquitectura de modelo separable” por Sun y similar al patrón *Documento-Vista* mencionado en [10], se dividen los componentes en tres partes:

- El componente Swing que implementa los aspectos de vista y controlador.
- El objeto del Look & Feel instalado en el componente Swing que define el aspecto exacto del componente.

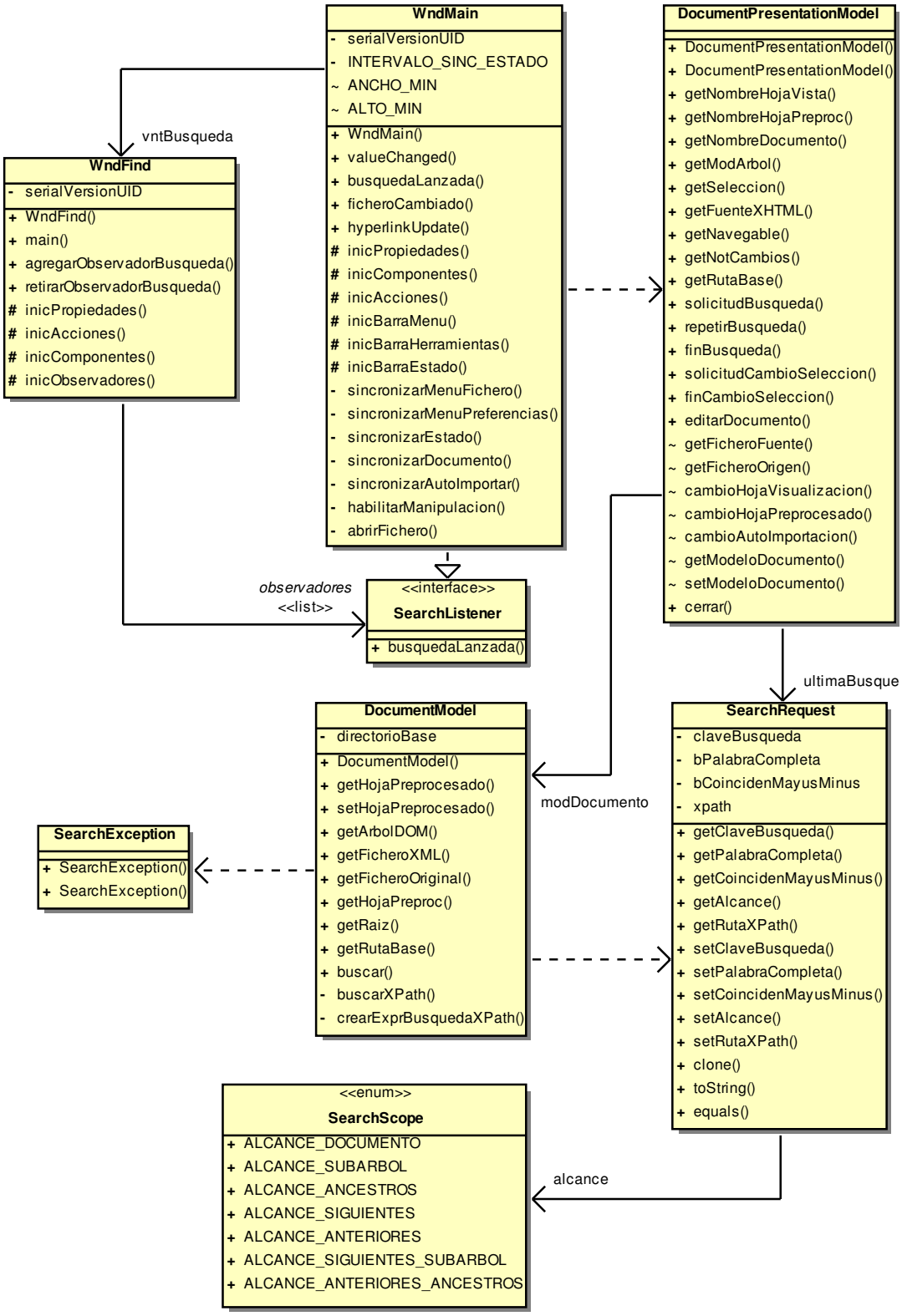


Figura 4.28: Diagrama de clases de diseño de búsquedas

- El modelo del componente, es decir, la información que contiene, y que modifica su visualización o comportamiento. En Swing, el modelo puede contener también detalles de la interfaz gráfica: botón activado o desactivado, por ejemplo.

Así, se han definido dos *TreeModel*:

DOMTreeModel Modelo sobre un árbol DOM XML estándar, que hace accesibles los nodos de tipo elemento.

DataDrivenTreeModel Especialización del anterior, este modelo incorpora la capacidad de realizar falsas podas, ocultando nodos de la visualización sin eliminarlos del árbol DOM.

La decisión de ocultar un nodo o sus hijos se halla en el propio documento XML. De esta forma, el usuario puede controlar este aspecto a través de las hojas XSLT, dirigiéndolo todo por datos.

En los campos `ATTRIBUTE_LEAF` y `ATTRIBUTE_HIDDEN` de *DataDrivenTreeModel* se hallan los atributos que deben tomar el valor del campo `ENABLED_VALUE` para activar la ocultación de los hijos o del propio nodo, respectivamente.

Por las mismas razones que *DataDrivenTreeModel*, se implementó lógica de dibujado dirigida por datos de los nodos del *JTree* mediante la clase *DataDrivenTreeCellRenderer*. Este dibujador puede modificar el icono y la etiqueta de un nodo a través de sus propios atributos `NODELABEL_ATTRIBUTE` y `NODEICON_ATTRIBUTE`.

Se extendieron las capacidades de *JTree* en *DOMTree*, implementando operaciones para encapsular la navegación por el árbol y abstraer al resto de la interfaz de los detalles, como `selectParent`.

En la figura 4.29 (página 127) se ilustran en un diagrama UML las relaciones entre las clases relacionadas con la visualización de árboles XML, incluyendo sus casos de prueba.

Múltiples métodos de acceso a una acción por el usuario

Otro patrón de diseño empleado comúnmente en las interfaces gráficas es el patrón Orden, y esta interfaz no es una excepción.

En este patrón, se define una interfaz para una acción genérica. El lanzador de las acciones recibe objetos que cumplen tal interfaz. Al generarse el evento de interés, reacciona ejecutando cada una de las acciones a través de dicha interfaz.

Este patrón permite aislar al elemento que lanza la acción del que realiza la acción. Así, podemos asociar cualquier acción a un control Swing, sin que deba saber exactamente qué hace esa acción. Sólo conoce la interfaz que debe cumplir la acción, limitándose a invocar al método correspondiente.

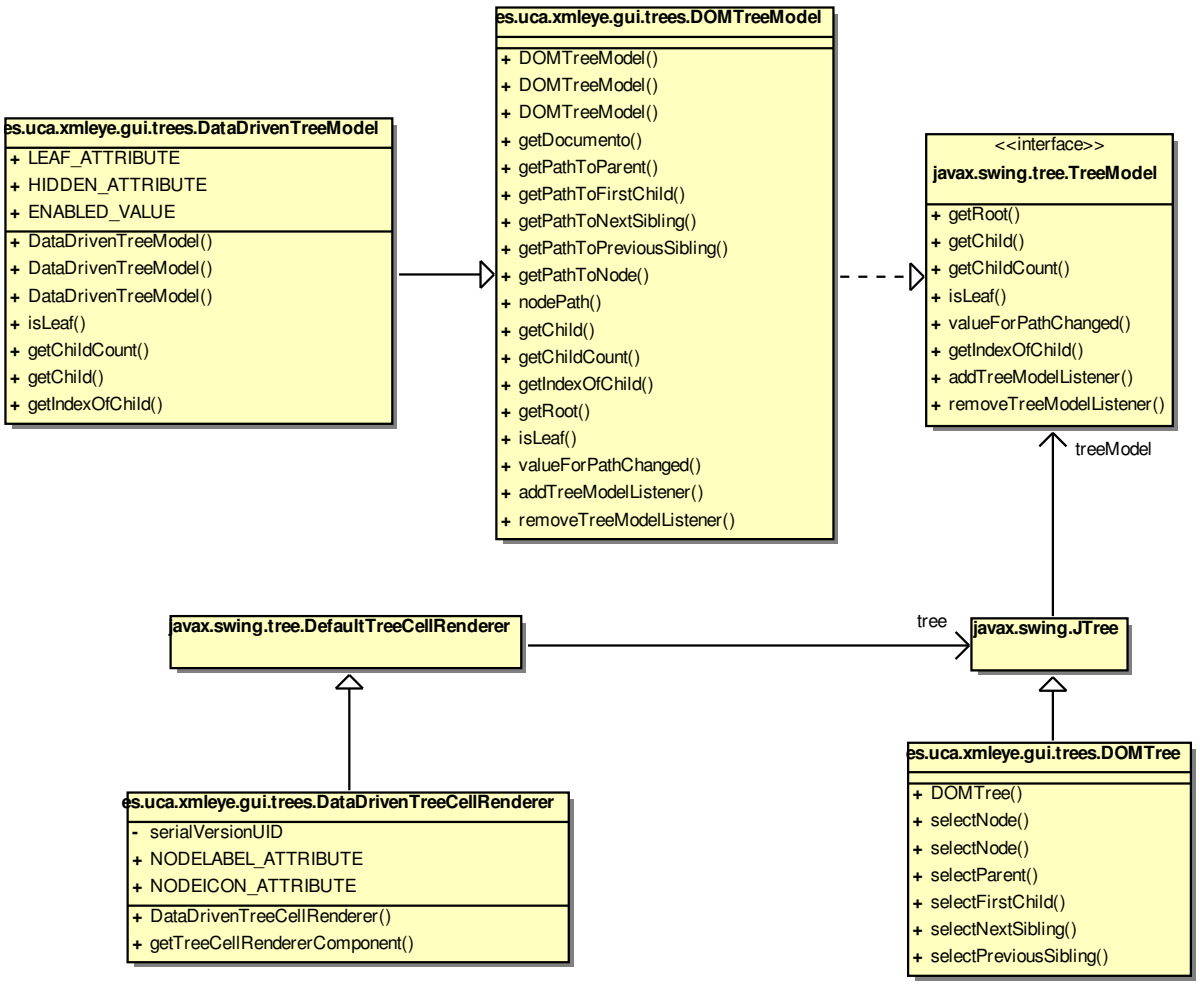


Figura 4.29: Diagrama de clases de manejo de árboles

Además de reducir el acoplamiento, también mejora la estructura del programa evitando la duplicación de código al codificar la misma lógica en varios componentes.

Es el caso usual de las acciones disponibles mediante la barra de herramientas, un acceso de teclado y un elemento del menú, por ejemplo: no importa exactamente quién lanza el evento, sólo qué acción se lanza.

En Swing, la interfaz a cumplir es *Action*, que permite no sólo unificar la orden a ejecutar, sino también parte de su visualización. Podemos asignar un acelerador de teclado, una descripción corta y un mnemónico a una acción, y al crear un botón o elemento de menú a partir de ella, éstos se inicializarán con los valores correctos.

4.6. Implementación

4.6.1. Perl

Se usaron la obra [48] y la web [1] como referencias para los aspectos de división en módulos y la implementación en Perl del paradigma orientado a objetos.

Varios detalles de implementación de ciertos patrones y otras prácticas recomendadas para Perl se obtuvieron de [55].

Perl y el paradigma OO

No hay que confundir el hecho de emplear el paradigma orientado a objetos con el uso de un lenguaje orientado a objetos.

Perl proporciona mecanismos para implementar la mayoría de la funcionalidad necesaria para este paradigma, pero no proporciona un soporte directo a través del lenguaje de muchas construcciones, como una sintaxis específica para declaración de clases con palabras reservadas para control de acceso, por ejemplo.

Otras veces ha de usarse una sintaxis algo más compleja de lo usual, como en el caso de las variables de instancia.

Para implementar los métodos, por ejemplo, Perl permite “bendecir” una referencia cualquiera con el nombre de un módulo. A partir de ese momento, se buscará la subrutina `metodo` dentro del módulo con el que se bendijo `$variable` al usar esta sintaxis:

```
$variable->metodo(@args);
```

Un módulo puede declarar a otros como base, que serán consultados mediante un recorrido primero en profundidad sobre el árbol de generalizaciones durante la búsqueda de un método. Este mecanismo es el que implementa la herencia y los métodos virtuales.

Internacionalización

La internacionalización de los mensajes se consigue en `ACL2::Procesador` a través del módulo `Locale::Maketext`, que permite usar una jerarquía de clases donde la clase base define la lógica común y los idiomas por omisión y sus hijas contienen un diccionario que realiza la localización para cada idioma (posiblemente especificando el país).

El uso es muy sencillo:

Listado 4.1: Ejemplo de uso de Locale::Maketext

```
# Obtenemos el manejador a la localización que más se ajuste,
# subclase de ACL2::Localizacion, que es a su vez subclase
# de Locale::Maketext.
my $lh = ACL2::Localizacion->get_handle();

# Buscamos en el diccionario y sustituimos
# la cadena original: buscando 'Hola [_1]' se hallaría
# 'Hello [_1]' y luego se sustituiría, dando 'Hello Pablo'.
print $lh->maketext("Hola [_1]",'Pablo');
```

Dicho módulo es técnicamente superior al conocido `Gettext`, por la flexibilidad que su enfoque orientado a objetos le aporta: se pueden definir funciones para manejar las formas plurales según el lenguaje, por ejemplo, simplemente redefiniendo un método en la clase correspondiente.

También numera los parámetros de las cadenas, permitiendo mayor flexibilidad que la usual cadena con parámetros para `printf`, que da resultados pobres en casos en los que haya que cambiar el orden de las palabras.

Distribución mediante PAR

Distribuir `ACL2::Procesador` era bastante complejo e incómodo en su anterior versión. Se deseaba facilitar este aspecto en la mayor medida de lo posible, permitiendo instalar de forma automática las dependencias o empaquetarlas en un formato cómodo de utilizar.

Este problema no es único a `ACL2::Procesador`: empezando por `YAXML::Reverse`, prácticamente todo módulo sufre de él. Existen muchas alternativas, como `perlcc` (retirado a partir de Perl 5.10), Perl2Exe o Perl2App, pero entre todas ellas destacan `PAR` y `PAR::Packer`.

De forma muy similar a los ficheros `.jar` de Java, `PAR::Packer` y su herramienta `pp` nos permiten crear ficheros `.par` con todo el código fuente de los conversores y sus dependencias que no forman parte de los módulos estándar de Perl.

Instalar estos conversores y sus hojas se convierte en algo tan sencillo como descargar el `.par` correspondiente a nuestra plataforma y descomprimirlo sobre el directorio raíz de XMLEye. Además, si nuestra aplicación es Perl puro, el `.par` generado puede ser usado en cualquier arquitectura y sistema operativo.

Si no queremos obligar a nuestros usuarios a tener que instalar un entorno Perl y los módulos `PAR` y `PAR::Packer` (cosa particularmente molesta en Windows), también podemos crear ejecutables monolíticos, específicos del sistema operativo y arquitectura bajo el que se creen, que incluyen hasta el propio intérprete de Perl con sus módulos estándar.

Estas imágenes no son tan grandes como se esperaría: los `.par` para `YAXML::Reverse` y `ACL2::Procesador` no llegan a los 100KiB, y los ejecutables, una vez comprimidos, no alcanzan los 2MiB. Además, estas distribuciones comprimidas incluyen todos los descriptores de formato y hojas de usuario necesarias.

4.6.2. Java

Un texto útil ha sido la referencia del lenguaje [2], escrita por los creadores de Java. La web [51] detalla además algunas prácticas recomendadas, funcionalidades no documentadas y soluciones temporales a fallos conocidos de Swing.

Un ejemplo es la disponibilidad no documentada de fuentes con anti-aliasing bajo la J2SE 5.0, o la necesidad de forzar manualmente un tamaño mínimo sobre los hijos de *JFrame*, un fallo documentado en la propia base de datos de Sun y corregido en la próxima J2SE 6.0 “Mustang”.

El uso del IDE Eclipse 3.3.2 ha sido fundamental gracias a su soporte para la realización sencilla de refactorizaciones a gran escala, pudiendo renombrar métodos y clases completas sin introducir errores en el código, entre otras cosas.

Controles HTML

Los controles para navegación HTML incluidos en Swing dejan bastante que desear a la hora de usar el teclado para navegar o seleccionar enlaces.

Para cubrir estas carencias, se especializó la clase *JEditorPane* en *Browser*.

Otros componentes útiles incluyen un observador de enlaces capaz de lanzar uno de los navegadores web del usuario (el navegador por defecto en Windows y Mac OS X) al pulsar enlaces con URL válidas.

Internacionalización

En el lado de Java, la internacionalización se obtiene a partir de los *ResourceBundle*s, de los que hay dos tipos:

- *PropertyResourceBundle*s, que acceden a ficheros `.properties` de fácil mantenimiento, al sólo contener cadenas de texto.
- Especializaciones de *ListResourceBundle*, que pueden contener cualquier tipo de objeto.

La capa de aplicación sólo maneja mensajes de texto y usa el primer tipo, y la capa de presentación usa el segundo tipo, al necesitar además iconos y atajos de teclado.

La clase *ResourceBundle* de la biblioteca estándar de Java es la ocupada de hallar e instanciar el *PropertyResourceBundle* o *ListResourceBundle* que se ajuste mejor a la localización del usuario.

Componentes especializados Para facilitar la internacionalización, se han definido una serie de clases genéricas que se inicializan a partir de un *ResourceBundle*.

En particular, se ha definido una especialización de *AbstractAction* llamada *ResourceBackedAction*, que extrae sus valores de un *ResourceBundle* que sigue un esquema específico de nombrado de claves.

La clase *AbstractAction* es la clase base de todas nuestras *Orden*es (ver §4.5.5), que proporciona implementaciones por defecto para la mayoría de los métodos de la interfaz *Action*, a seguir por toda *Orden* de Swing.

Todas las Órdenes usadas son clases internas de *WndMain*: las más complejas tienen entidad propia, y las demás son anónimas. Puede verse un diagrama UML de las primeras en la figura 4.30 de la página 132.

También hay menús (*ResourceBackedMenu*) y botones (*ResourceBackedButton*) que extraen sus valores de los *ResourceBundle*.

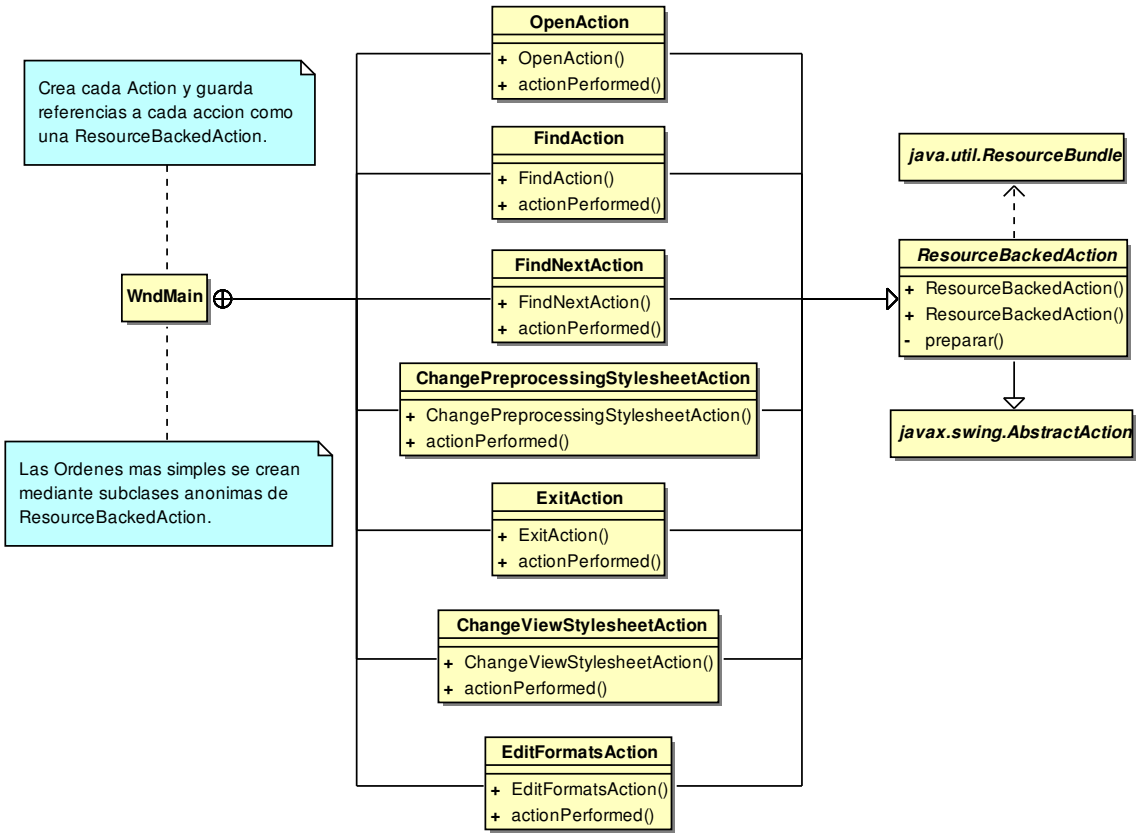


Figura 4.30: Diagrama de clases de Órdenes

4.6.3. Hojas XSLT

Las propias hojas XSLT deben seguir una serie de convenciones en su código y forma de operar. Estas convenciones se hallan documentadas en el apartado del manual de usuario dedicado a explicar cómo crear nuevas hojas XSLT.

4.7. Pruebas y validación

Se dará una breve introducción a los conceptos relacionados con las pruebas de software en XP y a continuación se describirá el plan de pruebas, junto con la especificación de los casos de prueba y su procedimiento de ejecución.

4.7.1. Pruebas en XP

La metodología XP realiza cuatro tipos de pruebas:

Pruebas de unidad Las pruebas de unidad comprueban de forma automática la funcionalidad de un conjunto reducido y cohesivo de clases. Son escritas por los propios programadores, siguiendo la práctica de XP de la programación basada en pruebas.

Pruebas de aceptación Las pruebas de aceptación son escritas por el propio cliente con asistencia de los miembros especializados en pruebas del equipo de desarrollo.

Más que probar el funcionamiento de un par de clases o de un método determinado, lo que se intenta probar es la implementación satisfactoria de una historia de usuario.

Así, las pruebas de aceptación suelen describirse mediante una serie de pasos en los cuales se utiliza el sistema completo para llevar a cabo una tarea. Cada paso tiene un resultado esperado asociado.

Pruebas de integración En XP, las pruebas de integración se realizan de forma constante. Aunque en mi caso no son importantes al ser un proyecto individual, XP requiere que todo cambio hecho en una sesión de programación (en parejas, por supuesto) sea integrado inmediatamente en el repositorio central.

Por supuesto, esto implica que dichas modificaciones deben de haber pasado antes por todas las pruebas con éxito, asegurándonos de que no se han introducido nuevos fallos en otras partes del programa inadvertidamente.

Pruebas de implantación Una de las prácticas recomendadas (que no obligatorias) de XP es la implantación incremental. Desde el inicio de un proyecto, el sistema debe de implantarse en un entorno similar al real, posiblemente a menor escala, y comprobarse su correcto funcionamiento.

Estas pruebas, sin embargo, no son especialmente importantes para una aplicación de escritorio como la de este proyecto.

4.7.2. Plan de pruebas

Alcance

Se determinó que, con las limitaciones de tiempo establecidas, no se podían realizar pruebas de todas y cada una de las partes de XMLEye, `YAXML::Reverse` y `ACL2::Procesador`.

Se decidió implementar pruebas automáticas sobre los elementos fundamentales y utilizar pruebas de aceptación manuales sobre lo demás. En particular, se descartaron aquellas clases cuya funcionalidad dependía en su mayoría de bibliotecas externas, como la interfaz gráfica o el controlador de transformaciones XSLT.

En un futuro, se espera reemplazar la mayor parte de las pruebas sobre la interfaz por pruebas automáticas, utilizando los nuevos modelos de presentación. Actualmente ya se ha comenzado esta labor sobre el diálogo de gestión de formatos.

Tiempo y lugar

Además del propio uso frecuente de los conversores y XMLEye, mis directores de proyecto contribuyeron a la ejecución de pruebas de aceptación de manera informal a lo largo de la ejecución del PFC.

Las pruebas se han realizado en todo momento durante el desarrollo del PFC (especialmente las automatizadas), siguiendo la práctica “Flujo” de XP. Al final de cada iteración, se dedicó un tiempo adicional para la realización de pruebas de aceptación manuales.

Naturaleza de las pruebas

La totalidad de las pruebas usadas son pruebas de caja negra, por ser fáciles de mantener a pesar de cambios en la implementación.

4.7.3. Diseño de pruebas

A primera vista, se puede determinar que existen cuatro elementos fundamentales a probar:

- `ACL2::Procesador`: ¿procesa correctamente la salida de ACL2? ¿Se obtiene un documento que corresponda sin pérdidas al original y que siga el formato de la DTD? ¿Se trata de un módulo Perl correctamente estructurado y documentado?
- `YAXML::Reverse`: ¿analiza sin errores los ficheros de entrada? ¿Realiza una conversión sin pérdidas ni modificaciones de la información del fichero fuente YAML? Al igual que en el caso anterior, ¿tiene un nivel mínimo de calidad como módulo Perl?
- El visor XMLEye: ¿se validan las entradas? ¿Se reacciona bien ante acciones no válidas del usuario? ¿Obtiene el usuario los resultados esperados?
- La integración entre los conversores y XMLEye: ¿se realiza con éxito? ¿Se capturan y muestran correctamente los fallos?

En base a esto, cada parte requiere una combinación determinada de varios tipos de pruebas:

ACL2::Procesador El desarrollo de este conversor se halla guiado por los enunciados Lisp y las salidas de ACL2 que ha de procesar. Por lo pronto, resulta natural estructurar las pruebas simplemente alrededor de dichos casos.

Se puede implementar un mecanismo de pruebas de regresión usando estos enunciados: una vez se haya validado una salida determinada como correcta, se puede mantener dicha salida y comparar las salidas posteriores con ella, notificando cualquier cambio de interés de forma automática.

En un futuro, cuando el alcance del procesador se amplíe a demostraciones de mayor envergadura, se añadirán pruebas de unidad sobre las clases de mayor importancia.

YAXML::Reverse De forma similar al caso anterior, el desarrollo y revisión de este conversor se apoya sobre una serie de entradas conocidas: cuando alguna entrada falla, se añade como caso de prueba y se revisa `YAXML::Reverse` hasta que vuelven a superarse todas las pruebas.

A diferencia de `ACL2::Procesador`, no necesitamos registrar las salidas anteriores "buenas" y compararlas con las actuales, sino que podemos usar el ciclo YAML → XML → YAML implementado y establecer directamente la equivalencia semántica de los dos ficheros YAML original y final. Este proceso se halla descrito a mayor nivel de detalle en §4.5.3 (página 109).

XMLEye Se podría dividir XMLEye en tres partes:

1. La interfaz de usuario: Las limitaciones de tiempo, combinadas con la dificultad y la escasez en herramientas automatizadas de prueba de las interfaces gráficas obliga a usar pruebas de aceptación manuales por lo pronto. En un futuro cercano, el nuevo diseño de la capa de presentación basado en modelos de presentación permitirá automatizar estas pruebas, limitándonos a enviar notificaciones de gestos a los modelos de presentación.

   Sin embargo, los componentes relacionados con los árboles XML son cruciales para la aplicación, y deben de incluir sin falta pruebas automatizadas sobre las propiedades que deben cumplir y el comportamiento que deben seguir.
2. Las hojas de usuario: aunque su realización sería técnicamente factible, las pruebas de unidad de estas hojas no podrían mantenerse estables, puesto que el formato de salida no lo es: al ser XHTML, está sujeto a muchos cambios y retoques a lo largo del tiempo. Se puede decir lo mismo acerca del paso de preprocesado.
3. La capa de aplicación: esta capa provee los servicios básicos sobre los que se apoya la capa de presentación. No probaremos aquellas partes que únicamente dependan de la capa de servicios técnicos, como el funcionamiento de las transformaciones XSLT, por ejemplo, pero sí los servicios de nivel superior.

   Por ejemplo, habrá que depurar los repositorios de hojas de estilo y de descriptores de formatos, la validación de los campos de los descriptores de formato, la correspondencia entre rutas XPath y nodos de un árbol DOM XML, o las extensiones XPath, entre otras cosas.

Integración La integración es otro aspecto difícil de probar automáticamente, dado que depende en gran medida del entorno empleado por el usuario y de la configuración que haya establecido.

Sin embargo, se presta bien a pruebas de aceptación manuales sobre una configuración realizada de antemano. Se habrá de probar que la integración reacciona bien a errores de los programas y de su invocación, fallando cuando debe y permitiendo una recuperación rápida.

4.7.4. Especificación de los casos de prueba

ACL2::Procesador

Cada una de las fuentes Lisp y su salida correspondiente a filtrar por `ACL2::Procesador` según las historias de usuario planteadas es utilizada como una prueba de regresión semiautomática.

Describiremos brevemente el contenido de las fuentes Lisp usadas como casos de prueba. Por claridad, usaremos notación infija. Todos estos enunciados se encuentran bajo el subdirectorio `t/testInputs` de `ACL2::Procesador`.

Durante las pruebas se desactiva todo almacenamiento de resultados intermedios y se obliga a actualizar el grafo completo de dependencias, para evitar que en ejecuciones distintas de las pruebas se consigan resultados distintos.

- `triple-rev`

  `triple-rev` fue el objetivo de la primera iteración. En esta demostración se comprueba para una función de inversión de listas Lisp `rev` que

  $$rev(rev(rev(x))) = rev(x)$$

  para toda lista que le pasemos, sea cual sea su estructura.

- `equal-app`

  En esta demostración se comprueban ciertas propiedades acerca de las funciones `app` para concatenación de listas, `dup` para duplicación de cada elemento y `properp` para comprobación de listas propias, terminadas mediante `nil`, como `(a b)` o `(a . (b))` pero no `(a . b)`.

  - Asociatividad: $app(app(a, b), c) = app(a, app(b, c))$
  - Distributividad de la duplicación de cada elemento respecto de la concatenación: $app(dup(a), dup(b)) = dup(app(a, b))$
  - Toda lista a la que se añade *nil* es propia: $properp(app(a, nil))$.

- `fallo-defun`

  Realmente, este guión no trata de probar nada: está diseñado para ver si `ACL2::Procesador` es capar de tratar los fallos en las demostraciones de los `defun`.

  En primer lugar, declaramos una función de cálculo de factorial correcta, que siempre da un resultado correcto y siempre para.

  La siguiente versión erróneamente usa $=$ en vez de zp, y por lo tanto no para nunca para otra cosa que no sea un natural. zp es verdadero para toda cosa que no sea un número mayor que cero: listas, átomos, enteros no positivos (incluyendo el cero, claro), etc. Sin embargo, $=$ sólo es verdadero si lo que se pasa es cero, por lo que ante un número negativo, por ejemplo, nunca llegaría al caso base y se quedaría siempre en el caso recursivo.

  En la tercera versión, no hay caso base: se ha reemplazado por una llamada recursiva a sí mismo. Y en la cuarta y última, no se reduce el tamaño de la entrada, imposibilitando también la parada.

- `hanoi`

  No es uno, sino realmente cuatro casos de prueba, al tratarse de un proyecto de ACL2 con tres ficheros Lisp. Tenemos un caso de prueba por fichero y otro caso de prueba más para la detección del grafo que forman y su correcto uso:

- `hanoi-use.lisp` es el fichero raíz del proyecto. Utiliza la orden `include-book` para acceder a la definición de la función que resuelve el problema de las torres de Hanoi, `hanoi::hanoi`, que se halla en el libro BOOKS/HANOI.
- `books/hanoi.acl2` es el fichero Lisp que sirve para certificar los contenidos del libro HANOI dentro de su mundo lógico inicial. Se podría haber llamado `cert.acl2` si se hubiera deseado emplear para cualquier libro dentro de `books`, siguiendo las convenciones usuales de manejo de libros de ACL2 [40], pero se ha preferido esta forma específica, para cuando se añadan más libros posteriormente.
- `books/hanoi.lisp` es el propio libro con las definiciones necesarias, que asume que se halla dentro del mundo lógico inicial definido por el fichero de certificación anterior. Define las funciones `move`, que genera una instrucción del tipo "mover un disco de la pila A a la pila B" y `hanoi`, que resuelve el problema de las Torres de Hanoi para n discos. También incluye un teorema, `len-append`, que establece que la longitud de la concatenación de dos listas es la suma de sus longitudes.

- `mapnil`

  En este caso, se demuestra la conmutatividad entre la duplicación de los elementos mediante `dup` y la sustitución de todo elemento de una lista Lisp por `nil` mediante `mapnil`.

- `memp`

  Dada una función `memp` que comprueba la pertenencia de un elemento a una lista y otra función `app` que concatena dos listas, se demuestra que

  $$memp(e, app(a, b)) \equiv memp(e, a) \vee memp(e, b),$$

  es decir, que todo elemento de la concatenación de las dos listas pertenecerá a una u a otra.

- `miscDefs`

  Comprobamos que `ACL2::Procesador` es capaz de filtrar una diversidad de funciones, como búsquedas en diccionarios, implementación de aritmética mediante el uso de la longitud de las listas, recolección de elementos únicos, potenciación, etc.

  Algunos ejemplos:

  - $mapnil(x)$ convierte todos los elementos a *nil*.
  - $add(x, y)$ realiza la suma mediante el número de elementos de ambas listas. Es decir, implementamos la suma creando una lista de 5 elementos cuando nos dan una de 2 y otra de 3.
  - $app(a, b)$ Concatenamos dos listas.
  - $mem(e, x)$ Devuelve el primer par de la lista tal que el primer elemento sea e, o nil.

- $lonesomep(e, lst)$ Comprobamos que e es único en la lista.
- $collect - lonesomep(a, b)$ Recoge los elementos de la lista a que son únicos en la lista b.
- $mult(x, y)$ Multiplica a través de longitudes de listas. Si le damos una lista con 2 elementos y otra con 3, nos crea una lista con 6.
- $fact(n, a)$ Calcula el productorio en aritmética sobre longitud de listas de las sublistas de la lista que le pasemos.
- $fact1(n, a)$ Versión recursiva final de `fact`.
- $foundp(x, a)$ Busca una clave dentro de un diccionario Lisp. Si el diccionario es la lista Lisp de la forma *((a 2) (b 3))*, sus claves son a y b.

- `suma`

  Este caso comprueba que se tratan de forma correcta los `defthm` que no requieren inducción alguna. Aquí, ACL2 sólo tiene que aplicar sus reglas predefinidas sobre la aritmética decimal para ver que $2 + 2 = 4$ o que $a + b + c = b + c + a$.

  También se comprueba que se manejan bien los elementos de listas Lisp con la sintaxis `|texto|`.

- `swaptree`

  Se define una función `swaptree` que invierte dos árboles Lisp, y se demuestra que es idempotente.

- `tail-rev`

  Se define una implementación trivial de la inversión de una lista `rev` sabiendo que es correcta. Hecho eso, definimos una implementación recursiva final obtenida por la transformación mediante desplegado-plegado `rev1`. Ahora comprobamos que está bien hecha, es decir, que da los mismos resultados que la versión original.

- `treecopy`

  Es parecida a `swaptree`, pero en este caso lo que se hace es copiar un árbol Lisp, y comprobar que dicha función repite siempre la entrada que se le suministra.

- `triple-rev-misspell`

  Este caso de prueba es una modificación de `triple-rev` con un fallo en el nombre de un parámetro formal introducido intencionadamente, que origina una demostración fallida de la longitud suficiente como para ser de interés. Además, permite comprobar que se tratan bien los errores de `defthm`.

- `tutorialWebACL2`

  Este otro caso de prueba es el proyecto de un solo fichero más completo de todos. En este caso, se desea probar que un algoritmo para la inversión de una lista Lisp que sólo hace uso de las funciones predefinidas *endp* (predicado para detectar el final de una lista), *cons* (creación de un par Lisp), *car* (extracción de la cabecera) y *cdr* (extracción de la cola) es equivalente al algoritmo trivial.

El proceso es algo más complicado que los ejemplos anteriores, y puede verse en [38], donde se usa "El Método", acumulándose teoremas a demostrar en forma de una pila, donde un teorema puede requerir para su demostración verificar otros teoremas o lemas.

Se hacen uso de algunos elementos más avanzados, como `thm`, que lanza una demostración pero no guarda sus resultados en el mundo lógico de ACL2, `local` que evita que las reglas generadas por un evento sean accesibles fuera de un `encapsulate` o libro, y `encapsulate`, que permite realizar demostraciones controlando qué reglas se producen y cuál es la signatura de las funciones definidas.

Las pruebas de regresión comparan los árboles XML de los resultados esperados con los obtenidos, empleando el módulo `XML::SemanticDiff`. Las diferencias detectadas son filtradas por un predicado que determinan si constituyen una regresión o no. Actualmente, algunas de las diferencias ignoradas son:

- Cambios en los tiempos de ejecución.
- Cambios en la versión de ACL2.
- Cambios de mayúsculas y minúsculas y barras en nombres de fichero en Windows.
- Mensajes de compilación durante la certificación de un libro (dependen del compilador y sistema operativo usado).

Las pruebas de regresión no son las únicas realizadas. Integradas dentro del marco de pruebas creado por el módulo `ExtUtils::MakeMaker` se hallan también algunas pruebas de carácter más general:

- El módulo principal debe de poder cargarse con éxito. Esto nos asegura de que no nos hayamos olvidado de instalar o referenciar a algún módulo importante en nuestro código.
- Se debe de haber retirado el texto inútil de las plantillas inicialmente generadas por `h2xs`, la herramienta ocupada de crear el esqueleto básico de todo módulo Perl.
- Todos los ficheros fuente del módulo Perl han de incluir el aviso de la licencia GPL.

YAXML::Reverse

Como otro módulo Perl más, `YAXML::Reverse` emplea el mismo marco de pruebas y las pruebas de carácter general de `ACL2::Procesador`. Este módulo usa también pruebas de regresión, pero son completamente automáticas en este caso, como ya se explicó anteriormente. Ciertas pruebas de regresión son más bien para la versión

refinada de YAXML implementada para cerrar el ciclo YAML → XML → YAML que para `YAXML::Reverse` en sí.

Algunos casos de prueba se corresponden con documentos reales usados por herramientas conocidas, y algunos sólo depuran ciertos aspectos esenciales. Existe cierto solapamiento entre ambos tipos. Los ficheros `.yaml` utilizados como casos de prueba son:

- `djangoJSON`

  Este es un ejemplo de un volcado de una base de datos realizado por el entorno de desarrollo de aplicaciones Web Django (http://www.djangoproject.com/). Tiene la particularidad de usar JSON, un subconjunto de YAML 1.1, motivando el cambio del módulo `YAML::Syck` a `YAML::XS`.

- `djangoJSON_blockIndicatorTest`

  Otro volcado de Django en el cual se hace patente la necesidad en YAXML de tener cuidado con el espaciado en blanco y no cambiarlo inadvertidamente. En particular, hay que tener cuidado con el último salto de línea final, evitando introducirlo cuando originalmente no estaba, utilizando el indicador "|-" en el estilo de bloque para escalares.

- `firefoxBookmarks`

  Una copia de seguridad de los marcadores de Firefox 3 realizada también en JSON, que en este caso tenía la complicación de emplear caracteres de UTF-8, en particular ideogramas japoneses. Se identificaron y resolvieron diversas cuestiones relacionadas gracias a este ejemplo.

- `fiveMinutes1`

  Es uno de los casos más básicos: un flujo de 4 documentos que emplean secuencias de YAML. Es compatible con YAML 1.0 en adelante, al igual que todos los casos de prueba que vienen a continuación.

- `fiveMinutes2`

  Este ejemplo muestra ejemplos de vectores asociativos de YAML 1.0, y fue el primer caso de prueba que mostraba la necesidad de rodear las limitaciones de XML respecto a los identificadores válidos para una etiqueta utilizando los elementos `_key` y `_value`.

- `fiveMinutes3`

  Desarrollando sobre el ejemplo anterior, este caso de prueba también usa claves de vectores asociativos no directamente representables en XML, pero además una de ellas requiere un especial cuidado a devolverla a YAML, o produciría un error de sintaxis, al contener la secuencia ": ".

- `fiveMinutes4`

Este caso de prueba utiliza los estilos literal y de líneas reunidas para los escalares. En el estilo literal el espaciado no es modificado excepto por el indentado de cada línea. En el estilo de líneas reunidas, todas las palabras separadas por únicamente un carácter de espaciado (como un espacio o un salto de línea) son reunidas en una sola línea.

- `fiveMinutes5`

  El último ejemplo de la serie [56] ilustra cómo pueden escribirse también las colecciones con distintos estilos, ya sea de bloque, con un elemento por línea, o de flujo, que permite definir más de un elemento en cada línea.

- `invoice`

  Este caso de prueba es uno de los ejemplos incluidos en la especificación YAML [7], y precisamente el que ilustra el uso de anclas y alias para evitar tener que repetir el contenido común a dos nodos.

- `mappingAnchoredSequence`

  Aquí probamos a transformar un documento en que el documento anclado no es un mapa, sino una secuencia.

- `podExample`

  Este ejemplo tan sencillo es el usado en la documentación POD de `YAXML::Reverse`, para poder garantizar que funcione.

- `yaxmlReverseMETA`

  Se trata del fichero `META.yaml` de `YAXML::Reverse`. Estos ficheros son utilizado por diversas herramientas automáticas, como las que utiliza el CPAN, para obtener información de autoría y dependencias.

Visor e integración

Pruebas de unidad Las pruebas de unidad definidas para cada clase comprueban una serie de propiedades acerca de su comportamiento. En lugar de dar una especificación exacta de cada prueba (que sólo sería repetir el código), listaremos las propiedades que verifican.

Estas pruebas, basadas en el marco JUnit, se ejecutan de forma automática y son un objetivo más dentro del fichero de tareas Ant.

Toda clase de pruebas tiene el nombre formado por el nombre de la clase cuyo funcionamiento comprueba y a continuación el sufijo “Test”.

CommandBuilderTest Se comprueba que se realiza la sustitución de las claves y la división en argumentos correctamente, independientemente de que los argumentos contengan espacios, mientras tengan comillas alrededor de la orden original.

DataDrivenTreeModelTest Todo nodo del documento XML oculto mediante el atributo correspondiente debe ser inaccesible a través de un recorrido exhaustivo.

Además, el modelo ha de informar de que todo nodo con el atributo requerido para la falsa poda activado no tiene hijos, y éstos deben ser completamente inaccesibles desde dicho modelo.

DOMTreeModelTest El modelo ofrecido por dicha clase debe de ser el mismo que ofrece el árbol DOM original del documento, limitándose a los nodos de tipo elemento.

DOMTreeTest Debe de poderse realizar una navegación completa a través del árbol, ignorando toda petición no válida, como intentar ir al padre de la raíz.

ExtensionFileFilterTest Se prueba que se añade automáticamente la extensión al nombre de fichero si no está presente, que se conserva si ya está, y que se conserva el nombre de fichero si no se usa un filtro *FiltroExtensión*.

FormatDescriptorRepositoryTest Utilizando un repositorio a dos niveles con el descriptor de formato `xml.format` incluido con XMLEye, se comprueba que se inicializa correctamente. También se prueba a cambiar un descriptor, guardarlo y volver a cargar, para ver si los cambios se han realizado.

A continuación se hacen pruebas sobre la capacidad de localizar las cadenas de un descriptor de formato, de que los cambios realizados a un descriptor vayan al repositorio de mayor prioridad, y que se puedan restaurar las opciones originales de un formato.

Finalmente, se comprueba que los mutadores de todo descriptor de formato lancen ciertas excepciones al pasarles algún valor no válido.

SerializedPreferencesManagerTest Se comprueba que se puede crear un fichero de preferencias nuevo, y que inicialmente está vacío. Debe además poder guardar de forma persistente escalares y listas.

StylesheetRepositoryTest Se verifica que las hojas incluidas en XMLEye pueden localizarse sin problemas dentro del repositorio, que se hallan debidamente configuradas y que su código XSLT compila sin problemas.

VectorNodeListTest Se prueba que el método `invertir` realmente cumple su tarea, y que la inversión es idempotente.

WndFormatsPresentationModelTest Se comprueba que cuando el usuario seleccione un elemento, cambie el nombre y confirme los cambios estos se realicen de forma correcta y persistente.

XHTMLSearchKeyHighlighterTest Al destacar una clave dentro de un documento XHTML los elementos XHTML deben de conservarse intactos. No debe modificarse otra cosa que el contenido del documento en sí, es decir, el fragmento situado dentro del elemento `body`.

Deben además de manejarse bien los casos en que la clave a marcar está justo antes o después de un elemento XHTML. Otros casos importantes a tratar incluyen el manejo correcto de marcado de sólo palabras completas o sin tener en cuenta mayúsculas y minúsculas.

XPathExtensionsTest Se comprueba que se obtienen coincidencias de la forma esperada para varias combinaciones posibles de tipos de clave y cadena de búsqueda.

Así, la clave de búsqueda podría componerse sólo de caracteres alfanuméricos. También podría contener espacios o caracteres no alfanuméricos. Se podría dar el caso de que el usuario introdujera algún carácter que debería escaparse para no ser interpretado como un metacarácter en una expresión regular.

El texto en el que buscar podría contener espacios iniciales o finales, o saltos de línea.

También podría darse el caso en que la clave y el texto fueran iguales, o que alguno de los dos estuviera vacío.

XPathPathManagerTest Se comprueba que la ruta XPath generada de cualquier nodo del árbol DOM de un determinado documento XML es correcta, y que la obtención de un nodo a partir de su ruta XPath se realiza también con éxito.

Pruebas de aceptación Pasaremos a establecer una prueba de aceptación para cada una de las historias de usuario acumuladas desde las primeras versiones de XMLEye hasta ahora. Dichas pruebas se ejecutan de forma manual.

- "Visualizar un documento XML arbitrario en forma de árbol, permitiendo expandir y contraer los nodos y buscar información."

  Cuadro 4.1, página 145.

- "El visor no debería saber absolutamente nada de ACL2. Nada en su interfaz gráfica ni en su implementación nos debe recordar que podemos trabajar con ACL2."

  Cuadro 4.2, página 146. Se asume en esta historia que XMLEye ha sido instalado desde cero, sin ningún otro añadido.

- "Añadir soporte para visualizar varios ficheros a la vez y enlazarlos entre sí."

  Cuadro 4.3, página 147. Se asume que se ha instalado ACL2, `ACL2::Procesador`, sus hojas de usuario y su descriptor de formato de la manera descrita en el manual de usuario.

- "Proporcionar una lista de documentos recientes."

  Cuadro 4.4, página 148.

- "Integrar el visor y el editor preferido del usuario."

  Cuadro 4.5, página 149.

Paso seguido	Resultado esperado
1. El usuario solicita la apertura de un fichero.	Se muestra el diálogo de elección de ficheros.
2. El usuario elige un fichero XML.	Tras la compilación de las hojas y el preprocesado, se muestra el primer nodo del documento.
3. El usuario cambia la hoja de preprocesado a `xml`.	El árbol del documento muestra el árbol XML original.
4. El usuario cambia la hoja de visualización a `xml`.	La visualización muestra todos los atributos, ancestros e hijos.
5. El usuario cambia la hoja de visualización a `xmlSource`.	La visualización muestra el código XML completo de cada nodo seleccionado tras su preprocesado.
6. El usuario navega por el árbol, usando cada una de las opciones disponibles de navegación [...]	La visualización se actualiza en cada cambio de nodo, mostrando la información correcta.
7. El usuario solicita expandir el árbol.	El árbol del documento se expande en su totalidad.
8. El usuario solicita contraer el árbol.	El árbol del documento se contrae, y se muestra solamente el primer nodo del documento.
9. El usuario realiza una búsqueda de una clave existente en el documento.	Se muestra el primer resultado, con el marcado de la clave de búsqueda en la visualización.
10. El usuario pulsa de nuevo en el botón de Buscar del diálogo o utiliza el elemento del menú Navegar “Buscar Siguiente”.	Se muestran el resto de los resultados.
11. El usuario lanza de nuevo la búsqueda al llegar al último resultado.	Se muestra de nuevo el primer resultado.
12. El usuario prueba con el resto de las opciones de filtrado en secuencia y compara con los resultados anteriores [...]	Se comprueba el filtrado del conjunto de resultados obtenido anteriormente.

Cuadro 4.1: Prueba de aceptación para “Visualizar XML genérico”

Paso seguido	Resultado esperado
1. El usuario abre XMLEye.	Se muestra la ventana principal.
2. El usuario abre un documento XML cualquiera.	Se muestra su primer nodo.
3. El usuario comprueba la lista de hojas de preprocesado disponibles.	Sólo se dispone de la hoja `xml`.
4. El usuario comprueba la lista de hojas de visualización disponibles.	Sólo se dispone de las hojas `xml` y `xmlSource`.
5. El usuario intenta abrir otro documento, y prueba a desplegar el filtro de tipos de documento.	Sólo se dispone de las opciones “Todos los ficheros” o “Fichero XML normal”.
6. El usuario cancela su acción y abre el diálogo de gestión de formatos.	Se muestra el diálogo con un único formato: “Fichero XML normal”.

Cuadro 4.2: Prueba de aceptación para “Hacer a XMLEye independiente de ACL2”

- “Vigilar directamente el fichero fuente en caso de cambios, en vez de solamente cuando se cierra el editor. Así, si uno abre el editor, hace un par de cambios y guarda (sin cerrar el editor), también se actualizará. En caso de que se desactive temporalmente la reimportación automática, también debería funcionar correctamente si en ese intervalo se hicieron cambios.”

  Cuadro 4.6, página 150.
- “Guardar las preferencias automáticamente al cerrar el programa.”

  Cuadro 4.7, página 150.
- “Visualizar un fichero Lisp integrando XMLEye con `ACL2::Procesador`.”

  Cuadro 4.8, página 151. Se asume que se ha instalado ACL2, `ACL2::Procesador`, sus hojas de usuario y su descriptor de formato de la manera descrita en el manual de usuario.
- “Implementar la hoja `summaries`, que pode el árbol y deje solamente los resúmenes, y la hoja `reverse`, que invierta el orden de los eventos, para mostrar primero las conclusiones y luego sus antecedentes.”

  Cuadro 4.9, página 151. Se asume que se ha instalado `ACL2::Procesador`, su descriptor de formato y sus hojas de usuario tal y como se describe en el manual de usuario.
- “Añadir información acerca del uso de una meta, como sus dependencias inversas.”

Paso seguido	Resultado esperado
1. El usuario abre un documento XML cualquiera.	Se muestra su primer nodo.
2. El usuario cambia las hojas de visualización y preprocesado a las hojas básicas `xml`.	Se muestra el documento XML tal y como está, listando en cada nodo sus atributos y dando las cabeceras y pies apropiados.
3. El usuario abre el caso de prueba `hanoi-use.lisp` de `ACL2::Procesador`.	Se muestra su primer nodo en una pestaña aparte, con las mismas hojas que en la primera pestaña.
4. El usuario cambia las hojas de visualización y preprocesado a las hojas `ppACL2`.	Se muestran las órdenes `include-book` y `defthm` que conforman al fichero.
5. El usuario selecciona el sumario del `defthm`.	Aparecen enlaces a `HANOI::HANOI` entre las reglas usadas.
6. El usuario activa el enlace a `HANOI::HANOI`.	Se abre `hanoi.acl2` en una pestaña, con el nodo en cuestión seleccionado.
7. El usuario vuelve a `hanoi-use.lisp`, y esta vez activa `HANOI::MOVE`.	Se cambia de nuevo a la pestaña de `hanoi.acl2`, moviéndose al nodo seleccionado.
8. El usuario cambia a la pestaña del documento XML.	El documento mantiene la selección anterior y utiliza las hojas de preprocesado y visualización `xml`.

Cuadro 4.3: Prueba de aceptación para “Visualizar y enlazar varios documentos entre sí.”

Paso seguido	Resultado esperado
1. El usuario abre un documento XML no presente aún en el historial.	Se muestra dicho documento y aparece su entrada en el historial.
2. El usuario cierra XMLEye.	Se guarda el historial de documentos recientes.
3. El usuario vuelve a ejecutar el programa y examina el historial.	Se sigue mostrando entre sus entradas el anterior fichero abierto.
4. El usuario selecciona dicha entrada.	Se abre de nuevo el documento.
5. El usuario cierra XMLEye, cambia al documento de localización y vuelve a ejecutar XMLEye.	Se sigue listando al fichero en su historial.
6. El usuario selecciona la entrada anterior.	Se informa que el fichero no existe y elimina la entrada del historial.

Cuadro 4.4: Prueba de aceptación para “Historial de documentos recientes”

Cuadro 4.10, página 152. Se asume que `ACL2::Procesador` se halla debidamente instalado.

- “Abrir documentos YAML en XMLEye sin que suponga una pérdida o alteración de la información disponible.”

  Cuadro 4.11, página 152. Se asume que se ha instalado `YAXML::Reverse` con su descriptor de formato y hojas de usuario debidamente.

Paso seguido	Resultado esperado
1. El usuario abre un documento XML genérico.	El sistema muestra dicho documento según las hojas actuales.
2. El usuario solicita abrir el diálogo de gestión de formatos admitidos.	Se abre el diálogo de gestión de formatos admitidos, incluyendo al menos al elemento "Fichero XML normal". No se puede acceder al formulario principal.
3. El usuario comprueba el editor establecido y cierra el diálogo.	Se cierra el diálogo, volviendo a poder acceder al formulario principal.
4. El usuario solicita la edición del documento.	Se lanza el editor que antes estaba especificado sobre el fichero actual. La interfaz gráfica puede seguir usándose en paralelo sin problemas.
5. El usuario cierra el editor, utiliza el diálogo de formatos admitidos para cambiar la orden a otro editor existente en el sistema y confirma sus cambios.	Los cambios son guardados.
6. El usuario solicita de nuevo la edición del documento.	Se lanza el editor especificado anteriormente, y la interfaz gráfica se puede seguir usando sin problemas.

Cuadro 4.5: Prueba de aceptación para "Integrar editor"

Paso seguido	Resultado esperado
1. El usuario abre un documento XML cualquiera.	El sistema muestra dicho documento según las hojas actuales.
2. El usuario solicita la edición del documento.	Se abre el editor registrado por el usuario para el formato empleado.
3. El usuario activa la actualización automática en las preferencias.	La entrada del menú pasa a estar marcada como activa.
4. El usuario cambia el contenido del documento, manteniendo su corrección sintáctica, y guarda los cambios.	El documento se vuelve a abrir, y los cambios se reflejan en él.
5. El usuario desactiva la actualización automática en las preferencias.	La entrada del menú pasa a estar marcada como inactiva.
6. El usuario deshace el cambio anterior, y guarda los cambios.	El documento no se reabre.
7. El usuario activa de nuevo la actualización automática en las preferencias.	La entrada del menú vuelve a aparecer como activa y el documento se reabre, reflejando los cambios realizados.
8. El usuario hace cambios que violan la corrección sintáctica de la entrada.	El proceso de análisis sintáctico falla, notificándose al usuario del hecho, y por lo demás operando de forma normal. La copia antigua del documento sigue abierta y funciona sin problemas.

Cuadro 4.6: Prueba de aceptación para “Actualizar automáticamente”

Paso seguido	Resultado esperado
1. El usuario, tras abrir un documento XML, cambia la hoja de preprocesado a un valor distinto.	Se actualiza el documento según la nueva hoja.
2. El usuario cierra XMLEye.	Se guarda las preferencias.
3. El usuario ejecuta de nuevo XMLEye, y abre un documento XML.	Se conserva la selección de hoja de preprocesado de la ejecución anterior.

Cuadro 4.7: Prueba de aceptación para “Guardar preferencias”

Paso seguido	Resultado esperado
1. El usuario abre el caso de prueba `triple-rev-misspell.lisp` de `ACL2::-Procesador`.	El sistema muestra el primer nodo del documento.
2. El usuario selecciona la hoja de preprocesado `ppACL2`.	Se actualiza el documento y muestra el árbol debidamente decorado con los nombres de los eventos y los iconos de éxito y fracaso, seleccionando y mostrando el primer nodo de la visualización. En particular, debería de haber un único elemento marcado como fracaso: `REV-REV`, hijo directo del nodo raíz.
3. El usuario selecciona la hoja de visualización `ppACL2`.	Se actualiza la visualización del nodo actual, pasando a mostrar solamente la información relacionada con ACL2: los nombres de cada evento y el código Lisp asociado.
4. El usuario navega por el árbol del documento [...]	Se visualiza la información correspondiente al nodo de la forma esperada. En particular, debería de informar del punto exacto del fracaso de `REV-REV` a través de una cadena de iconos de fracaso.

Cuadro 4.8: Prueba de aceptación para "Integrar XMLEye y `ACL2::Procesador`"

Paso seguido	Resultado esperado
1. El usuario abre un documento Lisp.	El sistema muestra el primer nodo del documento.
2. El usuario selecciona la hoja de visualización `ppACL2`.	El sistema actualiza el nodo seleccionado.
3. El usuario selecciona la hoja de preprocesado `summaries`.	El sistema actualiza el documento, que ahora sólo muestra las órdenes y los sumarios.
4. El usuario selecciona la hoja de preprocesado `reverse`.	El sistema actualiza el documento, en el que los eventos se hallan en orden inverso al que siguen en la fuente Lisp.

Cuadro 4.9: Prueba de aceptación para "Hojas `summary` y `reverse`"

Paso seguido	Resultado esperado
1. El usuario abre un documento Lisp.	El sistema muestra el primer nodo de dicho documento según las hojas actuales.
2. El usuario selecciona la hoja de preprocesado `ppACL2` y la hoja de visualización `ppACL2`.	Se actualiza el documento y la visualización, ahora estructurados ambos en órdenes y demostraciones de ACL2.
3. El usuario selecciona el primer evento usado por otros eventos.	Se muestra dicho nodo, con una lista de enlaces a los eventos que lo mencionan.

Cuadro 4.10: Prueba de aceptación para “Ver usos de una meta”

Paso seguido	Resultado esperado
1. El usuario abre el caso de prueba `invoice.yaml` de `YAXML::Reverse`.	Se muestra su primer nodo.
2. El usuario cambia a las hojas de preprocesado y visualización `yaxml`.	Se muestra un árbol con un único documento anónimo, que contiene los mismos elementos que el documento YAML original.
3. El usuario selecciona el elemento `bill-to` del documento.	Se muestran los atributos del elemento, apareciendo un enlace en el atributo `yaml:alias`.
4. El usuario activa el enlace.	Se selecciona el elemento `ship-to` del documento al que hace referencia el ancla.

Cuadro 4.11: Prueba de aceptación para “Integrar XMLEye y `YAXML::Reverse`”

5 Resumen

Puede verse como, en este trabajo, ocupado más de la conversión de una serie de formatos que de la usual manipulación de objetos del dominio presente en las aplicaciones de gestión, y basado en otro Proyecto anterior, el análisis ha tomado una porción relativamente pequeña de la duración del proyecto.

A esto se le añade que los dos "clientes" eran expertos del dominio de ACL2, y que se tenía una comunicación constante con ellos, como sugiere la metodología XP. Para YAML no hubo clientes, pero esta carencia ha sido cubierta a través de su especificación formal y el amplio número de herramientas que implementan y utilizan este metalenguaje.

En cuanto al diseño, éste se ha modelado de manera incremental, intentando evitar el diseño excesivo fuera de las necesidades estrictas del proyecto. Todos los componentes han sufrido importantes cambios en su diseño durante este proyecto: `ACL2::Procesador` y `YAXML::Reverse` han pasado a ser módulos Perl con la estructura estándar recomendada por el CPAN, y XMLEye se ha hecho completamente genérico y multidocumento.

5.1. Pruebas continuas

Un factor fundamental para el éxito de este proyecto fue la realización y planificación continua de pruebas durante toda su duración.

Así, se realizaban pruebas de aceptación constantemente, junto con las pruebas automáticas de regresión y de unidad. La principal ventaja de automatizar las pruebas es que aseguran una cierta funcionalidad continuamente sin afectar a la velocidad del desarrollo del proyecto.

Durante el desarrollo del proyecto, se presentaron algunos casos en los que las propias pruebas de unidad no habían capturado algunos fallos. Tras el refinamiento de las pruebas para detectar el fallo y la corrección de este, no solamente se consiguió arreglar el fallo, sino también garantizar que no volvería a aparecer en el proyecto.

5.2. Diseño iterativo y dirigido por pruebas

A menudo, intentar establecer un diseño sin comprender o conocer totalmente los requisitos resulta en software que es innecesariamente más complejo de lo que podría ser. Mayor complejidad implica mayor propensión a fallos y mayor dificultad en su mantenimiento y posterior expansión. Los argumentos usados para defender este diseño temprano es que el coste de realizar cambios en el diseño crece rápidamente a medida que se avanza en el desarrollo.

Esta es otra de las razones por las que existe tanto interés en las pruebas continuas automatizadas en el desarrollo de software. estas pruebas ayudan a reducir una de las fuentes principales del coste de estos cambios en el diseño: la introducción de nuevos defectos. Cada vez que se realiza un cambio, se ejecutan de nuevo todas las pruebas, y el cambio sólo se envía al repositorio central si son superadas. De esta forma, podemos asegurar que toda la funcionalidad garantizada en la anterior versión sigue funcionando actualmente.

No es sólo cuestión de evitar defectos y mantener al diseño flexible: se ha comprobado que los diseños resultantes de escribir primero las pruebas y luego implementar la funcionalidad son de mucha mayor calidad. En muchos casos, el código es difícil de probar no tanto por la complejidad de la lógica que entraña, sino porque no se pensó en las pruebas desde un comienzo, y por lo tanto no es lo bastante flexible y cohesivo.

Otra importante fuente de costes a la hora de revisar un diseño es la realización de los cambios necesarios sobre el propio código, sobre todo cuando el elemento a cambiar es referenciado por muchos otros. Estos costes se pueden reducir en gran medida si disponemos de editores con funcionalidades de refactorización y detección de referencias entre elementos del programa: así ha sido el caso con Java, para el que se utiliza el IDE Eclipse como editor. A diferencia de otros IDE, Eclipse no nos ata a él: el fichero que dirige toda la compilación es un fichero Ant normal y corriente, que puede ser usado fuera de Eclipse sin problemas. Se echa en falta una herramienta parecida para módulos Perl, que sin duda se halla motivada por la mayor dificultad que su desarrollo entraña.

5.3. Transformaciones declarativas dirigidas por datos

Uno de los objetivos en el desarrollo de XMLEye para este Proyecto fue su completa separación de ACL2. El hecho de definir las transformaciones no a través de código Java, sino como ficheros externos fácilmente instalables en XMLEye fue un punto clave: extraer la lógica de ACL2 de XMLEye y moverla a `ACL2::Procesador` fue sólo cuestión de retirar el antiguo diálogo de importación y reemplazarlo por el nuevo sistema de descriptores de formatos.

Al mismo tiempo, estos ficheros externos utilizan un lenguaje especialmente indicado para realizar transformaciones sobre los documentos XML, XSLT. Esto hace que sean mucho más fáciles de definir que si, por ejemplo, usáramos el API DOM.

5.4. Mejora como producto software

Un aspecto que se identificó al final del anterior Proyecto sobre el que se basa el actual fue la necesidad de facilitar otros aspectos no directamente relacionados con su funcionalidad: la instalación era completamente manual, y la documentación se hallaba en un único formato.

Durante el desarrollo de este Proyecto, se han mejorado notablemente los aspectos de instalación, con distribuciones para los módulos Perl que sólo requieren ser descomprimidas para los usuarios e instalación y comprobación automática de dependencias para los desarrolladores. Además, se han elaborado las primeras versiones de los paquetes Debian de los tres componentes, y se han recogido las experiencias obtenidas en una guía de elaboración de paquetes Debian públicamente disponible a través de Internet, e incluida en la presente memoria.

En adelante, toda la documentación se hará en el wiki público (http://wiki.shoyusauce.org/), que además dispone de la posibilidad de mantener múltiples traducciones de cada artículo. Actualmente toda la documentación del wiki se halla disponible en inglés.

6 Conclusiones

6.1. Valoración

El trabajo realizado durante este PFC se puede considerar un ejemplo de cómo Extreme Programming permite, incluso con limitaciones de tiempo, entregar un producto funcional y de calidad, limitando en la medida necesaria el alcance.

El subconjunto de la salida de ACL2 sigue estando limitado a casos de interés sobre todo didáctico, pero se han superado las primeras barreras frente a proyectos de interés científico, con el soporte para proyectos con múltiples ficheros y la separación completa del análisis de los documentos Lisp, las salidas de ACL2, y la producción del código XML.

Al mismo tiempo, se ha podido comprobar la generalidad obtenida en el visor, ahora con identidad propia bajo el nombre de XMLEye, añadiéndole soporte no para un lenguaje concreto, sino para la familia completa de lenguajes del metalenguaje YAML 1.1 y su subconjunto JSON.

Todas las propiedades deseables conseguidas en el Proyecto sobre el que se basa el presente se han mantenido e incluso reforzado:

- La extensibilidad no pasa solamente ya por nuevas formas de ver la salida de ACL2, sino que se pueden añadir nuevos formatos, integrándose con los editores y conversores que necesitemos. Todo sin tener que modificar una sola línea de XMLEye. Esto supone también una ampliación de la capacidad de reutilización de la lógica de XMLEye.
- La transportabilidad entre múltiples sistemas operativos y microarquitecturas se mantiene, y ahora es mucho más fácil instalar XMLEye y sus conversores, especialmente en Windows.
- Se ha traducido la interfaz completa al inglés, y el código también está en proceso de traducción. El nuevo wiki de XMLEye [18] permitirá localizar también fácilmente toda la documentación, y la forja [17] en RedIris recopilará los informes de errores, peticiones de nueva funcionalidad y los ficheros relacionados.

6.2. Mejoras y ampliaciones

6.2.1. Funcionalidad

Se extenderá el soporte de la salida de ACL2, empezando por tutoriales de mayor alcance como [44], cuya salida es 4 veces más larga que la mayor tratada por este programa, y hace uso de varias características aún no filtradas, como macros Lisp.

En cuanto a YAML, se añadirán hojas de usuario para formatos derivados de interés, como los marcadores de Firefox 3, por ejemplo. Habrá que ver si las limitaciones impuestas por el lenguaje Perl sobre `YAXML::Reverse` realmente suponen una pérdida de funcionalidad importante o no.

En futuras versiones, XMLEye incluirá la capacidad de publicar las demostraciones en formato de página Web. Queda por determinar si se hará a través de ficheros estáticos, o si se incrustará un servidor Web sencillo, como Jetty (http://www.mortbay.org/jetty-6/).

Habiendo traducido la interfaz y la documentación al inglés, queda por completar la traducción del código en sí. Esto será más complicado en `ACL2::Procesador`, ya que no hay herramientas de refactorización para Perl. El código de `YAXML::Reverse` ya se encuentra completamente en inglés.

6.2.2. Diseño

El diseño de XMLEye ya no cambiará en gran medida, pero continuará siendo refinado y pulido, al mismo tiempo que se implementan más pruebas de unidad sobre los propios modelos de presentación de los documentos y los formularios principal y de formatos admitidos.

La creación de nuevas interfaces, como la interfaz Web, ayudarán a localizar nuevas vías de mejora de las pruebas y del diseño actual. Se definirán nuevos tipos de visualizaciones, basadas en tecnologías como SVG (Structured Vector Graphics) o JavaFX, y se integrarán motores XHTML más avanzados, como el de los proyectos Lobo (http://www.lobobrowser.org/) o Flying Saucer (https://xhtmlrenderer.dev.java.net).

Se investigarán formas de definir algún tipo de pruebas de unidad sobre las hojas de usuario XSLT: una posibilidad es utilizar asertos XPath, por ejemplo, que son lo bastante flexibles como para asegurar comprobaciones potentes, y al mismo tiempo mucho más robustos que comparar directamente el código fuente con los resultados esperados.

El diseño general de los módulos ya sigue las mejores prácticas del CPAN, pero es posible que `ACL2::Procesador` siga cambiando a una escala considerable: el manejo

de macros seguramente requerirá una revisión importante del diseño actual, con la posibilidad de instrumentar el código Lisp original para obtener más información.

6.3. Otros aspectos de interés

El empaquetado ha sido notablemente mejorado, pero aún hay cosas por hacer: mediante Launch4J se podrían crear ejecutables para XMLEye en Windows que detectaran la carencia de un JRE y dirigieran al usuario a la página de descarga de Sun. De todas formas, existen paquetes para las versiones más recientes de la distribución GNU/Linux Ubuntu y archivos comprimidos listos para usar, además de las usuales distribuciones de código fuente.

Además, ahora el Proyecto completo se puede ejecutar al 100 % sobre software libre, gracias a los esfuerzos de los proyectos OpenJDK [45] e IcedTea [33] para crear una versión completamente libre de J2SE 5.0.

Se desean enviar los paquetes Debian de XMLEye y sus conversores al repositorio de paquetes Debian, y `YAXML::Reverse` al CPAN, para facilitar su adopción por usuarios potenciales.

7 Manual del usuario de XMLEye y conversores asociados

7.1. Instalación de XMLEye

En este apartado cubriré la instalación de XMLEye en sus diferentes formas, tanto a nivel de usuario local como a nivel de sistema. Para instalar los conversores asociados y sus hojas de usuario y descriptores de formato, referirse al apartado 7.2 (página 167).

7.1.1. Requisitos previos

Se necesita tener instalado un entorno Java compatible con J2SE 5.0 o superior. Su instalación se realiza automáticamente si utilizamos los paquetes Debian.

Windows

Tendremos que ir a `http://java.sun.com/javase/downloads/index.jsp` y descargar una edición reciente del JRE de acuerdo a nuestro sistema operativo. Tanto si hemos obtenido la versión que no requiere conexión como la que descarga sólo lo necesario a través de Internet, todo lo que tendremos que hacer es ejecutar el instalador y seguir las instrucciones.

GNU/Linux

Si utilizamos una distribución basada en Debian, podemos probar a instalar los paquetes `icedtea-7-jre` o `openjdk-6-jre`. OpenJDK es la iniciativa de Sun, que a fecha de hoy (03/07/2008) es prácticamente 100 % libre salvo por algunas pequeñas partes que el proyecto IcedTea ha reemplazado utilizando código del proyecto GNU Classpath. Podemos instalar una versión de OpenJDK 6.0 con los reemplazos de IcedTea en Ubuntu 8.04 "Hardy Heronz usarla como entorno Java por defecto con estas órdenes:

```
sudo aptitude install openjdk-6-jre
sudo update-alternatives --config java
```

Escogeremos la entrada de `openjdk-6-jre` y pulsaremos Intro, terminando con este paso. Si estamos utilizando openSUSE 10.3, podemos usar sin problemas el entorno J2SE 5.0 de Sun que incluye de fábrica. En caso de que no estuviera instalado por alguna razón, tendríamos que instalar los paquetes `java-1_5_0-sun*` a través del gestor de paquetes de YaST.

> NOTA
>
> Actualmente, OpenJDK sigue teniendo pequeños defectos en la forma en que sitúa los componentes Swing. Es posible que algunos diálogos tengan un aspecto distinto al normal, con botones demasiado grandes, por ejemplo. El JRE original de Sun no tiene estos problemas, pero no es 100 % libre. De todas formas, los efectos de este problema son puramente estéticos.

7.1.2. Instalación desde distribuciones precompiladas

> NOTA
>
> Si se usa una distribución basada en Debian, se debería considerar el uso de los paquetes Debian, que son mucho más cómodos de usar.

Un único usuario, GNU/Linux

El proceso es muy sencillo: sólo hay que descargar el `-dist-tar.gz` más reciente de XMLEye de https://forja.rediris.es/frs/?group_id=233 y descomprimirlo bajo nuestro directorio personal. Para ejecutar XMLEye, basta con ejecutar el guión `/home/`*`usuario`*`/xmleye/xmleye` tras pulsar la combinación ALT + F2. Una opción más cómoda para los habituales de la línea de órdenes es añadir la siguiente línea a `~/.bashrc`:

```
export PATH=$PATH:~/xmleye
```

Haciendo esto, se puede abrir cualquier documento compatible desde la línea de órdenes mediante:

```
xmleye (ruta absoluta o relativa)
```

Según este procedimiento, las hojas de usuario se instalarán en `/home/usuario/-xmleye/xslt`, y los descriptores de formatos en `/home/usuario/xmleye/formats`. Las opciones se guardarán en `/home/usuario/xmleye`.

Múltiples usuarios, GNU/Linux

Un caso más complejo es cuando queremos instalarlo para varios usuarios, pero queremos tener ajustes distintos para cada usuario (las hojas son comunes a todos). Al igual que en el caso anterior, deberemos tener instalado un JRE compatible con J2SE 5.0 o superior antes que nada, pero después usaremos una distribución diferente de los ficheros.

La distribución "ideal"sería la del paquete Debian, pero como es un poco más compleja de lo que necesitamos, nos limitaremos a un término medio. Primero descomprimiremos la última distribución de XMLEye (el enlace de descarga está en la sección anterior) a `/opt`, que crearemos si no existe ya:

```
sudo mkdir /opt
cd /opt
sudo tar xjf (ruta a distribución)
```

Tendremos que retocar el guión de lanzamiento un poco. XMLEye cuenta con dos variables de entorno, `XMLEYE_PREF_DIR` y `XMLEYE_FORMATS_DIR`, que indican la ruta en la que se guardarán las preferencias y los formatos personalizados por el usuario. Además, XMLEye supone que las hojas de usuario se hallan bajo el subdirectorio `xslt` de la ruta sobre la cual es lanzado.

Teniendo todo esto en cuenta, habrá que sustituir las líneas que afectan a `PROGRAM_-DIR` y `PROGRAM_JAR` por:

```
PROGRAM_DIR=/opt/xmleye
PROGRAM_JAR=xmleye.jar
export XMLEYE_PREF_DIR=$HOME/.xmleye
export XMLEYE_FORMATS_DIR=$HOME/.xmleye
mkdir -p $XMLEYE_PREF_DIR
```

De forma similar a la sección anterior, los usuarios podrían directamente ejecutar XMLEye a través de la ruta completa. Para que todos los usuarios puedan ejecutar XMLEye por nombre, se puede añadir esta línea a `/etc/environment`:

```
export PATH=$PATH:/opt/xmleye
```

Si queremos que aparezca una entrada de menú y que se asocien los ficheros XML con XMLEye, entonces tendremos que, además de seguir el paso anterior, crear el fichero `/usr/share/applications/xmleye.desktop` con este contenido:

```
[Desktop Entry]
Encoding=UTF-8
Version=1.0
Type=Application
Terminal=false
Exec=xmleye %U
Comment[es]=Visor genérico de XML dirigido por datos
Name=XMLEye
Comment=Generic data-driven XML Viewer
GenericName=Generic XML viewer
GenericName[es]=Visor genérico de XML
Icon=accessories-text-editor
Categories=Utility;TextEditor;
MimeType=application/xml;
```

Actualizaremos las bases de datos y reiniciaremos los menús de GNOME con:

```
sudo update-desktop-database
sudo update-menus
sudo killall gnome-panel nautilus
```

Ya debería de aparecer la entrada de XMLEye en el menú principal de GNOME.

Las hojas de usuario estarán bajo `/opt/xmleye/xslt`, y los descriptores de formato en `/opt/xmleye/formats`.

Windows

Descargamos y descomprimimos en alguna carpeta el `-dist-tar.gz` más reciente de XMLEye de https://forja.rediris.es/frs/?group_id=233.

Para ejecutar XMLEye, haremos doble clic en el fichero `xmleye.jar` de la carpeta en que hemos descomprimido la distribución.

7.1.3. Instalación desde paquetes Debian

Esta es la opción a seguir siempre que sea posible, ya que además de ser más sencilla, permitirá recibir actualizaciones de forma automática.

Los paquetes han sido desarrollados para Ubuntu Gutsy, pero deberían funcionar en cualquier distribución basada en Debian reciente.

Los pasos a seguir son:

1. Descargaremos la firma digital de los paquetes disponible bajo `http://www.shoyusauce.org/packages/claveDebian.asc`.
2. Añadiremos la firma al anillo de confianza de Apt. En primer lugar lanzaremos la opción *Sistema* → *Administración* → *Orígenes de software* bajo el menú principal de GNOME.

   Hecho esto, seleccionaremos la pestaña *Autentificación* y pulsaremos en el botón *Importar clave...*, tras lo cual seleccionaremos la firma que antes descargamos.

   Aún no cerraremos la ventana: nos queda una cosa por hacer.
3. Ahora añadiremos los repositorios de paquetes binarios y paquetes de fuentes de *XMLEye* y sus hojas de estilos. Esta vez iremos a la pestaña *Software de terceros*.

   Pulsaremos en *Añadir* e introduciremos esta línea tal y como está:

   ```
   deb http://www.shoyusauce.org/packages/ubuntu/ gutsy main
   ```

   Volvemos a pulsar en *Añadir*, pero esta vez introducimos esta línea:

   ```
   deb-src http://www.shoyusauce.org/packages/ubuntu/ gutsy main
   ```

   Ya podemos pulsar en *Cerrar* para cerrar este diálogo, y solicitar la actualización de nuestras listas de paquetes en el diálogo subsecuente pulsando en *Recargar*. Una vez haya terminado, estaremos listos para instalar *XMLEye* y otros paquetes de apoyo, como *pprocACL2*.
4. Lanzaremos *Sistema* → *Administración* → *Gestor de paquetes Synaptic* y pulsaremos en el botón *Buscar* de la barra de herramientas.

   Introduciendo "xmleye.[en] el campo de búsqueda obtendremos un único resultado en el que podremos hacer doble clic para marcar para su instalación. También podríamos seleccionar otros paquetes con los conversores, descriptores y hojas de estilos específicas de otros formatos, como "libacl2-procesador-perl.[o] "libyaxml-reverse-perl", de la misma forma.

   Una vez todos los paquetes que deseamos instalar se hallen marcados, pulsaremos en *Aplicar* de la barra de herramientas para confirmar los cambios.

5. ¡Listo! Ya podemos lanzar XMLEye a través de *Aplicaciones* → *Accesorios* → *XMLEye* del menú principal de GNOME.

> NOTA
>
> Sólo un detalle: todas las opciones que establezcamos irán a parar al subdirectorio `.xmleye` bajo nuestro directorio personal, es decir, `/home/`*`nombredeusuario`*.

La ruta bajo la cual tendremos que instalar las hojas de usuario será `/usr/share/xmleye/xslt`, y en `/usr/share/xmleye/formats` se hallarán los descriptores de formato.

7.1.4. Compilación del código fuente

> 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 [14].

Tras obtener un JDK compatible con J2SE 5.0 o superior, como el de OpenJDK 6.0 o 7.0, IcedTea 6.0 o 7.0, o los originales de Sun, tendremos que instalar además la herramienta Apache Ant y el entorno de pruebas de unidad JUnit, en una de sus versiones 3.X. En Ubuntu 8.04 "Hardy Heron", esto se puede hacer mediante:

```
sudo aptitude install openjdk-6-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/csl2-xmleye/XMLEye/trunk \
  xmleye
```

Ya podemos introducirnos en `xmleye` y aprovechar los objetivos ya definidos en el fichero `build.xml` de Ant. Si se desea instalar `xmleye` a partir de fuentes, se recomienda usar el objetivo `dist` e instalar la distribución según alguno de los métodos antes expuestos.

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 .ªcerca de". Útil a la hora de diseñar la interfaz.

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.

Más cosas a tener en cuenta: 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.

7.2. Instalación de conversores asociados

7.2.1. ACL2::Procesador: convertidor de demostraciones de ACL2

Como muestra de las capacidades de XMLEye, desarrollé en paralelo un conjunto de hojas de usuario de preprocesado y visualización, junto con un tipo de documento para ACL2. Esto nos permite demostraciones para dicho sistema en formato `.lisp` siguiendo una estructura arbórea donde cada nodo se muestra como un hipertexto con enlaces a otras partes de la demostración.

La subcarpeta `t/testInputs` del fichero de fuentes con nombre de la forma `ACL2-Procesador-*.tar.gz` más reciente contiene algunos ejemplos de interés, extraídos de tutoriales reales de aprendizaje del uso de ACL2.

Primero nos ocuparemos de las dependencias, y luego veremos tres formas de instalar el convertidor, el descriptor de formato y las hojas de usuario.

Dependencias

Las dependencias a instalar variarán según el método de instalación.

ACL2 2.9 o superior Se ha depurado `ACL2::Procesador` también con ACL2 3.1 y 3.3.

GNU/Linux En una distribución basada en Debian, instalar ACL2 es tan sencillo como ejecutar la siguiente orden (necesitamos *gcc* para poder certificar libros):

```
sudo aptitude install acl2 gcc
```

Otras distribuciones requerirán del proceso manual de instalación, documentado bajo `http://www.cs.utexas.edu/users/moore/acl2/v3-3/installation/installat` `html`. Es sencillo, pero la compilación de los libros puede llevar mucho tiempo (alrededor de 8-10 horas).

Windows Descargaremos e instalaremos la distribución completa de ACL2 preparada por Jared Davis, disponible en `http://www.cs.utexas.edu/users/moore/acl2/v3-3/distrib/windows/`. Incluye todo lo que podríamos necesitar para trabajar con ACL2:

- ACL2 3.3
- GCL 2.6.7, el entorno Lisp que necesitamos, junto con el compilador GCC necesario para certificar libros.
- El editor GNU Emacs en su versión 21.3.
- Documentación de ACL2.
- Una colección de libros previamente certificados.

Perl 5.8.6 o superior

GNU/Linux La gran mayoría de las distribuciones lo incluyen de fábrica, pero en el caso en que no lo tuviéramos, podríamos instalarlo en una distribución basada en Debian con:

```
sudo aptitude install perl
```

Otra opción sería descargar el código fuente de http://www.perl.com/download.csp y compilarlo, pero no se cubrirá dicha alternativa aquí.

Windows Utilizaremos la edición 5.10 más reciente de Strawberry Perl, disponible bajo http://strawberryperl.com/. Sólo hemos de seguir los pasos del instalador.

Módulos Perl: *PAR* Una vez hayamos instalado Perl (véase la sección), sólo necesitamos ejecutar una orden más. No es necesario seguir la configuración manual del acceso a CPAN en ninguno de los dos casos.

GNU/Linux

```
sudo cpan PAR
```

Windows

```
cpan PAR
```

Módulos Perl: *PAR::Packer*

GNU/Linux Si estamos usando una distribución basada en Debian reciente, podemos ejecutar:

```
sudo aptitude install libpar-packer-perl
```

De lo contrario, ejecutaremos:

```
sudo cpan PAR::Packer
```

Windows Tendremos que obtener la última copia del código fuente de *PAR::Packer* del repositorio SVN, ya que la versión más reciente en el CPAN a día de hoy (03/07/2008), la 0.980, no incluye el arreglo de un defecto que imposibilitaba su instalación en Strawberry Perl. Para ello, instalaremos el cliente Subversion TortoiseSVN, disponible bajo `http://tortoisesvn.tigris.org/`, y haremos un *checkout* de la dirección `http://svn.openfoundry.org/par/PAR-Packer/trunk/`.

Una vez hayamos descargado el código, abriremos una ventana del intérprete de órdenes, y dentro del directorio con el código fuente, ejecutaremos:

```
perl Makefile.PL
dmake
dmake test
dmake install
```

Si se nos pide instalar alguna dependencia en alguno de los pasos, aceptaremos.

Biblioteca *libxml2*

GNU/Linux Si estamos usando una distribución basada en Debian, ejecutaremos esta orden:

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

Windows No hay que hacer nada: viene ya incluida con Strawberry Perl.

Instalación mediante paquetes Debian

Si instalamos XMLEye mediante el paquete Debian, sólo tendremos que instalar el paquete `libacl2-procesador-perl`. La próxima vez que iniciemos XMLEye tendremos todo lo necesario a nuestra disposición, incluido ACL2.

Instalación desde distribución monolítica

Necesitaremos previamente haber instalado ACL2, y haber instalado XMLEye descomprimiendo la distribución de la forja. Descargaremos del área de ficheros (`http://forja.rediris.es/frs/?group_id=233`) de la forja de XMLEye el fichero `ACL2-Procesador-*-standalone-*.tar.gz` más reciente que se corresponda con la microarquitectura de nuestra CPU y nuestro sistema operativo, y lo descomprimiremos sobre el directorio en el que instalamos XMLEye.

Instalación desde distribución basada en PAR

Necesitaremos ACL2, Perl y el módulo PAR. El proceso es idéntico al de 7.2.1 (página 170), pero en este caso el fichero a descargar y descomprimir es `ACL2-Procesador-*-par-noarch.tar.gz`, que es independiente de la CPU y el sistema operativo escogido.

Instalación manual a partir de fuentes

Los pasos, tras instalar todas las dependencias, son los siguientes:

1. Descomprimimos el fichero de fuentes bajo `/tmp`, y nos introducimos en sus contenidos:

   ```
   cd /tmp
   tar xzf (ruta a las fuentes)
   cd ACL2-Procesador-*
   ```

2. Instalamos el preprocesador tal y como instalaríamos cualquier paquete Perl. Este proceso resultará muy familiar a cualquier usuario de las *autotools*:

   ```
   perl Makefile.PL
   sudo make
   make test
   sudo make install
   ```

   Es muy probable que Perl nos solicite en la primera orden instalar algunas dependencias. Siempre le diremos que sí. La última orden se podría sustituir por **`sudo checkinstall`** si deseamos poder desinstalarlo fácilmente, instalando `ACL2::Procesador` como un paquete Debian.

3. Lo siguiente es instalar las hojas de usuario, que es tan fácil (suponiendo que la ruta de hojas de usuario es `/home/`*`tunombredeuaurio`*`/xmleye/xslt` como:

   ```
   cp -r xslt/* $HOME/xmleye/xslt
   ```

4. Terminaremos instalando el descriptor de tipo a su sitio:

   ```
   cp types/* $HOME/xmleye/types
   ```

Ejecución independiente de XMLEye

Podemos comproabr que todo va bien ejecutando el guión principal. La forma de hacerlo variará según el método de instalación usado:

- Desde el paquete Debian, o desde fuentes: **`pprocACL2`** desde cualquier ruta.

- Desde la distribución monolítica: ***(ruta)*/pprocACL2**, utilizando la ruta bajo la cual se halla instalado XMLEye. Podríamos añadir esta ruta al PATH si quisiéramos.
- Desde la distribución basada en PAR: **perl -MPAR (ruta)/pprocACL2.par**, utilizando la ruta bajo la cual se halla instalado XMLEye.

Deberíamos obtener una salida como la que viene a continuación. Por razones técnicas, es posible que en ciertos entornos la distribución basada en PAR no produzca ninguna salida, pero esto no es un problema: seguirá funcionando de forma normal una vez le pasemos un fichero de entrada, imprimiéndose el XML resultante por salida estándar, y los mensajes de estado por la salida de errores.

```
Usage:
  pprocACL2 [opciones] [entrada Lisp ACL2]

Options:
  --help, -h, -?
          Se limita a mostrar un mensaje corto de ayuda.

  --man, -m
          Muestra un mensaje detallado de ayuda en formato man.
```

7.2.2. YAXML::Reverse: conversor de documentos YAML a XML

Este otro conversor se desarrolló para abrir con XMLEye la familia completa de documentos con lenguajes basados en YAML 1.1 o JSON. Al igual que `ACL2::Procesador`, incorpora las hojas de usuario y descriptores de formatos necesarios para su funcionamiento. Incluye soporte para anclias y alias de YAML.

También en este caso se pueden ver algunos ejemplos de entradas aceptadas en `t/-testInputs` de las fuentes, disponibles como ficheros con nombres del estilo de `YAXML-Reverse-*.tar.gz` en el área de ficheros de la forja (http://forja.rediris.es/frs/?group_id=233).

Para evitar repeticiones innecesarias, aprovecharemos las instrucciones de instalación de algunas de las dependencias y algunos de los pasos de los métodos de instalación de `ACL2::Procesador`. Comentaremos únicamente los aspectos que cambian.

Dependencias

Hay algunas dependencias de *YAXML::Reverse* cuyos métodos de instalación ya se han mencionado en la sección §7.2.1 análoga de `ACL2::Procesador`:

Perl Véase la página 168.

PAR Véase la página 169.

PAR::Packer Véase la página 169.

Bibliotecas XML Véase la página 170.

Bibliotecas XSLT También puede que nos hagan falta las bibliotecas para XSLT. Dependiendo del sistema operativo sobre el que trabajemos, los métodos de instalación cambiarán.

GNU/Linux Necesitaremos instalar los paquetes de desarrollo para las bibliotecas *exslt*, *gdbm* y *gcrypt*. En una distribución basada en Debian, usaremos:

```
sudo aptitude install libexslt-dev \
                      libgdbm-dev  \
                      libgcrypt-dev
```

Windows Como instalar manualmente las bibliotecas es demasiado complejo, usaremos una distribución precompilada del módulo *XML::LibXSLT*. Para ello, ejecutaremos esta orden bajo la interfaz de línea de órdenes:

```
ppm install XML::LibXSLT
```

Instalación mediante paquetes Debian

Una vez instalemos XMLEye mediante el paquete Debian, sólo habrá que instalar el paquete `libyaxml-reverse-perl`, cerrar y volver a abrir XMLEye y todo quedará listo.

Instalación desde distribución monolítica

En este caso utilizaremos el fichero más reciente que concuerde con nuestro entorno que siga el patrón `YAXML-Reverse-*-standalone-*.tar.gz`.

Instalación desde distribución basada en PAR

Esta vez las distribuciones (que siguen el patrón `YAXML-Reverse-*-par-*.tar.gz`) serán específicas del entorno, pero por lo demás es todo igual.

Instalación manual a partir de fuentes

El patrón del nombre de fichero cambia a `YAXML-Reverse-*.tar.gz` más reciente, y no se necesita ACL2, sino las bibliotecas de XSLT. El resto de las dependencias siguen siendo necesarias.

Ejecución independiente de XMLEye

El proceso es análogo al de la sección 7.2.1 (página 171), pero esta vez el guión principal es `yaml2xml` (y el PAR es `yaml2xml.par`), y la salida que deberíamos obtener tendría que ser similar a ésta:

```
$ yaml2xml

yaml2xml, using YAXML::Reverse 0.3.4
Usage: yaml2xml [path to .yaml]
```

7.2.3. Instrucciones genéricas

Estas instrucciones son las más detalladas, y sirven para cualquier hoja de usuario disponible actualmente y en el futuro, siempre que no varíe el diseño de XMLEye.

Instalación de hojas de usuario

Una hoja de usuario nos permite mejorar XMLEye en una de las siguientes formas:

- Ver la estructura de un documento XML de otra forma. Un ejemplo sería ver sólo un resumen del documento, cambiar el orden para su preprocesamiento, o decorar los nodos del árbol con nuevas etiquetas e iconos.
- Ver el contenido de un nodo del documento de otra forma. Podemos hacer cualquier cosa que se nos ocurra con XHTML 1.1 y CSS (no hay soporte para Javascript), y establecer vínculos entre nodos del árbol y nodos de otros documentos.

Realmente, no son más que conjuntos de hojas XSLT que siguen una determinada estructura para permitir su localización y su integración con otras hojas, ya que una hoja de usuario puede ser una especialización de otra hoja.

La ruta donde se guardan las hojas de usuario variará dependiendo del método de instalación que hayamos seguido. Para más detalles, referirse al apartado 7.1 (página 161). Nos encontraremos bajo dicha ruta una serie de ficheros (sólo de interés para

desarrolladores) y dos subdirectorios, `preproc` y `view`, en cuyo interior habrá un subdirectorio por hoja de usuario de preprocesado o visualización, respectivamente.

Instalar una hoja de usuario no es más entonces que asegurarnos que su subdirectorio se halle en el lugar correcto, y reiniciar XMLEye. Encontraremos entradas con los mismo nombres que los subdirectorio correspondientes en las entradas *Preferencias → Preprocesamiento* y *Preferencias → Visualización* de XMLEye. Por defecto tendremos instalado como mínimo las hojas de usuario de preprocesado y visualización `xml`, y la hoja de visualización `xmlSource` a partir de la versión 1.22.

Instalación de nuevos formatos de documentos

A pesar de su nombre, XMLEye puede visualizar documentos de cualquier formato, usando un conversor si no se trata de un documento XML originalmente, e integrarse con su editor (monitorizando el fichero en busca de cambios) siempre que se le dé la información correspondiente. Esa información viene integrada en un *descriptor de formato de documento*.

Dichos descriptores se nombran usando la extensión `.format` y siguen un formato XML sencillo, en cuyos detalles no entraremos aquí (véase el Manual del Desarrollador). Lo único que nos importa es su instalación, que es tan sencilla como copiar el fichero `.format` correspondiente a la ruta de tipos que le corresponde según el tipo de instalación que hayamos realizado.

Una vez instalado y reiniciado XMLEye, podremos abrir cualquier documento que tenga las extensiones especificadas en el tipo de documento instalado sin más problemas, siempre que su editor y preprocesador (si tiene alguno) se hallen debidamente instalados y tengan órdenes que puedan ejecutarse desde el directorio principal de XMLEye (posiblemente usando ejecutables bajo los directorios dentro de la variable de entorno `PATH`).

7.3. Uso y configuración

Este apartado de la guía trata acerca del empleo de XMLEye y de su configuración. Para cuestiones relacionadas con la instalación o con detalles de implementación, referirse a los apartados anteriores o al Manual del Desarrollador, respectivamente.

A lo largo de esta guía nos referiremos exclusivamente a las opciones de los menús, pero prácticamente toda acción tiene una combinación equivalente en el teclado,indicada en la parte derecha del elemento del menú correspondiente (y si no, es un fallo del que se agradecería una notificación ;-): informe de él en https://forja.rediris.es/tracker/?group_id=233). Además, las opciones más

comunes poseen un botón en la barra de herramientas, cuyo icono es una versión ampliada del icono de la entrada de menú correspondiente.

Podemos cambiar entre los botones de la barra de herramientas, el árbol y el navegador mediante CTRL + TAB en de izquierda a derecha y de arriba abajo, o mediante SHIFT + CTRL + TAB en dirección inversa.TAB y SHIFT + TAB siguen funcionando también de la forma usual, pero se hallan redefinidos en el navegador para navegar por los hipervínculos.

La forma de lanzar el programa se detalla en la guía de instalación, y variará según el método seguido.

7.3.1. Apertura y edición de documentos

Bajo el menú *Fichero*, disponemos de *Abrir...*, en donde podremos seleccionar un fichero de cualquier tipo, o uno de los formatos que tengamos instalados.

Una vez hayamos seleccionado el fichero, XMLEye detectará a partir de la extensión de qué formato se trata y aplicará el convertidor indicado, si lo hay. Una vez el fichero se halle disponible en formato XML, pasará por la hoja de preprocesado y se mostrará su árbol en pantalla junto con la visualización de su nodo raíz.

Para editar el fichero fuente con el editor definido en el descriptor de formato, usaremos *Editar fuente*. Si *Preferencias* → *Actualización automática* se halla activo, XMLEye monitorizará el fichero en segundo plano y actualizará el árbol XML cada vez que se produzcan cambios en él, indpendientemente de si estuviera o no el editor abierto.

Por último, podemos acceder a los 5 últimos documentos abiertos con éxito, o *Salir* del programa.

Para cerrar un documento, actualmente no hay ninguna entrada en el menú *Fichero*, pero puede hacerse pulsando la cruz de su pestaña correspondiente. Para cerrar todos los documentos, podemos pulsar en el botón de cierre general en el extremo derecho del componente de pestañas.

7.3.2. Navegación por los documentos

El documento XML tras ser preprocesado sigue una estructura arbórea normal y corriente, por lo que podemos realizar las acciones usuales mediante las entradas del menú *Navegar*. En particular:

Buscar... Abre un diálogo de búsqueda en el que podemos buscar siguiendo diversos parámetros. Además de las opciones usuales de búsqueda por palabras completas o por concordancia de mayúsculas, se puede marcar el ámbito de la búsqueda.

Buscar siguiente Repite la última búsqueda realizada, mostrando un error en caso de que no se halla realizado una anteriormente. Si hemos llegado al último resultado de la última búsqueda, volveremos al primero después.

La repetición de una búsqueda es muy rápida, ya que XMLEye se ocupa de almacenar los resultados tras la primera petición y va dando directamente los resultados que le siguen dispués.

Padre Sube un nivel en la jerarquía del árbol, o no hace nada si estamos en la raíz.

Hijo Baja un nivel en la jerarquía del árbol, o no hace nada si estamos en un nodo hoja.

Hermano anterior Se desplaza al anterior nodo hijo del padre del nodo actual, o no hace nada si estamos en su primer hijo.

Hermano siguiente Se desplaza al siguiente nodo hijo del padre del nodo actual, o no hace nada si estamos en su último hijo.

Expandir todo Expande el contenido de todos los nodos del árbol, de tal forma que todo nodo quede directamente visible, a menos que haya sido ocultado explícitamente por la hoja de visualización.

Contraer todo Contrae el contenido de todos los nodos del árbol, de tal forma que únicamente el nodo raíz quede visible.

Aunque casi toda acción es accesible mediante ratón, se recomienda aprender los accesos directos mediante teclado, ya que agilizan enormemente los desplazamientos. En la tabla 7.1 (página 178) se listan todos los accesos directos mediante teclado para navegar por el árbol del documento, tras hacer clic en él.

Podemos cambiar entre los documentos haciendo clic en sus pestañas correspondientes, o pulsando en la flecha hacia abajo justo a la izquierda de la cruz en el extremo derecho de la barra de pestañas: se desplegará una lista con los nombres de los documentos abiertos, pudiendo saltar directamente al documento que deseemos.

7.3.3. Navegación por la vista del nodo actual

Este área muestra la información generada por la hoja de usuario de visualización a partir del nodo actual en formato XHTML, pudiendo activar los enlaces a través del ratón o pulsando INTRO tras seleccionarlo a través de un recorrido con TAB y SHIFT + TAB.

Para viajar a los componentes Swing anterior o siguiente, se usarán las combinaciones CTRL + TAB o CTRL + SHIFT + TAB respectivamente, siguiendo las recomendaciones de Sun.

Estas combinaciones y otras se recogen en la tabla 7.2 (página 178).

Combinación de teclas	Efecto
ARRIBA	Selecciona el nodo anterior en la vista actual del árbol.
ABAJO	Selecciona el nodo siguiente en la vista actual del árbol.
IZQUIERDA	Sube al nodo padre si el nodo está cerrado, y cierra el nodo actual de lo contrario.
DERECHA	Abre el nodo actual si está cerrado, y baja al primer hijo de lo contrario.
INICIO	Selecciona el primer nodo en la vista actual del árbol.
FIN	Selecciona el último nodo en la vista actual del árbol.
ALT + ARRIBA	Selecciona al nodo padre del actual.
ALT + ABAJO	Selecciona al primer hijo del nodo actual.
ALT + IZQUIERDA	Selecciona al hermano anterior del nodo actual.
ALT + DERECHA	Selecciona al hermano siguiente del nodo actual.

Cuadro 7.1: Accesos de teclado para navegación por el árbol

Combinación de teclas	Efecto
Flechas	Desplazamiento unitario en la dirección indicada.
Inicio	Desplazamiento al inicio de la vista.
Fin	Desplazamiento al final de la vista.
AvPág	Desplazamiento de un bloque hacia delante.
RePág	Desplazamiento de un bloque hacia atrás.
Tab	Selección cíclica del siguiente enlace.
SHIFT + TAB	Selección cíclica del anterior enlace.
RePág	Activación del enlace seleccionado.
CTRL + TAB	Cambio al siguiente componente.
CTRL + SHIFT + TAB	Cambio al anterior componente.

Cuadro 7.2: Accesos de teclado para navegación por la vista del nodo actual

7.3.4. Personalización de la presentación

Podemos cambiar en cualquier momento la hoja de usuario empleada para el preprocesado del documento ocupada de generar la estructura arbórea que vemos en el panel izquierdo. Lo haremos seleccionando una opción distinta del submenú *Preferencias → Preprocesado*. Se regenerará el árbol XML del documento y se nos mostrará el nodo raíz.

Podemos hacer lo mismo con la visualización de cada nodo bajo el submenú *Preferencias → Preprocesado*. Se actualizará el nodo actual de forma automática.

> NOTA
>
> Los dos tipos de hojas de usuario son independientes, pero a veces conseguirá muchos mejores resultados si usa una hoja de visualización que esté diseñada de forma acorde a la hoja de preprocesado utilizada. Tal es el caso de la hoja de visualización `ppACL2`, por ejemplo.

7.3.5. Personalización de los formatos aceptados

Cualquier usuario puede personalizar los ajustes de los descriptores de formatos instalados, cambiando su nombre mostrado en el diálogo de selección, la orden de conversión, la orden de edición y las extensiones correspondientes a su libre albedrío. Esto se hace en el diálogo disponible bajo *Preferencias → Formatos admitidos...*.

La clave `%s` será sustituida por la ruta al documento en las órdenes de conversión y edición. La orden de conversión puede dejarse vacía para formatos que ya se hallen en XML. Ninguna extensión puede contener espacios.

La copia global de todo descriptor siempre se mantiene intacta, por lo que podemos volver a ella cuando queramos pulsando el botón *Restaurar*. Podemos confirmar los cambios realizados con el botón *Aceptar*, o cancelarlos mediante *Cancelar*.

8 Manual del desarrollador

8.1. Cómo abrir nuevos formatos con XMLEye

8.1.1. 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 formato, y cómo convertirlo a 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 formato*. 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 formatos a su sitio.

8.1.2. 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 poder ejecutarse desde el directorio de instalación de XMLEye. Es decir, o bien pertenece a un directorio en nuestro PATH, o se halla instalado bajo el directorio de XMLEye, o se lanza a través de otro programa, que sí cumple la condición anterior (por ejemplo, un intérprete de Python o Perl).

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

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

Listado 8.1: Guión principal de 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.

8.1.3. Creación de un descriptor de formato

Para decirle a XMLEye que acepte un nuevo formato, 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:

Listado 8.2: Descriptor de formato para YAXML::Reverse

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

Yendo línea a línea por el fichero anterior, nos vamos encontrando 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.

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

El descriptor de documento viene representado globalmente por un elemento de nombre `format` y del espacio de nombres de descriptores de formatos de XMLEye. Es importante usar el espacio de nombres correcto, ya que XMLEye validará cada descriptor durante su carga mediante un XML Schema, que tiene en cuenta dicho detalle.

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

Estas dos líneas indican el nombre del formato 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. 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 resultado de la conversión, si la hubo), 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 *Actualización automática*.

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

Por último, el elemento `extensions` incluye una serie de subelementos `extension` que relacionan ciertas extensiones de fichero con el formato de documento en cuestión. En otros estándares parecidos como la especificación de `shared-mime-info` de Freedesktop.org ([28]) implementan también un sistema basado en el contenido del fichero, pero por simplicidad no lo he considerado.

8.2. Cómo añadir nuevas visualizaciones de documentos y elementos

8.2.1. Introducción

A través de los descriptores de formatos de documentos, XMLEye puede efectivamente abrir cualquier formato 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 (o el documento XML original) tal y como queda. Utilizaremos las tecnologías XPath [13] y XSLT [12] para definir nuestras *hojas de usuario*: conjuntos cohesivos de hojas de estilos XSLT dedicadas a realizar una determinada transformación. XMLEye implementa algunas extensiones sobre dichas tecnologías para permitir su localización y especialización.

Por ejemplo, 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 usuario 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 usuario de visualización*, que crean esos resúmenes en formato XHTML para cada nodo del árbol seleccionado por el usuario, una vez preprocesado.

En cuanto a posibles problemas de rendimiento: las 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. Además, si algún paso de la transformación de visualización es particularmente costoso, podemos usar la hoja de preprocesado para añadir nodos ocultos y almacenar los resultados temporales que necesitemos.

En el resto de esta sección veremos la estructura general de los repositorios que agrupan a las hojas de usuario, mencionaremos cómo localizar las hojas de usuario y heredar de una hoja de usuario base, y los aspectos particulares de las hojas de preprocesado y de visualización. Terminaremos ilustrando estos conceptos mediante un paseo por las partes más importantes del código de una hoja de preprocesado y de una hoja de visualización.

8.2.2. Estructura de los repositorios 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` 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 con la URI `current_preproc`.

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 grandes rasgos, a la ruta de la hoja de preprocesado seleccionada automáticamente por el usuario. Veremos los detalles 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` de esta hoja XSLT) importando a la URI especial `current_view`.

¿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`, como veremos posteriormente, no tendremos que preocuparnos por esto.

> NOTA
>
> Si lo tenemos que implementar de todas formas, recomiendo utilizar la función XPath **`generate-id`** definida en el estándar XSLT.

Es obvio 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, `view.xsl` 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 en el cambio a XSLT 2.0, pero las dejaré ahí de todas formas, ya que tampoco tienen ningún impacto negativo.

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 apartado veremos por qué nos interesaría hacerlo así.

8.2.3. Localización de una hoja de usuario

Una requisito fundamental del diseño de XMLEye ha sido siempre su internacionalización, es decir, la implantación de la infraestructura necesaria para poder localizar todo el contenido de la interfaz 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. El proceso de localización es realmente muy simple, y se basa en una selección inteligente del punto de entrada a través del cual se carga el resto de los contenidos de una hoja de usuario. 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 de su punto de entrada, `current_preproc`, a uno de estos ficheros XSLT en el mismo orden:

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

El proceso para las hojas de visualización es análogo a éste, sustituyendo `preproc` por `view` en las rutas. 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. Cada punto de entrada concretará la misma serie de variables y/o plantillas con nombre con contenidos específicos de la combinación idioma-dialecto escogida, e incluirá con `xsl:include` a las hojas que proporcionen la lógica requerida. Al usar una inclusión y no una importación (con `xsl:import`, que después usaremos en otro contexto), nos aseguraremos de que se produzcan errores en caso de colisión de identificadores, y evitaremos sorpresas.

2. La hoja principal con toda la funcionalidad no contendrá cadenas de texto, sino que quedará completamente dependiente de las variables y plantillas con nombre 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.xsl`, que actúa como punto de entrada por defecto e incluye a `principal.xsl`, 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.xsl` incluye a `principal.xsl`, ha de emplear la ruta relativa completa, aunque se halle en el mismo directorio: `view/xml/principal.xsl`, por ejemplo.

8.2.4. Herencia a partir de una hoja 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.

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 análogas para las hojas de preprocesado.

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

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

Como de costumbre, podemos especializar a varios niveles. Así, `summaries` y `reverse` son a su vez especializaciones de `ppACL2`.

8.2.5. Ejemplo de hoja de preprocesado: `xml`

Para tener una mejor idea de cómo elaborar una nueva hoja de preprocesado, daremos un paseo por el código de la hoja más sencilla de preprocesado, `xml`, que normalmente especializaremos para cubrir nuestras necesidades concretas. Iremos mencionando al mismo tiempo algunos conceptos clave de XSLT, pero se recomienda de todas formas completar la información aquí presente mediante la especificación oficial [12]. En la próxima sección examinaremos una hoja de visualización.

Iremos alternando código y explicaciones. Comencemos:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xmleye="http://www.uca.es/xmleye">

<!-- ... a continuación  ...  -->

</xsl:stylesheet>
```

Aquí tenemos primero a la declaración XML, indicando que se trata de un documento XML 1.0 codificado en UTF-8. Después tenemos al elemento raíz de toda hoja de estilos XSLT, `stylesheet`, bajo el espacio de nombres de XSLT. Estamos usando XSLT 1.0: existe XSLT 2.0 actualmente, pero aún no está disponible en XMLEye. Vayamos avanzando por el interior del interior del elemento `stylesheet`:

```
<xsl:template match="/">
  <xsl:apply-templates select="." mode="process-node"/>
</xsl:template>
```

Este es el primer patrón de nuestra hoja de usuario. En XSLT, vamos básicamente recorriendo el árbol XML original a partir de la raíz, `/`, un nodo que representa al documento completo, y no a un elemento en particular. A cada paso aplicamos la plantilla (`xsl:template`) más específica cuya condición (véase el atributo `match`) se cumpla.

Utilizando el elemento `xsl:apply-templates`, pedimos que se siga recursivamente procesando el nodo actual, pero cambiando del modo por defecto por el que empezamos la transformación a `process-node`. Cambiando entre modos podemos cambiar entre distintos conjuntos de plantillas fácilmente, y procesar varias veces el mismo nodo. Es interesante ver que si en una especialización de esta hoja redefinimos esta regla y cambiamos la ruta del `xsl:apply-templates`, podemos podar todo excepto una determinada parte del documento. La siguiente plantilla no produce ninguna salida:

```
<xsl:template match="*" />
```

Esta plantilla vacía coincide con cualquier elemento (que no nodo: excluye a los atributos y nodos de texto, por ejemplo) en el modo por defecto. Al ser tan general, cualquier otra plantilla que definamos tendrá prioridad sobre ella. La utilidad de esta regla es desactivar las reglas por defecto de XSLT, a saber:

- Los atributos no son recorridos.
- Se desciende recursivamente por los elementos.
- Se imprimen los nodos de texto como están.

Mejor dicho: si transformáramos un documento XML con una hoja XSLT vacía sin opciones ni plantillas, lo que haríamos sería quitarle todo el marcado XML y dejarlo en texto puro. Pasamos a la siguiente plantilla:

```
<xsl:template match="*" mode="process-node">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:attribute name="UID">
      <xsl:value-of select="generate-id()"/>
    </xsl:attribute>
    <xsl:apply-templates select="."/>
    <xsl:copy-of select="text()"/>
    <xsl:apply-templates select="." mode="process-children"/>
  </xsl:copy>
</xsl:template>
```

Otra plantilla para cualquier elemento, pero para el modo `process-node` que vimos antes. Esta plantilla es el corazón de la hoja de preprocesado `xml`, y hace un par de cosas por nosotros:

1. Crea una copia del elemento actual.
2. Copia los atributos del elemento actual.
3. Añade el atributo `UID` con el identificador único que necesitamos para ser capaces de seleccionar un nodo.
4. Procesa recursivamente el nodo actual en el modo por defecto. En esta hoja no hace nada de por sí, pero al especializarla podremos añadir plantillas en el modo por defecto que sean aplicadas para añadir los atributos o elementos que consideremos necesario. Podríamos por ejemplo usar algunos de estos atributos especiales:

   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.

   He aquí un ejemplo, extraído de la hoja de visualización `yaxml`, 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>
```

5. Copia los nodos de texto.
6. Procesa recursivamente el nodo actual en el modo `process-children`. Así podemos determinar qué hijos del nodo actual serán procesados, y podremos retirar los que no nos interesen.

Como vemos, esta última plantilla sigue el patrón de diseño Método Plantilla (valga la redundancia), en una versión del principio de Hollywood: "No nos llames; nosotros te llamaremos". En las especializaciones de esta hoja, añadiríamos plantillas que redefinirían partes de su comportamiento y esta plantilla sería la que las aplicaría. Hecha esta puntualización, seguimos:

```
<xsl:template match="*" mode="process-children">
  <xsl:apply−templates select="*" mode="process-node"/>
</xsl:template>
```

Esta plantilla se aplica a todos los nodos en el modo `process-children`: por defecto se procesan todos los hijos del nodo actual. Toda especialización de `xml` puede añadir

plantillas más específicas que sólo procesen algunos de los hijos de ciertos elementos, retirando los demás.

En resumen, si queremos crear nuestra hoja de preprocesado a partir de la hoja `xml`, haremos lo siguiente:

- Añadiremos este elemento como primer hijo de `xsl:stylesheet`:

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

  Al importar y no incluir la hoja, nos aseguramos que todas las reglas y variables de la hoja base tengan menos prioridad, y así podremos redefinirlas como queramos sin que se produzcan colisiones.
- Para añadir nueva información a un elemento, definiremos una plantilla en el modo por defecto que cree los atributos y elementos deseados con `xsl:attribute` o `xsl:element`, por ejemplo.
- Para podar elementos del árbol, definiremos plantillas bajo el modo `process-children` que procesarán sólo algunos de los hijos, efectivamente retirando al resto.
- Para podar todo excepto una parte del árbol, redefiniremos la plantilla del nodo raíz en el modo por defecto para que la ruta del `xsl:apply-templates` señale a esa parte.
- Para cambiar la forma de procesamiento de algunos nodos por completo, podemos definir nuevas plantillas en el modo `process-node` para ellos. Tendremos que tener cuidado de no olvidar añadir el atributo `UID` con el identificador único.

8.2.6. Ejemplo de hoja de visualización: `xml`

En este caso no llegaremos a mostrar otra vez todo el código, ya que muchas partes se repiten. En su lugar, nos limitaremos a las partes más interesantes. Primero nos centraremos en su punto de entrada por defecto, `xml.xsl`, que proporciona cadenas en inglés, y luego pasaremos a describir una plantilla interesante en `main.xsl`.

Punto de acceso por defecto: `ppACL2.xsl`

```
<xsl:stylesheet version="1.0"
  xmlns:xmls="http://www.uca.es/xmleye/xml-stylesheet"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!-- ... omitido ... -->

</xsl:stylesheet>
```

Hemos definido un nombre de espacios para esta hoja de usuario. Lo necesitaremos para nuestras variables con las cadenas de texto localizadas: así evitaremos que otras hojas que especialicen a la nuestra tengan problemas de colisiones de identificadores. Pero primero tenemos que hacer algo más:

```
<xsl:include href="view/xml/main.xsl"/>
```

Este `xsl:include` incluye a la hoja XSLT principal que implementa toda la funcionalidad, y que emplea las cadenas de texto que define este punto de acceso. Como ya se dijo antes, aquí sí conviene usar inclusiones y no importaciones para que, si existen colisiones de identificadores, se produzca un mensaje de error y se eviten sorpresas. En el caso de la herencia, el razonamiento es el contrario: si estuviéramos definiendo una especialización de `xml`, utilizaríamos un `xsl:import` a la URI `view_xml`, para que nuestras definiciones tomaran prioridad sobre las de la hoja base. Ahora ya podemos listar las variables e ir dándoles las cadenas localizadas de texto:

```
<xsl:variable name="xmls:name">Name</xsl:variable>
<xsl:variable name="xmls:value">Value</xsl:variable>
<xsl:variable name="xmls:attributes">Attributes</xsl:variable>
```

En casos más complejos necesitaremos plantillas con nombre en vez de variables, pero aquí no son necesarias.

Resumiendo:

- Utilizaremos variables y plantillas con nombre en los puntos de acceso para almacenar las cadenas o rutinas de generación de salida localizadas al idioma y dialecto en cuestión. Sus identificadores se hallarán en un espacio de nombres propio de la hoja de usuario, para evitar colisiones.
- Incluiremos con `xsl:include`, en vez de importar, las hojas XSLT con la funcionalidad de la hoja de usuario desde el punto de acceso. Ello nos asegurará de que si existe alguna colisión de identificadores, se produzca un error, y no nos llevemos después sorpresas.

Hoja XSLT principal: `main.xsl`

Esta es la hoja que se ocupa de generar el código XHTML a mostrar al usuario. Se compone de dos plantillas:

```
<xsl:template match="*">
  <xsl:call-template name="skeleton">
    <xsl:with-param name="rtf">
      <xsl:if test="@*">
        <h2><xsl:copy-of select="$xmls:attributes"/></h2>
        <table border="1" padding="1"
```

```
                width="60%" align="center">

          <!-- ... omitido ... -->

          </table>
        </xsl:if>
      </xsl:with-param>
    </xsl:call-template>
</xsl:template>
```

Esta es la plantilla a través de la cual se genera la visualización de cualquier nodo seleccionado. Resulta interesante ver cómo llama (usando `xsl:call-template`) a una plantilla con nombre, pasándole mediante `xsl:with-param` el fragmento de XHTML generado internamente como el argumento `rtf`. Usar esta plantilla con nombre nos permite asegurar que tendremos un esqueleto básico para todo nodo, en el que podremos rellenar los distintos espacios con el código que necesitemos. Además, las hojas derivadas podrán aprovechar esta plantilla con nombre sin problemas, o incluso la redefinirán.

Esta plantilla sólo se ejecuta una vez por selección: el punto de entrada global a toda transformación de visualización, `view.xsl`, dirige directamente la transformación al nodo seleccionado. Es lo contrario a lo que se hacía en el preprocesado, en el que se iba recorriendo recursivamente el árbol. La ventaja de cambiar el nodo actual al elemento seleccionado en vez de usar sólo el subárbol con él como raíz como entrada a la transformación es que, si lo necesitamos, podemos usar toda la información del árbol para generar la visualización. Ahora veremos la plantilla `skeleton` a la que se le proporciona el fragmento de XHTML:

```
<xsl:template name="skeleton">
  <xsl:param name="rtf"/>
   <html>
     <body>

        <!-- Lista de ancestros -->
        <xsl:if  test="ancestor::*">
        <h2><xsl:copy-of select="$xmls:ancestors"/></h2>
        <ol>
          <xsl:for-each select="ancestor::*">
          <li>
            <a href="#xpointer({ext:getPath(.,/)})">
              <xsl:choose>
                <xsl:when test="@nodelabel">
                  <xsl:value-of select="@nodelabel"/>
                </xsl:when>
                <xsl:otherwise>
```

```
                <xsl:value-of select="name(.)"/>
              </xsl:otherwise>
            </xsl:choose>
          </a>
        </li>
        </xsl:for-each>
      </ol>
      <hr/>
      </xsl:if>

      <xsl:copy-of select="$rtf"/>

      <!-- ... lista de hijos ... -->
    </body>
  </html>
</xsl:template>
```

Esta plantilla nunca será utilizada automáticamente, pues carece de un atributo `match`, sino que la invocaremos por su nombre, definido en el atributo `name`. Se declara el argumento antes utilizado mediante `param`, y el resto consiste en crear el esqueleto de la visualización XHTML con la lista de los ancestros y los hijos, dentro del cual insertaremos el fragmento XHTML proporcionado.

Antes de insertar ese fragmento XHTML, se añade una lista de enlaces a los nodos ancestros del actual, si los hay (`xsl:if` implementa dicho comportamiento condicional). Podemos ver cómo se utiliza la variable `xmls:ancestors` con una de las cadenas localizadas, y cómo podemos implementar estructuras iterativas y selectivas múltiples utilizando los elementos `xsl:for-each` y `xsl:choose`, respectivamente.

En este fragmento podemos ver además cómo crear enlaces entre los nodos del documento. Empleamos un enlace normal y corriente de XHTML, pero con un ligero matiz: la sintaxis de los enlaces utiliza una variante del esquema *xpointer()* de XPointer [16] sin sus extensiones XPath. A nivel general, se permiten URL del formato `(ruta al fichero)#xpointer(patrón XPath)`, donde debe de estar al menos el patrón. Si no está la parte del fichero, se asume que la ruta enlaza a un nodo del mismo documento, y si está, entonces enlaza a un nodo determinado de otro documento.

Como crear rutas tiene cierta complejidad, se ha definido en el espacio de nombres `xalan://es.uca.xmleye.xpath.XPathPathManager` la función de extensión XPath `getPath`, que calcula la ruta entre un nodo y un ancestro suyo (el nodo actual y la raíz del documento en este caso). Evidentemente, no podemos usar esta función cuando enlazamos a nodos de otros ficheros. En ese caso, deberemos de buscar nuestra propia forma de identificar de forma unívoca al nodo destino con una ruta XPath.

NOTA

También podemos usar enlaces a direcciones Web (del estilo de `http://...`, o anclas HTML (`#ancla`).

En resumen:

- La transformación de visualización no recorre recursivamente el árbol, sino que pasa directamente al nodo actualmente seleccionado.
- Resulta útil utilizar plantillas con nombre para refactorizar nuestro código XSLT y evitar repetirnos. Un uso concreto en la generación de XHTML es el de implementar un esqueleto mínimo de la visualización.
- Podemos crear enlaces entre nodos del mismo documento y de documentos distintos. En caso de que se traten de nodos del mismo documento, se dispone de la función de extensión `getPath` para ayudar a generar la ruta XPath a incluir en el enlace.

A Guía de desarrollo de paquetes Debian

A.1. Introducción

A.1.1. ¿Qué es un paquete Debian?

Un paquete Debian, en principio, no es más que un fichero comprimido que contiene los ficheros de la aplicación, cierta información adicional y un par de guiones específicos de apoyo. A través de un paquete Debian, podemos instalar de forma muy sencilla cualquier software que necesitemos, sin tener que entrar en detalles de cómo está hecha o cómo se configura inicialmente la aplicación.

Además, la información adicional contenida en dicho paquete permite al sistema que se ocupa de gestionarlos (*dpkg*) mantener un control de las dependencias. Así, si instalamos una determinada aplicación, todas las bibliotecas requeridas por ésta serán automáticamente instaladas. El robusto control de dependencias del sistema de empaquetado de Debian es precisamente uno de sus puntos más fuertes frente a otros sistemas de empaquetado, como *RPM* (Red Hat Package Manager).

A.1.2. ¿Por qué se desarrollan paquetes?

Uno podría preguntarse las razones detrás del desarrollo de un paquete: ¿no podríamos simplemente compilar nosotros mismos el código? Efectivamente, podríamos, pero existen una serie de factores que hacen poco factible dicho enfoque:

- *Compilar lleva tiempo, y esfuerzo*: además del considerable tiempo de CPU requerido para compilar cualquier aplicación con un nivel mínimo de complejidad, hay que pensar en sus dependencias. Éstas son transitivas, con lo que tendremos que compilar una biblioteca que sea usada por otra biblioteca que sí use directamente el programa que queremos.

  Un ejemplo extremo puede ser el conocido reproductor multimedia *mplayer*, cuyas dependencias se extienden de forma recursiva a lo largo de decenas de bibliotecas, en una versión con toda la funcionalidad activada.

También hay que pensar en los sistemas «exóticos» de compilación que algunos sistemas usan: no siempre tenemos la suerte de contar con sistemas estándar como el de las autotools, donde sabemos de antemano que basta con hacer esto en el 80 % de los casos:

```
./configure
make
make install
```

- *Falta de uniformidad*: Supongamos que ya hemos compilado todo. ¿Cómo instalamos el programa? No existe un enfoque uniforme a lo largo del gran número y variedad de aplicaciones existentes: cada proyecto es un mundo. Tampoco sabemos en principio cómo configurarlo.

  Además, es muy probable que el programa en su estado actual no esté pensado para ser instalado de dicha forma: puede que haya rutas escritas directamente en el código, o que se hagan ciertas suposiciones poco estándares.

  Más aún: ¿y las asociaciones de fichero? ¿Y la entrada del menú? ¿Estarán bien hechas, y funcionarán bajo todos los entornos de escritorio? ¿Se podrá desinstalar el programa?

En resumen, podría decirse que la utilidad de un paquete Debian se halla en garantizar una experiencia de instalación cómoda y uniforme a lo largo de todo el software disponible, independientemente de cómo esté hecho. Así, se pone algo de orden al problema que se presenta ante la gran configurabilidad y heterogeneidad que presentan los sistemas GNU/Linux.

De todas formas, para no perder la libertad que tenemos al descargar el código fuente, podemos instalar no un paquete con los ejecutables y bibliotecas compilados, sino con su código fuente adaptado para nuestra distribución. Son los paquetes de código fuente (*source packages*): normalmente se usan cuando queremos compilar partiendo de la misma base que el paquete binario de nuestra distribución.

Por ejemplo: el paquete `kernel-source-2.4.27` contiene el código fuente de la versión del kernel de Linux que emplea Debian. Este código trae de por sí una serie de modificaciones que lo diferencian del kernel que podríamos bajarnos de la página oficial, y se halla ya configurado con las mismas opciones que nuestro propio kernel.

A.1.3. ¿Quién desarrolla paquetes?

Así, vemos que hacer un paquete no es algo trivial, pudiendo requerir todo tipo de cambios a un programa. Sin embargo, su conveniencia les hace imprescindibles: no podemos esperar obtener un número aceptable de usuarios a menos que nuestro software sea fácil de instalar, configurar y usar.

Por ello, tendremos que distinguir entre dos roles, que podrán ser cubiertos por las mismas o distintas personas:

- Equipo de desarrollo del programa en sí (equipo original de desarrollo o *upstream*): son los que realmente se ocupan de añadir funcionalidad y corregir errores. El vocablo inglés, cuyo significado es «río arriba», hace referencia al hecho de que las nuevas funciones «fluyen» de este equipo hacia nuevas revisiones del paquete.
- Desarrolladores del paquete: son los que deben adaptar el programa al sistema de empaquetado, controlar las dependencias, y en resumen, hacer que el programa se integre bien con el resto del sistema. Sobre todo en las primeras versiones, es posible que tenga que hacer cambios o mejoras al código de la aplicación, que tendrá que discutir con el equipo original de desarrollo.

  Es frecuente que un paquete vaya pasando por distintos responsables, y que haya todo un equipo detrás y no sólo un responsable (como en el caso del Ubuntu MOTU Team[1], y se halla formado sobre todo por usuarios de dicha distribución).

A.1.4. Notas acerca de esta guía

Escribí esta guía para recoger cierta información que anda dispersa por la Red, y parte de mi (poca) experiencia como la combinación de un repositorio *reprepro* con sistemas de versionado y demás.

Existen otras guías, fantásticas sin duda, como [58]. No intento reemplazarlas, sino complementarlas con más información de otros tipos que podría ser útil, como, por ejemplo, como conseguir tener fácilmente nuestro propio repositorio.

A.2. Preparativos

Antes de empezar a crear nuestro primer paquete, vamos a preparar el entorno con todas las herramientas que necesitaremos. Por supuesto, si no deseamos distribuir nuestro paquete a través de Internet, o gestionar varios paquetes dependientes entre sí, no nos será realmente necesario crear un repositorio local y publicarlo bajo un servidor Web.

> NOTA
>
> A lo largo de esta guía, supongo que el usuario empleará Ubuntu Linux «Gutsy Gibbon» 7.10. Las órdenes exactas podrían variar ligeramente, pero por lo general todo debería funcionar en cualquier sistema basado en Debian.

[1] Es el responsable de mantener los paquetes de los repositorios *universe* y *multiverse* de Ubuntu.

A.2.1. Cómo instalar las herramientas básicas

En primer lugar, requeriremos ciertas herramientas para empezar a desarrollar paquetes Debian. Para ello, ejecutaremos (como haremos a lo largo de esta guía) bajo una terminal de texto, como la disponible desde el menú principal en *Aplicaciones* → *Accesorios* → *Terminal*, la siguiente orden:

```
sudo aptitude install cdbs debhelper autotools-dev fakeroot \
     desktop-file-utils linda lintian devscripts dh-make \
     debian-policy automake-1.9 autoconf
```

Posteriormente a lo largo de este manual iremos describiendo para qué sirven algunos de estos paquetes. Adelantaremos algunos usos útiles de *dpkg*, que después nos serán de mucha ayuda:

dpkg -l *patrón* Permite buscar paquetes que cumplan un cierto patrón. Así, **dpkg -l '*sdl*'** nos buscará todos los paquetes relacionados con la biblioteca SDL. Es importante usar comillas simples, para evitar que una posible sustitución automática del shell interfiera con la orden.

dpkg -L *paquete* Lista todos los ficheros de un paquete determinado.

dpkg -S *fichero* Esta orden es muy útil: lista el paquete al cual pertenece un determinado fichero del sistema.

dpkg-deb -c *fichero* Lista los contenidos de un paquete Debian. Útil para comprobar que todo está en su lugar.

dpkg-deb -x *ficheroruta* Extrae los contenidos del paquete Debian a la ruta que le indiquemos.

dpkg-deb -e *fichero* Extrae el directorio de control `DEBIAN` de un paquete. Así podemos ver los guiones de postinstalación y postdesinstalación, por ejemplo, y otra información de interés.

A.2.2. Cómo conseguir paquetes limpios

Normalmente, desarrollamos el paquete Debian dentro de nuestro sistema habitual. Esto quiere decir que en nuestro sistema ya habrá un gran número de paquetes instalados por nosotros mismos. ¿Cómo podemos asegurarnos de que no hemos olvidado alguna dependencia? Peor aún: ¿se está compilando el programa con las bibliotecas que debiera, o está usando alguna versión que nosotros olvidamos haber instalado hace ya tiempo?

En el sistema de empaquetado Debian, estos problemas se resuelven creando e instalando el paquete no dentro del propio sistema, sino de una imagen de un sistema

«limpio», sin otros paquetes que los esenciales. Dicha imagen se conoce como una *jaula* chroot, a partir de la aplicación del mismo nombre, que nos «encierra» dentro de un árbol de directorios, haciéndonos pensar de forma temporal que se trata del sistema de ficheros completo.

Adicionalmente, el uso de esta jaula nos permite probar el proceso de instalación completo una y otra vez sin peligro de dañar a nuestro sistema, y evitar posibles conflictos que puedan surgir.

Aunque nosotros mismos podríamos (con el suficiente tiempo y esfuerzo) hacernos una jaula y ejecutar las órdenes de construcción en ella manualmente, la mejor opción es hacer uso de alguna serie de guiones escritos específicamente para la construcción de paquetes Debian en éstas. Básicamente, lo que hacen es añadir el proceso de entrada y salida de la jaula sobre lo que ya hacen otros guiones como *dpkg-buildpackage* y *debuild*.

En principio, con *pbuilder* nos bastaría. Sin embargo, tiene un problema: la imagen que crea es simplemente un gran fichero `tar.gz`, que descomprime cada vez que construimos el paquete. Esto hace que sea mucho más lento que una construcción habitual. Lo combinaremos con *cowdancer*, que usa un mecanismo de *copy-on-write*[2] para evitarlo, reduciendo en gran medida dicha sobrecarga.

Así, los pasos a seguir son:

1. Instalamos los dos paquetes que necesitamos, junto con sus dependencias:

   ```
   sudo aptitude install pbuilder cowdancer
   ```

2. Generamos de forma automatizada la imagen del sistema Ubuntu con la siguiente orden. Este proceso puede llevar bastante tiempo: tiene que descargar un Ubuntu mínimo a través de la red.

   ```
   sudo cowbuilder --create \
    --othermirror \
    "deb http://archive.ubuntu.com/ubuntu gutsy universe multiverse"
   ```

3. Tras un buen rato, nuestra imagen estará lista. Añadimos las siguientes líneas al fichero `~{}/.pbuilderrc`:

   ```
   # Seleccionamos los componentes de Ubuntu
   # (son distintos a los de Debian, que son
   #  main, contrib y non-free)
   COMPONENTS="main universe multiverse"

   # Uso de cowbuilder en vez de pbuilder normal (más rápido)
   ```

[2] Este mecanismo usa el árbol de directorios original directamente en un principio, y realiza todas las posteriores escrituras sobre copias de los originales, para mantener el árbol de directorios original limpio, y no tener que realizar una lenta copia completa por otro lado.

```
export PDEBUILD_PBUILDER=cowbuilder

# Aqui pondremos nuestro correo como desarrollador de
# paquetes
export DEBEMAIL="tudireccion@decorreo"
```

A.2.3. Cómo simplificar la distribución

Supongamos que alguien quiere instalar nuestro paquete. ¿Cómo lo consigue? ¿Se lo descarga de alguna web, teniendo que buscar una y otra vez cada vez que necesite algo? ¿Tendrá el cliente que comprobar una y otra vez dicha página web, o tendrá el desarrollador de la aplicación que implementar algún mecanismo de actualización automática, aunque sólo se trate de un guión Perl? No olvidemos que, además, en el caso de una distribución GNU/Linux, podemos tener miles de paquetes instalados (1646 en mi caso, retirando los paquetes esenciales). No parece factible simplemente descargarlo de alguna página web.

En realidad lo que se hace es agrupar los paquetes en repositorios. Se conoce como *repositorio Debian* a un árbol de directorios con una cierta estructura, desde el que el sistema de empaquetado de software de Debian, *dpkg*, pueda instalar software. Dicho repositorio puede estar disponible de forma local en nuestro disco duro, como el que haremos aquí, o en otra máquina que aloje un servidor HTTP (web) o FTP.

Este paso sólo es estrictamente necesario cuando tengamos que construir nosotros mismos varios paquetes que dependan entre sí. De todas formas, recomiendo seguirlo, ya que nos permite depurar ciertas cosas como que el paquete se halle correctamente firmado y nuestra clave pública bien distribuida. Además, podemos usarlo para actualizar a partir de él nuestra versión instalada del mismo paquete, y así reproducir exactamente lo que un usuario normal vería.

Podríamos gestionar manualmente el contenido del repositorio a través de las herramientas del paquete `apt-utils`, pero por simplicidad, usaremos el sistema *reprepro* (anteriormente *mirrorer*) para gestionar todo el proceso.

Seguiremos estos pasos:

1. Instalamos el paquete correspondiente con:

   ```
   sudo aptitude install reprepro
   ```

2. Creamos el repositorio dentro de `/var/packages/ubuntu`:

   ```
   sudo mkdir -p /var/packages/ubuntu/conf
   ```

3. Situamos en el fichero `/var/packages/ubuntu/conf/distributions` las siguientes líneas:

```
Origin: Mi Nombre
Label:  Mi Nombre
Codename: gutsy
Architectures: i386 source
Components: main
Description: Mi propio repositorio
```

Como vemos, tenemos una distribución, *gutsy*, que contiene paquetes binarios para la arquitectura IA-32 (Intel 80386 o superior) y paquetes de código fuente repartidos en un único componente *main*.

4. Ahora ya podemos manipular el repositorio de distintas formas. Posteriormente veremos cómo obtener los ficheros `.deb` y `.dsc`. Por ejemplo:

   - Añadimos paquetes binarios con:

     ```
     sudo reprepro -b /var/packages/ubuntu \
          includedeb gutsy (fichero .deb)
     ```

   - Añadimos paquetes de fuentes con:

     ```
     sudo reprepro -b /var/packages/ubuntu \
          includedsc gutsy (fichero .dsc)
     ```

   - Retiramos un paquete mediante:

     ```
     sudo reprepro -b /var/packages/ubuntu \
          remove gutsy (nombre)
     ```

A.2.4. Control de versiones

Todo desarrollo de software se puede beneficiar del uso de un sistema de control de versiones. A través de éste, podremos siempre acceder a todas las versiones realizadas en el pasado de cualquier fichero, y mantener múltiples copias de trabajo siempre sincronizadas. Son especialmente importantes en proyectos de software libre, donde puede haber potencialmente muchos participantes.

Además, el uso explícito de un sistema de control de versiones elimina una molestia a la hora de desarrollar paquetes Debian: la avalancha de ficheros que tendríamos que gestionar de otra forma, con ficheros `tar.gz` (más conocidos como *tarballs*), parches y demás que habrá que mantener versión a versión.

Como siempre, existe una gran variedad, pero en este caso elegí Subversion, descendiente a nivel conceptual del conocido CVS, por su excelente documentación, estabilidad y simplicidad. Adicionalmente, podremos hacer uso de un conjunto de guiones de ayuda desarrollados por la comunidad Debian, que nos ayudan a estandarizar un poco la estructura del repositorio.

> NOTA
>
> Para los preocupados por su espacio en disco: el enfoque seguido por Subversion, al igual que en la mayoría de los sistemas de control de versiones, es mucho más eficiente que lo que nosotros podríamos hacer simplemente guardando todas las versiones. Sólo guarda, comprimidos, los cambios de una versión a otra.

El proceso de preparación de nuestro propio repositorio de Subversion y una copia de trabajo local es sencillo:

1. Instalamos Subversion y el paquete de desarrollo de paquetes (valga la redunancia) bajo repositorios Subversion:

   ```
   sudo aptitude install subversion svn-buildpackage
   ```

2. Creamos el repositorio central en `~/.svnDebian`:

   ```
   svnadmin create .svnDebian
   ```

3. Ahora creamos una copia de trabajo, sobre la que crearemos y modificaremos ficheros, que luego enviaremos al repositorio central para que otras personas puedan propagar los cambios a sus propias copias de trabajo:

   ```
   svn checkout \
       file:///home/minombredeusuario/.svnDebian packages
   ```

Por lo pronto, no haremos nada más: la estructura de nuestro repositorio será generada de forma automática, y diferirá ligeramente de la estándar que el libro de Subversion describe.

A título de referencia para la fase de creación, he aquí algunas de las órdenes que podemos usar dentro de una copia de trabajo. Para más información, referirse al libro electrónico [14].

`svn add` ***`fichero`*** Marca un fichero o directorio para añadir al repositorio.

`svn rm` ***`fichero`*** Marca un fichero o directorio para eliminar del repositorio.

`svn mv` ***`fichero ruta`*** Mueve un fichero o directorio a otra ruta. El fichero o directorio no debe haber sufrido modificaciones desde su último envío.

`svn cp` ***`fichero ruta`*** Copia un fichero o directorio a otra ruta. El fichero o directorio no debe haber sufrido modificaciones desde su último envío.

`svn mkdir` ***`directorio`*** Crea un nuevo directorio y lo añade al repositorio.

`svn commit -m` ***`mensaje`*** Confirma todos los cambios hechos hasta el momento, y los envía al repositorio, registrando el envío con el mensaje que hayamos proporcionado. Tiene semántica transaccional: el envío se produce por completo, o no se produce en absoluto.

`svn status` Muestra el estado de todos los ficheros bajo el árbol actual, con una letra antes de su ruta. Algunas de las que pueden aparecer son:

- **?** No pertenece al repositorio (quizás habría que añadirlo con **`svn add`**).
- **A** El fichero va a ser añadido en el próximo envío.
- **D** El fichero va a ser borrado en el próximo envío.
- **M** El fichero ha sido modificado desde el último envío.

`svn export` ***`rutaDirOrigen rutaDestino`*** Exporta el directorio origen, que forma parte de una copia de trabajo, a la ruta destino, retirando todos los ficheros usados por Subversion. Muy útil cuando queremos redistribuir código, por ejemplo, y no la copia de trabajo entera.

A.2.5. Autenticación

Sólo nos queda una cosa más para tener el entorno completamente listo antes de desarrollar nuestro paquete Debian: preparar nuestra firma digital personal. Adicionalmente, la integraremos con nuestro repositorio local, y usaremos un agente de claves para evitar escribir una y otra vez su contraseña a la hora de construir el paquete.

¿Por qué querríamos usar una? No olvidemos que estamos haciendo un paquete que efectivamente va a ser instalado en el sistema final bajo la propiedad del superusuario. Una persona malintencionada podría manipular los contenidos del paquete y comprometer a los sistemas que lo instalen, añadiendo *rootkits* o puertas traseras entre nuestros ficheros.

Para un usuario muy centrado en seguridad, el mayor nivel de seguridad obtenible con un nivel razonable de esfuerzo, sin llegar a tener que analizar en profundidad el código, sería directamente descargar el código de alguna fuente fiable, y comprobar su integridad mediante algún algoritmo de *hashing*. Lo más común es encontrar *checksums* MD5, que aunque no criptográficamente seguros, son rápidos de calcular, y cumplen el propósito original de evitar descargas corruptas.

No llegaremos a esos extremos. En lugar de eso, aprovecharemos una de las aplicaciones de los algoritmos de cifrado asimétrico. En ellos, todo usuario tiene dos claves: la pública, que damos a todo el mundo, y la privada. Si ciframos algo con una clave, se puede descifrar con la otra. Así no nos hace falta transmitir la clave a la otra parte, como en los algoritmos tradicionales simétricos. En particular, si ciframos algo con nuestra clave privada, y el usuario lo descifra con nuestra clave pública, éste puede estar seguro que el paquete es auténtico.

Dado que cifrar todo un paquete puede ser potencialmente muy costoso, lo que se suele hacer es cifrar el resultado de alguna función de hash aplicada sobre el fichero, como SHA-1 o alguna variante criptográficamente más fuerte. De hecho, en los repositorios Debian, se va un paso más allá: sólo se firma el fichero central con

el listado de los paquetes disponibles, sus tamaños y sus *checksums* MD5, para tener el menor impacto posible en rendimiento. En nuestra configuración, se trata de `/var/packages/ubuntu/dists/gutsy/Release`, y su firma se halla en `/var/packages/ubuntu/dists/gutsy/Release.gpg`.

El proceso de instalación es algo más elaborado que en los anteriores casos, dado que se extiende a lo largo de unos cuantos ficheros de configuración:

1. Primero instalamos los paquetes que necesitamos: `gnupg` implementa la funcionalidad de cifrado en sí, `gnupg-agent` es el agente ocupado de almacenar temporalmente nuestra contraseña y `pinentry-gtk2` nos permite usar un diálogo gráfico para introducir la contraseña, en vez de usar la terminal:

   ```
   sudo aptitude install gnupg gnupg-agent pinentry-gtk2
   ```

2. Ahora generaremos nuestro par de claves, siguiendo las instrucciones que se nos irán indicando:

   ```
   gpg --gen-key
   ```

   El campo de correo electrónico es muy importante para el resto de pasos, y debe coincidir con el que usamos anteriormente en DEBEMAIL (ver sección A.2.2 (página 200)).

3. Añadiremos los ajustes necesarios al agente de GnuPG, en el fichero `~/.gnupg/gpg-agent.conf`:

   ```
   pinentry-program /usr/bin/pinentry-gtk-2
   no-grab
   default-cache-ttl 1800
   ```

   Así, sólo tendremos que introducir nuestra contraseña cada 30 minutos como mucho, y emplearemos una interfaz gráfica basada en GTK 2.0 (estilo GNOME).

4. Hemos de indicar a GnuPG que haga uso del agente para pedir la contraseña de la clave privada. Hay que descomentar la siguiente línea de `~/.gnupg/gpg.conf`:

   ```
   use-agent
   ```

5. Al instalar el paquete `gnupg-agent` se nos añadió el script necesario para su arranque en `/etc/X11/Xsession.d/90gpg-agent`. Hemos de reiniciar el servidor X, saliendo de nuestra sesión y pulsando `CTRL + ALT + Retroceso`.

6. Debemos decirle a *reprepro* que use dicha clave para firmar los paquetes. Añadiremos la siguiente línea con el email que usamos en la clave GPG a `/var/packages/ubuntu/conf/distributions`:

   ```
   SignWith: <tudireccion@decorreo>
   ```

7. Por último, hemos de exportar nuestra clave pública a un fichero, para poder pasársela a los demás, y añadirla a la lista de claves confiables de *apt*:

```
gpg -a --export tudireccion@decorreo > claveDebian.asc
sudo apt-key add claveDebian.asc
sudo cp claveDebian.asc /var/packages
```

A.2.6. Distribución a través de Internet

Instalación del servidor

Podemos aprovechar nuestro repositorio local para distribuir nuestros paquetes a través de la red, simplemente publicando el directorio `/var/packages` bajo un servidor HTTP o FTP. En particular, aquí veremos cómo hacerlo con el servidor Apache 2.

Evidentemente, hacerlo o no es completamente opcional. Los pasos a seguir son:

1. Si no tenemos una dirección IP estática, podemos en su lugar registrarnos en sitios como no-ip (http://www.no-ip.org) o DynDNS (http://dyndns.com), donde conseguiremos un nombre de dominio que actualizaremos periódicamente con nuestra IP.
2. Ahora hemos de asegurarnos de que el puerto 80 por TCP se halle accesible, cambiando los ajustes del cortafuegos y del encaminador en caso de que usemos NAT (Network Address Translation). Con Ubuntu recién instalado, el cortafuegos no tendrá ninguna regla definida, por lo que en principio no habría que hacer nada en cuanto a la configuración del servidor.
3. Instalamos el servidor Apache mediante:

   ```
   sudo aptitude install apache2
   ```

4. Crearemos el fichero `/etc/apache2/sites-available/repoDebian`, con el siguiente contenido, donde sustituiremos la dirección de nuestro servidor y el correo que empleamos en la firma GPG:

   ```
   <VirtualHost *:80>
           ServerAdmin tudireccion@decorreo

           DocumentRoot /var/packages
           ServerName direccion.delservidor.org:80
           ErrorLog /var/log/apache2/error.log

           LogLevel warn

           CustomLog /var/log/apache2/access.log combined
           ServerSignature On

           # Permite que la gente navegue por el directorio
   ```

```
        <Directory "/var/packages">
                Options Indexes FollowSymLinks MultiViews
                DirectoryIndex index.html
                AllowOverride Options
                Order allow,deny
                allow from all
        </Directory>

        # Oculta el directorio conf
        <Directory "/var/packages/*/conf">
                Order allow,deny
                Deny from all
                Satisfy all
        </Directory>

        # Oculta el directorio db
        <Directory "/var/packages/*/db">
                Order allow,deny
                Deny from all
                Satisfy all
        </Directory>
</VirtualHost>
```

5. Activaremos dicha página web y desactivaremos la página por defecto:

```
cd /etc/apache2/sites-enabled
sudo a2dissite default
sudo a2ensite repoDebian
```

6. Reiniciamos el servidor mediante:

```
sudo /etc/init.d/apache2 restart
```

7. ¡Listo! Habría que probar a cargar en un navegador la dirección `http://127.0.0.1/` y ver que todo coincide.

Instrucciones para un usuario de nuestro paquete

Ahora nos pondremos por un momento en la piel de un usuario de nuestro paquete. ¿Cómo podría tener nuestro paquete siempre a la última, con la seguridad de que es la versión genuina?

En este caso, vamos a intentar hacerlo todo a través de interfaces gráficas, más amigables para un usuario medio. Recomiendo añadirnos a nosotros mismos como un cliente más, para facilitar la depuración. Sólo tendríamos que sustituir nuestro nombre de host real con `127.0.0.1` en el resto de instrucciones, o mejor aún, añadir

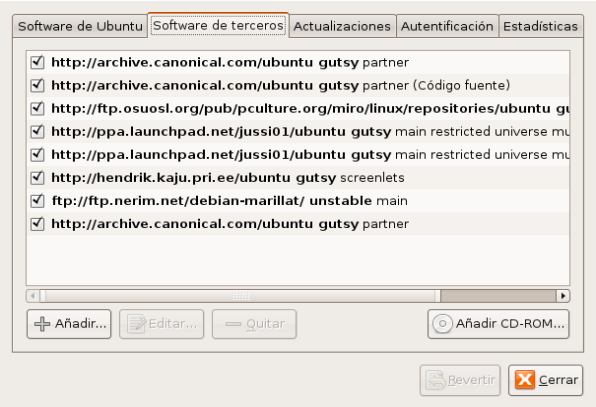


Figura A.1: Pestaña de Software de Terceros

la siguiente línea a `/etc/hosts`, reemplazando la que tuviera la misma dirección si hubiera alguna:

```
127.0.0.1 localhost direccion.delservidor.org
```

En primer lugar, el usuario debe de haber descargado el fichero `http://direccion.delservidor.org/claveDebian.asc` que exportamos antes, con nuestra clave pública.

Accederemos, a partir de la barra de programas en la parte superior de la pantalla, al elemento *Sistema* → *Administración* → *Orígenes de software*.

En el diálogo que aparece, pasamos a la pestaña *Software de terceros*, cuyo aspecto será similar al de la figura .

Pulsaremos en el botón **Añadir** una primera vez, y pegaremos la siguiente línea:

```
deb http://tunombre.deservidor.org/ubuntu gutsy main
```

Volvemos a pulsarlo, y ahora introducimos ésta, correspondiente a un repositorio de paquetes de código fuente (de ahí el *deb-src*):

```
deb-src http://tunombre.deservidor.org/ubuntu gutsy main
```

Cambiamos a la pestaña *Autentificación*, que será parecida a la de la figura , y pulsamos **Importar clave**. Seleccionaremos el fichero `claveDebian.asc` antes descargado.

Guardamos los cambios realizados pulsando **Cerrar**, y finalmente pulsamos el botón **Recargar** de la barra de herramientas de la ventana principal. Cuando se cierre

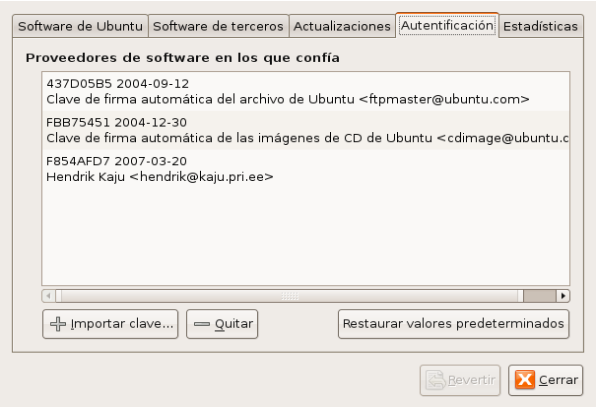


Figura A.2: Pestaña de Autentificación

automáticamente el diálogo que aparece, ya podremos usar como siempre *Synaptic*, empleando el botón **`Buscar`** para hallar el paquete que deseamos, marcarlo con un doble clic e instalarlo mediante **`Aplicar`**. Además, a partir de ahora, el sistema *apt* se ocupará de monitorizar dicho repositorio y mantener actualizado el sistema.

A.3. Creación y mantenimiento del paquete

En este capítulo, daré un ejemplo de un paquete razonablemente sencillo, pero completo: un emulador de SNES, conocido como ZSNES. Veremos todas las fases, desde que nos descargamos el código fuente hasta que tenemos el paquete instalado en nuestro sistema y funcionando. Existen muchas guías de creación de paquetes, pero en mi opinión la información se halla bastante fragmentada. De todas formas, en el wiki de Ubuntu hay una excelente introducción acerca del tema [58], y también hay una guía por parte de la comunidad Debian [49], aunque en mi opinión la de Ubuntu está más actualizada.

Para simplificar, describiré el proceso de forma secuencial. Sin embargo, lo normal es que sea iterativo, teniendo muchas revisiones intermedias del paquete hasta dejarlo listo para su distribución. Se ven aspectos más avanzados en el siguiente capítulo.

A.3.1. Adaptaciones previas al uso de Subversion

Creación de un esqueleto

Antes de poder introducir los ficheros fuente de nuestro paquete en el repositorio Subversion que previamente preparamos, hemos de crear una primera versión de nuestro paquete.

Tras descargar el código fuente de la página oficial (http://www.zsnes.com/index.php?page=files) a /tmp/packages/zsnes151src.tar.bz2, crearemos el esqueleto básico del paquete mediante **dh_make**, una de las muchas herramientas del paquete *debhelper* de ayuda. Ejecutaremos las siguientes órdenes en una terminal dentro del directorio /tmp/packages:

```
tar -xjf zsnes151src.tar.bz2
mv zsnes_1_51 zsnes-1.510
cd zsnes-1.510
dh_make -e tudireccion@decorreo -c GPL -s --createorig
```

La carpeta que hemos creado (zsnes-1.510) obedece al convenio seguido por Debian nombrepaquete-versionUpstream, donde la versión del programa original se entiende como una serie de números separados por puntos: así, zsnes-1.510 es más reciente que zsnes-1.6, y menos que zsnes-1.600.

Por otro lado, las opciones pasadas a **dh_make** son:

-e tudireccion@decorreo Especifica nuestra dirección email como desarrollador del paquete. Se usa en el registro de cambios (de ahora en adelante el *Changelog*), y para realizar firmas digitales.

-c GPL Indica que el código original sigue la General Public License. Otras opciones incluyen la LGPL, BSD o la licencia artística. En otro caso, se nos dejará un hueco (posteriormente veremos dónde) para que lo rellenemos con el texto de la licencia en cuestión.

-s Existen varios tipos de paquete Debian: de un solo binario, de varios, bibliotecas, o paquetes que emplean CDBS (el resto usan únicamente *debhelper*). Aquí hemos decidido hacer un paquete de un solo binario, mediante debhelper. Después veremos también cómo hacer un paquete con CDBS.

–createorig Creamos en el directorio padre un fichero zsnes_1.510.orig.tar.gz con el código fuente original, para poder comparar con la versión que usemos al construir el paquete y volcar las diferencias a un fichero diff.

Edición

Ya tenemos el esqueleto del paquete. Todos los ficheros específicos de él se hallan bajo el directorio /tmp/packages/zsnes-1.510/debian. Si examinamos dicho directorio, veremos que hay un gran número de ficheros. No utilizaremos los ejemplos incluidos, indicados por la extensión .ex, así que los retiraremos, junto con el fichero README.Debian, dado que no hay nada especial acerca de nuestro paquete:

```
rm debian/*.ex,EX debian/README.Debian
```

Iremos rellenando cada fichero de control en `debian` con los datos necesarios. Iremos detallando su sintaxis y semántica a lo largo de esta sección.

changelog Éste es el registro de cambios de nuestro paquete. Aquí iremos indicando los cambios realizados a lo largo de cada versión del paquete, no del software original. Este fichero es el que nuestros usuarios leerán para ver qué hay de nuevo en cada versión del paquete.

Escribiremos nuestra primera entrada:

```
zsnes (1.510-0ubuntu1) gutsy; urgency=low

  * Versión inicial del paquete

 -- Antonio Garcia <nyoescape@gmail.com>  Fri, 19 Feb 2008 13:49:12 +0100
```

Vemos cómo la versión actual del paquete junto con su última fecha y autor del cambio se hallan codificados en el registro. También se tiene en cuenta la distribución (en nuestro caso *gutsy*, de Ubuntu), y la urgencia del cambio (por lo general baja, a menos que se trata de una vulnerabilidad de seguridad o algo del estilo).

Para una misma versión del software original, tendremos distintas versiones del paquete, separadas del número de versión original por un guión, como vemos aquí. En particular, los paquetes de Ubuntu [19] usan el esquema `-XubuntuY`, indicando que se trata de la Y-ésima versión del paquete de Ubuntu originado de la X-ésima versión del paquete Debian (0 si no proviene de un paquete Debian). Los números de versión de paquete comienzan por 1.

> NOTA
>
> La razón de este esquema de versionado es para permitir una fácil integración con los paquetes Debian. Normalmente, la política de Ubuntu es sólo crear nuevos paquetes o versiones de éstos si el paquete Debian está anticuado o tiene algún problema. Así, si los de Debian sacan una nueva versión, como la `-3`, a partir de `-2ubuntu3`, se reflejará dicha información de forma correcta.

Así, para la próxima versión del paquete, sólo tendremos que añadir la entrada en cuestión al registro, y nuestros guiones de ayuda harán el resto del trabajo.

> NOTA
>
> Mucho cuidado con el formato del registro, es muy rígido. El espaciado debe ser exactamente el mismo que en el ejemplo, como los dos espacios entre la dirección de correo y la fecha, o el espacio inicial al inicio de la misma línea.

compat Para este fichero no hay que hacer nada: sólo contiene un número entero, indicando qué versión del paquete *debhelper* estamos usando.

control Este fichero es muy importante: describe todos los paquetes que estamos definiendo y enuncia sus dependencias. El formato es también bastante rígido, pero muy simple. Utiliza una serie de campos delimitados por ':' y saltos de línea.

Por supuesto, no existe ninguna receta mágica que nos diga las dependencias de un programa cualquiera. Para ello, normalmente tendremos que examinar la documentación del desarrollador original, y/o el guión de compilación que utilice: como aquí usan las *autotools*, podríamos consultar `src/configure.in`. Una buena referencia respecto a las *autotools* es el Autobook [53].

Por suerte, los desarrolladores de ZSNES han incluido dichas dependencias en `docs/-install.txt`, con lo que no tendremos que ir buscando en los guiones de compilación.

El fichero que usaremos será éste:

```
Source: zsnes
Section: games
Priority: optional
Maintainer: Antonio Garcia <nyoescape@gmail.com>
Build-Depends: cdbs, debhelper (>= 4.1.0), autotools-dev, fakeroot,
  desktop-file-utils, g++ (>= 4), libsdl1.2-dev, nasm (>= 0.98),
  zlib1g-dev (>= 1.2.3), libpng12-dev (>= 1.2), libncurses5-dev,
  libgl1-mesa-dev
Standards-Version: 3.7.2

Package: zsnes
Architecture: i386
Depends: $shlibs:Depends
Description: Emulador de Super Nintendo
 Emulador de la consola Super Nintendo con más funciones
 disponibles. Permite guardar y cargar estados, grabar
 demostraciones, y aplicar diversos filtros. Tiene una
 compatiblidad inmejorable.
```

NOTA

En éste y en cualquier otro fichero de control de Debian, no debemos olvidar poner un salto de línea justo al final del fichero.

Examinando el fichero de campo a campo, tenemos:

Source: zsnes Indica que el paquete fuente del que derivan todos se llama "zsnes".

Section: games Por la política de Debian [36], todo paquete se halla en alguna sección de las disponibles. Así indicamos qué tipo de aplicación es: un juego, un editor, etc.

Priority: optional Indica la importancia del paquete: desde imprescindibles (*required*), pasando por importantes (*important*), estándar (*standard*), opcionales (*optional*), y extra (tienen conflicto con alguno de más prioridad).

Maintainer: Antonio Garcia <nyoescape@gmail.com> Nombre y dirección de contacto del desarrollador del paquete.

Build-Depends: ... Paquetes requeridos para poder compilar este paquete. Incluye las herramientas para paquetes Debian y las dependencias del propio programa.

Standards-Version: 3.7.2 Versión de la política de Debian que este documento sigue. Realmente se halla compuesta por varios documentos, todos situados bajo el directorio `/usr/share/doc/debian-policy`.

Package: zsnes Nombre de uno de los paquetes binarios generados a partir del fuente. Aquí sólo hay uno y tiene el mismo nombre.

Architecture: i386 Arquitectura a la que va dirigida el paquete. Existe una gran variedad de valores, pero nos interesan sobre todo *i386* (la IA-32 habitual), *source* (código fuente) y *all* (código sin una arquitectura definida, como programas Java, o guiones de algún lenguaje interpretado como Perl o Python).

Depends Paquetes requeridos para que éste se instale y funcione correctamente. La variable `${shlib:Depends}` incluye las dependencias deducidas de forma automática en cuanto a bibliotecas dinámicas se refiere.

Description Incluye una descripción corta de una sola línea y otra más larga de varias líneas del contenido del paquete. Al igual que siempre, su formato es muy rígido: toda línea de la descripción larga comienza por un espacio, y líneas vacías únicamente añaden un punto ('.'). El campo termina tras la primera línea sin dicho espacio inicial.

NOTA

Mucho cuidado con los acentos y demás en el nombre del desarrollador del paquete y otros campos: podrían causar problemas en el interior de la jaula *chroot*, que sólo tiene soporte para los caracteres ASCII de 7 bits.

copyright Contiene la información relativa a la licencia del paquete y del programa original, junto con datos acerca de los autores originales, su copyright y de dónde descargamos el código fuente.

En este caso sólo tenemos que rellenar sin más los campos. No se fuerza ningún formato particular sobre el fichero. Es importante sustituir Üpstream Author(s)"por Üpstream Authorsz fijarnos en la información en `docs/authors.txt` del código fuente, o los verificadores de paquetes que veremos después darán avisos al respecto.

dirs En este fichero listamos los directorios en que vamos a instalar algún fichero. Si no lo listamos aquí, dicho directorio no va a hallarse disponible durante la construcción del paquete, así que hay que tener cuidado. Las rutas deben de seguir el Filesystem Hierarchy Standard (FHS), disponible a través de la orden **`man hier`** desde cualquier terminal.

Algunas rutas importantes y sus contenidos son:

`/bin` Ejecutables usados en modo monousuario. Normalmente realizan tareas de mantenimiento a bajo nivel, entre otras cosas. Instalados a través de paquetes Debian.

`/boot` Configuración de GRUB, ficheros de imagen de los *kernels* disponibles, etc.

`/dev` Árbol de directorios donde cada dispositivo conectado al sistema es un fichero.

`/etc` Ficheros de configuración global (para todos los usuarios).

`/home` Directorios de casa de cada usuario, con espacio para cada uno de ellos.

`/mnt` Dispositivos externos montados temporalmente: particiones de Windows, CD, DVD, pendrives, etc.

`/proc` Árbol de directorios con información del *kernel* en cada fichero: procesos en ejecución, dispositivos disponibles, etc.

`/root` Directorio de casa del superusuario.

`/sbin` Ejecutables para uso del superusuario.

`/usr` Datos, programas y bibliotecas compartidos por todos los usuarios.

`/usr/bin` Ejecutables para todos los usuarios, instalados a través de paquetes Debian.

`/usr/lib` Bibliotecas para todos los usuarios, instalados a través de paquetes Debian.

`/usr/local` Similar a `/usr`, pero para uso del administrador.

`/usr/share` Datos compartidos por todos los usuarios.

`/usr/share/man` Páginas de *man* disponibles. Toda página se halla dentro de una sección. En particular, la de *zsnes* estaría en la 1, tras ver las instrucciones disponibles a través de la orden **`man man`**.

Dado que tenemos que instalar el ejecutable para todos los usuarios y una página *man*, nuestro fichero `dirs` contendrá:

```
usr/bin
usr/share/man/man1
```

> NOTA
>
> En este fichero, las rutas *no* incluyen una barra inicial, como suelen hacer. Para ser más exactos, son rutas a crear dentro del área temporal de construcción del directorio `debian/zsnes`, donde colocaremos todos los ficheros tal y como se descomprimirán después bajo el directorio raíz, `/`.

`rules` Aquí está el fichero más importante de todos. Es el que decide qué hay que hacer exactamente para compilar e instalar el paquete completo: documentación, binarios, guiones y ficheros de datos.

De todas formas, en términos generales, no es más que un *makefile*, si bien uno que puede hacerse muy complejo: el objetivo *build* compila, *install* instala dentro del área de construcción del paquete, y *clean* retira los ficheros generados durante la construcción. Esta última se halla bajo la ruta relativa `debian/zsnes` respecto del directorio principal del paquete.

Dado que escribir una y otra vez un *makefile* completo para muchas aplicaciones parecidas era una pérdida de tiempo, se han desarrollado diversos paquetes que factorizan cierta funcionalidad común, como instalar páginas *man*, tipos MIME, entradas de menú, y cosas del estilo: son los guiones del paquete `debhelper`. Prácticamente nadie hoy en día desarrolla sus paquetes sin estos guiones.

Algunos desarrolladores han decidido ir un paso más allá, y factorizar reglas para perfiles completos de aplicaciones. Así, si sabemos que se trata de una aplicación desarrollada a través de las autotools, sólo tendremos que aplicar dicho perfil, añadiendo las opciones oportunas que pasar a `configure`, por ejemplo. Esto es el Common Debian Build System (CDBS) [15], que usaremos en esta guía. Existen perfiles para aplicaciones Python, Perl, GNOME, KDE, Java (basadas en Ant), o incluso para aquellas con un simple *makefile*.

Por supuesto, para paquetes complicados, este sistema se queda corto, pero son minoría comparados con los demás. Además, no es sólo cuestión de simplicidad: factorizando la mayor proporción posible de reglas, blindaremos nuestro paquete ante cambios en la política de Debian en el futuro.

De todas formas, en general, la comunidad de desarrolladores se halla muy dividida entre usar o no CDBS: aunque factoriza mucha complejidad, resulta difícil de comprender y aprovechar en casos difíciles, a menos que seamos capaces de leer complejos ficheros *makefile* por nosotros mismos, dado que no existe mucha documentación detallada al respecto: para CDBS, el código es la mejor documentación.

Dado que resulta imposible entender bien CDBS si no se comprende antes el sistema tradicional, en esta sección explicaremos las dos alternativas. Comenzaremos por el sistema "tradicionalçon los guiones de debhelper, y luego veremos cómo CDBS factoriza la mayor parte de estas reglas.

Reglas con *debhelper* Partiendo del esqueleto que automáticamente nos ha creado **`dh_make`**, lo retocamos para este paquete en particular. Vamos a ver qué tal ha quedado, y luego explicaremos qué partes exactamente hemos cambiado, y por qué:

Listado A.1: Reglas de un paquete Debian creado con debhelper

```
#!/usr/bin/make -f
# -*- makefile -*-

# This file was originally written by Joey Hess and Craig Small. As a
# special exception, when this file is copied by dh-make into a dh-make
# output file, you may use that output file without restriction. This
# special exception was added by Craig Small in version 0.37 of dh-make.

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

# El código se halla en un subdirectorio, no en la raíz
SRCDIR = src
# Opciones a pasar a configure (--enable-release activa optimizaciones)
CONFIGURE_FLAGS = --disable-cpucheck --enable-release --with-x --with-opengl
# Opciones a usar en el compilador
CFLAGS = -Wall -g

ifneq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
        CFLAGS += -O0
else
        CFLAGS += -O2
endif

configure: configure-stamp
configure-stamp:
        dh_testdir
        touch configure-stamp
```

```
build: build-stamp

build-stamp: configure-stamp
        dh_testdir

        cd $(SRCDIR) && \
          force_arch=i586 ./configure $(CONFIGURE_FLAGS) --prefix=/usr
        $(MAKE) -C $(SRCDIR)

        touch $@

clean:
        dh_testdir
        dh_testroot
        rm -f build-stamp configure-stamp

        # Limpiamos también las herramientas internas usadas por ZSNES en su
        # compilación
        -$(MAKE) -C $(SRCDIR) clean tclean
        # Borramos los ficheros temporales generados por la compilación
        $(RM) $(SRCDIR)/tools/depbuild $(SRCDIR)/config.{log,status,h} $(SRCDIR)/Makefile

        dh_clean

install : build
        dh_testdir
        dh_testroot
        dh_clean -k
        dh_installdirs

        $(MAKE) -C $(SRCDIR) install DESTDIR=$(CURDIR)/debian/zsnes

# Build architecture -independent files here.
binary-indep: build install
# We have nothing to do by default.

# Build architecture -dependent files here.
binary-arch: build install
        dh_testdir
        dh_testroot
        dh_installchangelogs
        dh_installdocs
        dh_installexamples
#       dh_install
#       dh_installmenu
#       dh_installdebconf
```

```
#       dh_installlogrotate
#       dh_installemacsen
#       dh_installpam
#       dh_installmime
#       dh_python
#       dh_installinit
#       dh_installcron
#       dh_installinfo
        dh_installman $(SRCDIR)/linux/zsnes.1
        dh_link
        dh_strip
        dh_compress
        dh_fixperms
#       dh_perl
#       dh_makeshlibs
        dh_installdeb
        dh_shlibdeps
        dh_gencontrol
        dh_md5sums
        dh_builddeb

binary: binary-indep binary-arch
.PHONY: build clean binary-indep binary-arch binary install configure
```

Aunque es bastante largo, conceptualmente es sencillo, gracias al uso de *debhelper*. Un par de cosas a destacar:

1. Dado que nuestro código fuente se halla bajo un subdirectorio y no en el directorio raíz del paquete, añadimos una variable `SRCDIR`, cuyo valor tendremos en cuenta para realizar un cambio de directorio antes de cada orden de compilación.
2. Por otro lado, `CURDIR` contiene la ruta del directorio raíz actual desde el cual se está construyendo el paquete. La ruta `$(CURDIR)/debian/zsnes` contiene un árbol de directorios que sigue el FHS, y se corresponde con los ficheros contenidos en el paquete `zsnes`.
3. Bajo el objetivo de compilación `build-stamp` añadimos las órdenes requeridas para compilar: invocamos al guión `configure` con las opciones necesarias e iniciamos la compilación. La opción `-C` pasada a **`make`** hace el cambio de directorio antes de comenzar, justo como **`cd`** hace para las demás. Hay que hacerlo para cada orden y no al principio debido al hecho de que tras cada orden volvemos al directorio original, al restaurarse el estado anterior del shell.
4. Repetimos el cambio en la invocación a **`make`** para los otros objetivos `clean` e `install`. Puede verse cómo se pasa la variable de entorno `DESTDIR` con la ruta al área de construcción para la instalación: evidentemente, el *makefile* debe

de estar hecho para tener esto en cuenta. Tenemos la suerte para este paquete de que ya sea así: de lo contrario, tendríamos que adaptar dicho fichero, ¡y posiblemente el resto del programa!

5. Por último, en el objetivo `binary-arch` tenemos una serie de llamadas a distintos guiones de *debhelper*. Comentaremos y descomentaremos según nos haga falta: así, por ejemplo, un programa escrito en Perl no necesita **dh_link** ni **dh_strip**, al no generar ejecutables.

   Hemos añadido un argumento a **dh_installman** con la página *man* que queremos que se instale. Al igual que con todo lo demás, si no hubiera una, tendríamos que crearla nosotros. Lo más usual en este caso es escribir un fichero SGML o XML DocBook y transformarlo a *nroff* (el formato de las páginas *man*) mediante *docbook-to-man* o una hoja de estilos XSLT, por ejemplo. Podríamos partir del ejemplo creado antes por **dh_make**>, `zsnes.sgml.ex`.

Un detalle importante: la mayoría de los guiones suponen que los ficheros bajo nuestro directorio `debian` siguen una serie de convenciones. Por ejemplo, **dh_installdocs** supone que existe algún fichero `debian/zsnes.docs` o `debian/docs` que liste la documentación a instalar. En este caso, aprovechamos la documentación que ya trae ZSNES, con lo que tendríamos esto en `debian/docs`:

```
docs/srcinfo.txt
docs/README.SVN
docs/opengl.txt
docs/stdards.txt
docs/authors.txt
docs/todo.txt
docs/install.txt
docs/thanks.txt
docs/support.txt
docs/README.LINUX
docs/readme.txt/about.txt
docs/readme.txt/faq.txt
docs/readme.txt/history.txt
docs/readme.txt/gui.txt
docs/readme.txt/advanced.txt
docs/readme.txt/index.txt
docs/readme.txt/games.txt
docs/readme.txt/netplay.txt
docs/readme.txt/readme.txt
docs/readme.txt/support.txt
docs/readme.htm/styles/release.css
docs/readme.htm/styles/print.css
docs/readme.htm/styles/jipcy.css
```

```
docs/readme.htm/styles/radio.css
docs/readme.htm/styles/corner.png
docs/readme.htm/styles/plaintxt.css
docs/readme.htm/styles/shared.css
docs/readme.htm/images/zsneslogo.png
docs/readme.htm/images/netplay.png
docs/readme.htm/images/quick.png
docs/readme.htm/images/saveslot.png
docs/readme.htm/images/cheat.png
docs/readme.htm/images/gui.png
docs/readme.htm/images/config.png
docs/readme.htm/images/game.png
docs/readme.htm/images/f1_menu.png
docs/readme.htm/images/misc.png
docs/readme.htm/netplay.htm
docs/readme.htm/about.htm
docs/readme.htm/games.htm
docs/readme.htm/advanced.htm
docs/readme.htm/gui.htm
docs/readme.htm/support.htm
docs/readme.htm/readme.htm
docs/readme.htm/history.htm
docs/readme.htm/license.htm
docs/readme.htm/faq.htm
docs/readme.htm/index.htm
docs/readme.1st
```

NOTA

Hemos usado ya otro fichero más del mismo estilo: `dirs` es realmente el fichero que **`dh_installdirs`** utiliza, por ejemplo.

Reglas con CDBS Ahora que ya sabemos cuál es la estructura real de un fichero de reglas, lo reescribiremos empleando CDBS:

Listado A.2: Reglas de un paquete Debain creadas utilizando CDBS

```
#!/usr/bin/make -f
# -*- makefile -*-

DEB_SRCDIR = src

include /usr/share/cdbs/1/rules/debhelper.mk
```

```
include /usr/share/cdbs/1/class/autotools.mk

DEB_CONFIGURE_SCRIPT_ENV += force_arch=i586
DEB_CONFIGURE_EXTRA_FLAGS = --disable-cpucheck --with-x \
  --enable-release --with-opengl
DEB_INSTALL_MANPAGES_zsnes += $(DEB_SRCDIR)/linux/zsnes.1
```

Sorprendentemente, esto es todo: ocho líneas. Los dos **`include`** se ocupan de importar los conjuntos de reglas de apoyo para el uso interno de *debhelper* e implementar el soporte para el perfil de las autotools, respectivamente.

A continuación, pasamos las mismas opciones a `configure` que antes: pedimos que compile para Pentium o superior, que emplee aceleración 3D y un interfaz gráfico, optimice algo más de lo normal (`--enable-release`) y no intente autodetectar nuestra CPU. Finalmente, indicamos que instale la página man `src/linux/zsnes.1` que incluye ZSNES.

Usaremos esta versión de `debian/rules` para realizar el resto del documento, aprovechando algunas funcionalidades adicionales que aporta. Sin embargo, internamente, es exactamente lo mismo de antes.

Construcción preliminar

Con todo listo, ya podemos construir el paquete. Situándonos en el directorio principal del paquete, `/tmp/packages/zsnes-1.510`, ejecutaremos:

```
dpkg-buildpackage -rfakeroot
```

Tras un cierto tiempo, nos preguntará la contraseña de nuestra clave privada para firmar el paquete de forma automática. Poco después, tendremos en `/tmp/packages` nuestra primera versión del paquete Debian, `zsnes_1.510-0ubuntu1_i386.deb`, junto con un fichero `.dsc` que describe el paquete fuente que también hemos construido, y un `diff.gz` con las diferencias respecto a las fuentes originales.

Inyección en el repositorio

Con nuestra primera versión del paquete lista, sólo nos queda inyectar el paquete en el repositorio Subversion. Nos situaremos en `~/packages` y ejecutaremos:

```
svn-inject -c2 -o /tmp/packages/zsnes_1.510-0ubuntu1.dsc \
  file:///home/tunombredeusuario/.svnDebian
```

Tras un cierto tiempo, ya tendremos enviado al repositorio central nuestro paquete, y nuestra copia de trabajo habrá sido creada, con lo que no necesitaremos más `/tmp/-packages`.

La opción `-o` evita que se guarde el código fuente en el repositorio, usando únicamente archivos `tar.gz` en el subdirectorio `tarballs` del directorio principal del repositorio, que no se hallará bajo control de versiones.

El repositorio creado tiene la siguiente estructura:

~/packages/tarballs Contiene los ficheros `tar.gz` con las fuentes originales de nuestros paquetes.

~/packages/zsnes/build-area Se trata del directorio destino en el que se depositarán todos los paquetes y demás ficheros que vayamos produciendo.

~/packages/zsnes/branches Almacena las distintas ramas de desarrollo. Normalmente contendría las distintas versiones del código original, pero al usar tarballs, es prácticamente inútil en nuestro caso.

~/packages/zsnes/tags Permite asociar números de versión con determinadas revisiones del repositorio. Posteriormente veremos cómo se usa.

~/packages/zsnes/trunk Aquí se almacena la versión actual del paquete. Sólo almacenamos el directorio `debian`, para ahorrar la complejidad y tiempo de descarga necesario de otra forma.

A.3.2. Preparación de una primera versión

Construcción definitiva

Es el momento de reconstruir el paquete, pero esta vez haciendo uso de la jaula *chroot*, para asegurar que efectivamente hemos listado todas las dependencias correctamente.

Dado que la orden es bastante larga, lo mejor es añadir un alias a dicha orden, o mejor aún, declarar una función de *dash* (la versión reducida del shell *bash* de Ubuntu) que haga dicha tarea, en `~/.bashrc`. Añadiremos estas líneas:

```
function svn-b ()
  buildarea="`pwd`/../build-area";
  sudo cowbuilder --update
  svn-buildpackage  \
    --svn-builder="pdebuild --auto-debsign --buildresult $buildarea" \
    --svn-postbuild="rm $buildarea/*_source.changes";
```

```
function svn-tag ()
  svn-buildpackage --svn-only-tag
```

Nos situaremos en `~/packages/zsnes/trunk` y ejecutaremos:

```
svn-b
```

Tras un tiempo prudencial, e introducir nuestras contraseñas de GPG y superusuario, tendremos una primera versión del paquete, comprobada dentro de una jaula *chroot*.

Verificación

Sin embargo, no basta con que se compile e instale correctamente. Además, el paquete debe de cumplir todas las políticas de Debian, como pertenecer a un determinado conjunto de secciones, tener todos sus ficheros de control bien escritos, respetar el estándar FHS que antes mencionamos, etc.

De hecho, si queremos que nos lleguen a aceptar cualquier paquete en un repositorio Debian oficial, como la sección *universe* o *multiverse* de Ubuntu, o la sección *unstable* o *testing* de Debian, lo mejor que podemos hacer para evitar hacer perder tiempo a nuestro supervisor o supervisores (que serán los que efectivamente suban al repositorio el paquete las primeras veces) es pasar antes nuestro paquete por los dos verificadores disponibles: Linda y Lintian, y retirar todos los avisos.

> NOTA
>
> Según parece, Linda, escrito en Python, es más reciente que Lintian, y también más rápido. Por otro lado, Lintian está escrito en Perl. Aparte de eso, no hay muchas diferencias: simplemente ejecutaremos ambos por seguridad, ya que tampoco supone un esfuerzo adicional considerable.

Lanzaremos los verificadores sobre el `.deb`, que se hallará en `~/packages/zsnes/build-area`, pidiendo explicaciones de los avisos y errores con `-i`:

```
linda -i ~/packages/zsnes/build-area/zsnes_1.510-0ubuntu1_i386.deb
lintian -i ~/packages/zsnes/build-area/zsnes_1.510-0ubuntu1_i386.deb
```

Según parece, no todo está bien en nuestro paquete:

```
W: zsnes: extra-license-file usr/share/doc/zsnes/license.htm
N:
N:   All license information should be collected in the debian/copyright
N:   file. This usually makes it unnecessary for the package to install
N:   this information in other places as well.
N:
N:   Refer to Policy Manual, section 12.5 for details.
N:
E: zsnes: FSSTND-dir-in-usr usr/man/
N:
N:   As of policy version 3.0.0.0, Debian no longer follows the FSSTND.
N:
N:   Instead, the Filesystem Hierarchy Standard (FHS), version 2.3, is
N:   used. You can find it in /usr/share/doc/debian-policy/fhs/ .
N:
E: zsnes; Manual page zsnes.1.gz installed into /usr/man.
 The manual page shown is installed into the legacy location of
 /usr/man, where they should be installed into /usr/share/man.
E: zsnes; FSSTND directory /usr/man in /usr found.
 As of policy version 3.0.0.0, Debian no longer follows the FSSTND.
 Instead, the Filesystem Hierarchy Standard (FHS), version 2.1, is
 used. You can find it in /usr/share/doc/debian-policy/fhs/ .
E: zsnes; FSSTND directory /usr/man/man1 in /usr found.
W: zsnes; File /usr/share/doc/zsnes/license.htm is considered to be an
 extra license file.  The file shown above is considered to be another
 license file, where as the license for a package should be contained
 in the copyright file, which should be installed into
 /usr/share/doc/<pkg>.
```

Analizando esta salida, vemos que hay cuatro problemas con nuestro paquete según Linda, y dos según Lintian. El primer problema es fácil de tratar: hemos incluido un fichero de licencia con la documentación (`extra-license-file`), que no debería estar, ya que el archivo `copyright` debería contener toda la información. Basta en este caso con retirar la línea problemática de `debian/docs`.

El segundo problema es más complejo: el directorio `/usr/man` no es parte del estándar actual, FHS, sino de uno más antiguo, el FSSTND (FileSystem Standard). En principio, nuestras reglas están bien escritas. Veamos si efectivamente nuestro paquete pone la página man en su sitio:

```
~/packages/zsnes/trunk$ dpkg \
  -c ../build-area/zsnes_1.510-0ubuntu1_i386.deb
```

Sí que lo hace: `/usr/share/man/man1/zsnes.1.gz` aparece como debería. Sin embargo, también aparece `/usr/man/man1/zsnes.1.gz`. Así, hay algo fuera de nuestras reglas que está instalando en dicho sitio la página *man*.

Vamos a descomprimir en un directorio temporal el código fuente original, y echar un vistazo con la inestimable ayuda de *grep*, empleando la opción `-R` para hacer una búsqueda recursiva por el árbol de directorios y `-l` para solamente listar los ficheros con coincidencia y no el contenido que sigue el patrón:

```
cd /tmp
tar -xzf ~/packages/tarballs/zsnes_1.510.orig.tar.gz
cd zsnes-1.510.orig
grep -Rl zsnes.1 *
docs/srcinfo.txt
docs/readme.txt/readme.txt
docs/readme.htm/readme.htm
src/Makefile.in
```

Ya lo hemos encontrado: descartando los tres ficheros que forman parte de la documentación, sólo queda `src/Makefile.in`. Tiene que ser el culpable, ya que este fichero es usado por el guión `configure` para crear el *makefile* con el que hacer la compilación e instalar el programa.

Veremos exactamente dónde está el problema, empleando la opción `-C` para ver un contexto de un determinado número de líneas alrededor de las apariciones:

```
grep -C4 zsnes.1 src/Makefile.in
install:
        @INSTALL@ -d -m 0755 $(DESTDIR)/@prefix@/bin
        @INSTALL@ -m 0755 @ZSNESEXE@ $(DESTDIR)/@prefix@/bin
        @INSTALL@ -d -m 0755 $(DESTDIR)/@prefix@/man/man1
        @INSTALL@ -m 0644 linux/zsnes.1 $(DESTDIR)/@prefix@/man/man1
uninstall:
        rm -f @prefix@/bin/$(notdir @ZSNESEXE@) @prefix@/man/man1/zsnes.1

clean:
        rm -f $(Z_OBJS) $(PSR) $(PSR_H) @ZSNESEXE@
tclean:
```

Ya hemos encontrado las dos líneas problemáticas, al final del objetivo `install`: crean el directorio e instalan la página *man*. Sólo hay que quitarlas. Sin embargo, hay un problema: no podemos tocar el código directamente, ya que no lo tenemos bajo control de versiones. Es más: no se considera buena práctica, ya que en futuras versiones los desarrolladores originales podrían arreglar ese problema, y se haría difícil ver quién ha hecho qué tras hacer unos cuantos cambios más.

Por eso, se considera buena práctica emplear un sistema de parches para hacer este tipo de modificaciones. Algunas alternativas [57] incluyen el más antiguo, *dpatch*, el

más reciente (y aparentemente superior, según los desarrolladores de SUSE) *quilt* y el que usaremos aquí, *simple-patchsys*, que como su nombre indica, está más inclinado hacia ser sencillo que ser potente. Sin embargo, nos basta para casos sencillos como éste.

Para usarlo, añadimos la línea que instala soporte para él en `debian/rules`, justo después del primer `include`:

```
include /usr/share/cdbs/1/rules/simple-patchsys.mk
```

Pasaremos a crear el parche en sí. Para ello, tenemos que obtener el árbol de fuentes tal y como lo vería `debuild` antes de compilar, y no como lo tenemos ahora. Para ello, usaremos **`svn-buildpackage`**, no sin antes enviar nuestros otros cambios al repositorio (de otra forma, no nos dejará seguir):

```
~/packages/zsnes/trunk$ svn commit -m \
  "Integrado simple-patchsys del CDBS y corregido documentación"
Enviando       trunk/debian/docs
Enviando       trunk/debian/rules
Transmitiendo contenido de archivos ..
Commit de la revisión 6.
~/packages/zsnes/trunk$ svn-buildpackage --svn-export
buildArea: /home/antonio/packages/zsnes/build-area
[...]
I: mergeWithUpstream property set, looking for upstream source tarball...
[...]
Exportación completa.
rm -rf /home/antonio/packages/zsnes/build-area/tmp-0.00194501814349124
Build directory exported to /home/antonio/packages/zsnes/build-area/zsnes-1.510
```

Iremos al directorio exportado de construcción, que contendrá todos los ficheros necesarios, y generaremos el parche usando **`cdbs-edit-patch`**:

```
~/packages/zsnes/trunk$ cd ../build-area/zsnes-1.510/
~/packages/zsnes/build-area/zsnes-1.510$ cdbs-edit-patch 01-man-fhs.patch
```

El programa nos indicará que nos hallamos ahora en un subshell, y que hagamos las modificaciones necesarias. Usando un editor cualquiera, retiraremos dichas líneas de `src/Makefile.in`. Hecho esto, pulsaremos `CTRL + D` o introduciremos la orden **`exit`** para salir del subshell y dejar que cree el parche `debian/patches/-01-man-fhs.patch` con los cambios que hemos hecho.

Sólo hemos de colocar ahora el parche en el sitio correcto, y guardar los cambios en el repositorio:

```
cd ~/packages/zsnes/trunk
cp -r ../build-area/zsnes-1.510/debian/patches debian
svn add debian/patches
svn commit -m "Arreglado problema con la página man en /usr/man"
```

Ahora reconstruiremos el paquete:

```
svn-b
```

Ya, por fin, Lintian y Linda no dan ningún aviso. Probaremos a instalar el paquete, y ver qué tal funciona:

```
sudo dpkg -i ../build-area/zsnes_1.510-0ubuntu1_i386.deb
```

Tras probar un poco el emulador con alguna ROM libremente disponible, como las de PDRoms (http://www.pdroms.com), decidimos que el paquete está en condiciones de ser usado. Usaremos la otra función que definimos antes para marcar esta versión del paquete, copiando los contenidos de `trunk` en un nuevo subdirectorio de `tags` cuyo nombre será la versión actual del paquete, para después confirmar los cambios que se habrán realizado automáticamente en `debian/changelog`, en preparación para la siguiente versión de nuestro paquete:

```
svn-tag
svn commit -m "Copiado a tags versión 1.510-0ubuntu1 del paquete"
```

Envío al repositorio Debian

Después de haber verificado nuestro paquete y haber marcado la versión en el repositorio Subversion, es momento de enviarlo al repositorio Debian, para que nuestros usuarios puedan acceder fácilmente a él. Añadiremos tanto el paquete fuente como el paquete compilado:

```
sudo -E reprepro includedeb gutsy \
 ~/packages/zsnes/build-area/zsnes_1.510-0ubuntu1_i386.deb
sudo -E reprepro -S main -P low includedsc gutsy \
 ~/packages/zsnes/build-area/zsnes_1.510-0ubuntu1.dsc
```

Deberíamos probar el paquete justo como un usuario normal lo haría. Pero, primero, tenemos que desinstalar la versión que instalamos antes a partir de un fichero:

```
sudo aptitude remove zsnes
sudo aptitude update
sudo aptitude install zsnes
```

A.3.3. Actualización del paquete a una nueva versión del programa

Cada cierto tiempo, deberemos de actualizar nuestro repositorio con los cambios realizados en el código. Para ello, disponemos de la orden **`svn-upgrade`**, a la que le pasaremos un fichero `tar.gz` con el nuevo código fuente del paquete.

Ahora probaremos a actualizar nuestro paquete con el código correspondiente a la última versión en el repositorio Subversion del equipo de ZSNES. Primero, prepararemos el tarball con el código original. Descargamos el código fuente, y lo exportamos a otro directorio, para retirar los ficheros usados por *Subversion*:

```
svn checkout https://svn.bountysource.com/zsnes/trunk zsnes-svn
svn export zsnes-svn zsnes-1.510.SVN5215
```

He decidido usar el esquema `1.510.SVN5215` para indicar que se trata de la revisión 5215 del repositorio. De esta forma, si sale una nueva versión oficial, o si empleo una revisión más reciente del repositorio Subversion, como `1.520` o `1.510.SVN5216`, éstas serán considerada más recientes, y la actualización se podrá hacer de forma correcta. Podemos comprobar que es efectivamente así mediante la siguiente orden:

```
dpkg --compare-versions "1.510-0ubuntu1" gt "1.510.SVN5215-0ubuntu1" && \
  echo "1.510-0ubuntu1 mayor que 1.510.SVN5215-0ubuntu1" || \
  echo "1.510-0ubuntu1 menor que 1.510.SVN5215-0ubuntu1"
```

Ahora que ya tenemos el código, inspeccionaremos un momento para ver si todos los ficheros importantes se hallan en su sitio. Normalmente, no se suelen almacenar ficheros generados automáticamente en el repositorio, así que pueden que falten algunas cosas importantes. Mirando en `src`, vemos que falta el guión `configure` que necesitamos para compilar. Lo generaremos ejecutando las siguientes órdenes bajo `src`, el mismo directorio donde se halla su fichero fuente, `configure.in`:

```
aclocal
autoconf
```

También faltan los directorios `docs/readme.txt` y `docs/readme.htm`, que añadiremos en su sitio. Con esto tenemos los ficheros fuente listos. Vamos a crear el tarball que necesitamos:

```
/tmp$ tar -czf zsnes-1.510.SVN5215.tar.gz zsnes-1.510.SVN5215
```

El siguiente paso es añadir dicho código a nuestra área de trabajo, e indicar que vamos comenzar a empaquetar una nueva versión del programa. Volvemos al directorio con la versión actual de nuestro paquete y ejecutamos:

```
~/packages/zsnes/trunk$ svn-upgrade /tmp/zsnes-1.510.SVN5215.tar.gz
```

Se harán los cambios necesarios en `debian/changelog` y el resto del repositorio. Cambiaremos el número de versión del paquete a `0ubuntu1` y confirmamos dichos cambios antes de volver a reconstruir el paquete:

```
~/packages/zsnes/trunk$ svn commit -m \
  "Actualizado con rev 5215 upstream"
~/packages/zsnes/trunk$ svn-b
```

Sin embargo, la reconstrucción falla. Buscando entre los distintos mensajes, podemos ver una línea del estilo:

```
Trying patch 01-man-fhs.patch at level 1 ... 0 ... 2 ... failed.
```

En resumen: el parche que antes hicimos para corregir el problema de `Makefile.in` no se ha podido aplicar. Tras inspeccionar sus contenidos, vemos que efectivamente `Makefile.in` ha cambiado demasiado:

```
install:
@INSTALL@ -d -m 0755 $(DESTDIR)/@bindir@
@INSTALL@ -m 0755 @ZSNESEXE@ $(DESTDIR)/@bindir@
@INSTALL@ -d -m 0755 $(DESTDIR)/@mandir@/man1
@INSTALL@ -m 0644 linux/zsnes.1 $(DESTDIR)/@mandir@/man1
```

Vaya, parece que ellos mismos han corregido ya el problema que teníamos antes. Ésta es precisamente la razón por la que usamos un parche, en vez de cambiar directamente el código. Sólo tenemos que eliminar el parche, confirmar los cambios y reconstruir:

```
~/packages/zsnes/trunk$ svn rm debian/patches
D  debian/patches/01-man-fhs.patch
~/packages/zsnes/trunk$ svn commit -m \
  "01-man-fhs.patch: Retirado (corregido por upstream)"
Deleting       zsnes/trunk/debian/patches/01-man-fhs.patch
Transmitting file data .
Committed revision 11.
~/packages/zsnes/trunk$ svn-b
```

Esta vez ya no da el fallo del parche, pero indica que faltan las bibliotecas de desarrollo de Qt para la nueva interfaz. Añadiremos la dependencia en `libqt4-dev` al fichero `control` y volveremos a intentarlo tras confirmar otra vez nuestros cambios.

Esta vez falla diciendo que no puede construir el fichero `ui_zsnes.h`, que hace falta para compilar. Probaremos a compilar sobre el directorio `/tmp/zsnes-1.510.SVN5215/-src` tras instalar las dependencias en nuestro propio sistema:

```
./configure
make
```

Da el mismo problema, así que no es culpa de nuestro paquete. Vamos a mirar con *grep*, a ver qué puede ser:

```
grep -R ui_zsnes.h *
makefile.ms:$GUI_D/gui.cpp: $GUI_D/ui_zsnes.h
src/gui/gui.h:#include "ui_zsnes.h"
Makefile.in:GUI_QO=$(GUI_D)/moc_gui.cpp $(GUI_D)/ui_zsnes.h
Makefile:GUI_QO=$(GUI_D)/moc_gui.cpp $(GUI_D)/ui_zsnes.h
makefile.dep:gui/gui.o: gui/gui.cpp gui/gui.h ui_zsnes.h
```

Si nos fijamos, se puede ver que en `makefile.dep` la ruta no se corresponde con la de las anteriores entradas, por lo que seguramente ahí estará el fallo. Esta vez buscamos por este fichero:

```
grep -R makefile.dep *
src/configure:touch -t 198001010000 makefile.dep
src/Makefile.in:main: makefile.dep $(Z_QOBJS) $(Z_OBJS)
src/Makefile.in:include makefile.dep
src/Makefile.in:makefile.dep: $(TOOL_D)/depbuild Makefile
src/Makefile.in: $(TOOL_D)/depbuild @CC@ "@CFLAGS@" @NASMPATH@
"@NFLAGS@" $(Z_OBJS) > makefile.dep
src/Makefile.in: rm -f makefile.dep $(Z_OBJS) $(Z_QOBJS) $(PSR)
$(PSR_H) @ZSNESEXE@
src/Makefile:main: makefile.dep $(Z_QOBJS) $(Z_OBJS)
src/Makefile:include makefile.dep
src/Makefile:makefile.dep: $(TOOL_D)/depbuild Makefile
src/Makefile: $(TOOL_D)/depbuild gcc " -pipe -I. -I/usr/local/include
-I/usr/include -D__UNIXSDL__ -I/usr/include/SDL -D_GNU_SOURCE=1
-D_REENTRANT -D__OPENGL__ -DNO_DEBUGGER -DNDEBUG -march=athlon-xp -O2
-fomit-frame-pointer -s -DQT_SHARED -I/usr/include/qt4
-I/usr/include/qt4/QtCore -I/usr/include/qt4/QtGui " nasm "
-w-orphan-labels -D__UNIXSDL__ -f elf -DELF -D__OPENGL__ -DNO_DEBUGGER
```

```
-O1" $(Z_OBJS) > makefile.dep
src/Makefile: rm -f makefile.dep $(Z_OBJS) $(Z_QOBJS) $(PSR) $(PSR_H)
zsnes
src/autom4te.cache/output.0:touch -t 198001010000 makefile.dep
src/autom4te.cache/output.1:touch -t 198001010000 makefile.dep
src/configure.in:touch -t 198001010000 makefile.dep
```

Puede verse que en `src/Makefile` es donde se crea a través de una herramienta propia de los desarrolladores de ZSNES, que obtiene automáticamente las dependencias. Recordando la lista anterior de ficheros que mencionaban a `ui_zsnes.h`, se nos viene a la cabeza `gui/gui.h`. Vamos a probar a cambiar la ruta de inclusión a `gui/ui_zsnes.h`:

```
cd ~/packages/zsnes/trunk
svn-buildpackage --svn-export
cd ../build-area/zsnes-1.510.SVN5215
cdbs-edit-patch 02-fix-depbuild.patch
```

Hacemos el cambio antes mencionado con el editor *vim* y salimos del shell, tras lo cual reintentaremos la construcción del paquete:

```
vim src/gui/gui.h
exit
```

Ahora la reconstrucción ha tenido éxito. Ya sólo tendríamos que seguir los mismos pasos anteriores de verificación, validación manual, marcado en el repositorio (tras retirar el aviso «NOT RELEASED YET» añadido automáticamente a `debian/changelog`, claro), y envío. Cualquiera de nuestros usuarios será notificado eventualmente de la actualización y podrá instalar la versión más reciente del paquete.

Una última nota: si se quiere, se puede integrar la verificación y publicación en la función **`svn-b`** antes definida, cambiando las líneas correspondientes en `~/.bashrc` por:

Listado A.3: Funciones Bash de ayuda para construcción de paquetes

```
function svn-b () {
  buildarea="`pwd`/../build-area";
  rutapaquete="${buildarea}/\${package}_\${debian_version}*.deb";
  rutadsc="${buildarea}/\${package}_\${debian_version}*.dsc";
  rutarepo="/var/packages/ubuntu";
  sudo cowbuilder --update
  svn-buildpackage \
    --svn-builder="pdebuild --auto-debsign --buildresult ${buildarea}" \
```

```
    --svn-postbuild="rm ${buildarea}/*_source.changes; \
          lintian -i ${rutapaquete}; linda -i ${rutapaquete}; \
          cd ${rutarepo} && \
          sudo -E reprepro remove gutsy \${package} && \
          sudo -E reprepro includedeb gutsy ${rutapaquete} && \
          sudo -E reprepro -S main -P low includedsc gutsy ${rutadsc}";
}
function svn-tag () {
  svn-buildpackage --svn-only-tag
}
```

A.4. Otros aspectos de interés

A.4.1. Integración de repositorio propio con la jaula *chroot*

En este tutorial no hará falta, pero sí es posible que nos haga falta para casos más complejos. Cuando haya interdependencias entre varios paquetes que deseemos crear, tendremos que añadir nuestro repositorio (que estará alojado en su propio servidor web) a la lista de los repositorios de la jaula *chroot*.

Para ello, nos introduciremos en la jaula e importaremos la firma digital de nuestros paquetes:

```
sudo cowbuilder --login --save-after-login
wget http://direccion.delservidor.org/claveDebian.asc
apt-key add claveDebian.asc
```

Con esto, la jaula ya confiará en nuestros paquetes. Tenemos que añadir entonces la siguiente línea a `/etc/apt/sources.list`, usando el editor *vim*:

```
deb http://direccion.delservidor.org/ubuntu gutsy main
```

Actualizamos la lista de paquetes:

```
aptitude update
```

Hemos terminado, así que sólo queda salir del shell de la jaula:

```
exit
```

A.4.2. Sincronización de un repositorio en Internet con un repositorio local

En muchos casos no dispondremos de los medios necesarios como para dejar nuestro ordenador como servidor web al exterior para que sirva nuestros paquetes. En estos casos, podemos subir nuestros ficheros `.deb` a una forja o algo del estilo y dejar que los usuarios se los bajen.

Pero, ¿y si son muchos paquetes, o si queremos mantener actualizados a los usuarios? Lo mejor sería replicar completo el repositorio en un servidor conectado permantentmente a Internet. La mejor opción es, por tanto, obtener espacio de alojamiento con acceso mediante SFTP o FTP.

Una vez lo hayamos conseguido, podríamos subir el árbol completo de directorios de nuestra máquina al servidor remoto, pero hacer esto una y otra vez tardaría demasiado. La mejor opción es hacer un *mirror inverso* a través de la herramienta *lftp*, que nos pedirá la contraseña antes de conectarse:

```
lftp -d -e "mirror -venR ruta local \
                         ruta remota \
                         dirección del servidor"
```

Si queremos que no nos pregunte una y otra vez la contraseña, siempre podemos añadir líneas como éstas a `~/.netrc`, cuidando de que sólo sea legible por nosotros:

```
machine mi.servidor.com
login minombredeusuario
password micontraseña
```

A.4.3. Adaptación de aplicaciones Java

La principal diferencia al crear un paquete Java es, sin duda, el hecho de que, a pesar de tener que compilar, el paquete en sí no tiene ninguna arquitectura definida. Ello debería verse reflejado en `debian/control`.

Además, debería tener en `Build-Depends` algún compilador de Java, y en `Depends` el paquete virtual `java2-runtime`, además de una alternativa concreta, como la de Sun, `sun-java6-jre`, o preferiblemente la basada en OpenJDK, `icedtea-java7-jre`, usando el operador lógico OR (`|`). Los paquetes virtuales no añaden ninguna funcionalidad: se limitan a hacer cosas como ayudarnos a instalar varios paquetes de una sola vez (usándolos en su campo `Depends`), o permitiéndonos usar un paquete cualquiera que nos provea de una cierta funcionalidad.

Así, si curioseamos en el paquete `sun-java6-jre`, veremos que tiene a dicho paquete virtual en el campo `Provides` («Proporciona»):

```
$ aptitude show sun-java6-jre
Paquete: sun-java6-jre
[...]
Proporciona: java-virtual-machine, java1-runtime, java2-runtime
[...]
```

Si usamos el compilador de Sun en Build-Depends (paquete `sun-java6-jdk`), habremos de cambiar nuestro `.pbuilderrc` para que use un interfaz distinto de entrada, que nos deje aceptar los términos de la DLJ de Sun. Añadimos estas líneas:

```
# Para poder aceptar la licencia DLJ
export DEBIAN_FRONTEND="readline"
```

Además de eso, tendremos como de costumbre que adaptar nuestro programa Java para que sea fácilmente compilable de forma automática, y para que se integre bien dentro del sistema. Un problema es que no deberíamos escribir rutas absolutas en el código Java, o si no perderemos toda la transportabilidad que deseábamos en un primer momento. La forma más fácil de implementar lo que deseamos es simplemente utilizar variables de entorno o propiedades del sistema.

A veces no nos servirá cualquier JRE de los disponibles para ejecutar nuestro programa, y tendremos que prescindir de usar `java2-runtime`. Es el caso usual de las aplicaciones basadas en Swing: el JRE por defecto en Gutsy, GCJ, sólo implementa completamente SWT. Tendremos que hacer que el sistema instale y active otro JRE.

Para poder establecerlas de forma separada del programa Java antes de lanzarlo, lo que necesitamos es un guión de shell, que será el que instalaremos en `/usr/bin`. El fichero `.jar`, siguiendo la política de Debian, debe ir en `/usr/share/java`.

El que sea fácilmente compilable de forma automática depende de qué sistema hayamos estado usando. Si nos hemos basado en un IDE, seguramente aquí tengamos problemas. Lo que deberíamos hacer es reescribir la fase de compilación usando un guión para *Apache Ant*. Esto, de paso, nos permitirá aprovechar el perfil de CDBS, simplificando bastante nuestra tarea. *Ant* es efectivamente un sustituto del *GNU Make* escrito en Java, para aplicaciones Java. Utiliza ficheros muy del estilo de los *makefile*, sólo que escritos en XML.

Con todo esto, aún hay un par de complicaciones adicionales. Normalmente, los ficheros fuente Java se hallan escritos en UTF-8. Sin embargo, la jaula *chroot* que antes hicimos en A.2.2 (página 200) no tiene en principio una localización compatible instalada. Vamos a tener que entrar en la jaula e instalar los paquetes necesarios, comprobando la situación que aquí describimos con **`locale`** (nos mostrará que usamos la localización estándar «C»):

```
sudo cowbuilder --login --save-after-login
aptitude install language-pack-es
exit
```

Ahora, lo que hemos de hacer es asegurarnos que cuando llamemos a *Ant*, lo hagamos estableciendo una localización con UTF-8:

```
LANG=es_ES.UTF-8 ant compile
```

Deberíamos además invocar las pruebas de unidad sobre el código, para asegurarnos de que el código usado en el paquete no presente regresiones. Además de añadir el paquete `junit` a `Build-Depends`, tenemos que tener en cuenta que si en alguna prueba de unidad usamos un componente Swing, aunque no se muestre nada en pantalla, requeriremos un servidor X, o de lo contrario la prueba fallará. Dado que ejecutar un servidor X completo en una jaula *chroot* es un gasto inútil de recursos, vamos a usar un falso servidor X, que añadiremos también a `Build-Depends`: `xvfb`. Para tener disponible el servidor X durante una orden, únicamente ponemos **`xvfb-run`** antes de ella. Combinándolo con lo de las localizaciones, tendremos que ejecutar una orden como ésta:

```
LANG=es_ES.UTF-8 xvfb-run ant (objetivo-compilación)
```

A.4.4. Actualización del escritorio

Antes conseguimos crear un paquete con un binario, junto con su página *man*. Sin embargo, para que realmente consideremos al paquete completo, deberíamos integrarlo con el gestor de escritorio del usuario. Veremos cómo añadir el acceso directo al menú, y cómo asociar los ficheros ROM de Super Nintendo al emulador ZSNES.

Esta tarea se ha hecho en los últimos años mucho más sencilla gracias a las especificaciones ofrecidas por el grupo FreeDesktop.org [27], que hace de centro de reunión para diversos proyectos relacionados con entornos gráficos. Así, KDE y GNOME emplean el mismo formato para describir las entradas de su menú, por ejemplo.

Accesos directos e iconos

Normalmente, existen dos sitios del menú de aplicaciones donde podemos instalar un acceso directo a nuestra aplicación: el menú Debian, y el propio menú normal de aplicaciones. Por completitud, trataremos ambos, aunque hoy en día el menú Debian no se usa demasiado. Además, no aparece por omisión en un sistema Ubuntu: hay que instalar el paquete `menu` para poder usarlo.

Para que se añada una entrada al menú Debian, hemos de asegurarnos de que la línea con **dh_installmenu** se halle descomentada en `debian/rules`. Dado que nuestro paquete se llama `zsnes`, crearemos un fichero `debian/zsnes.menu` con este contenido:

```
?package(zsnes):needs="X11" section="Games/Arcade" \
  title="ZSNES" command="/usr/bin/zsnes" \
  icon="/usr/share/pixmaps/zsnes.xpm"
```

El significado de los campos es el siguiente:

needs="X11" Indica que el programa tiene una interfaz gráfica, requiriendo un servidor X.

section="Games/Arcade" Especifica dónde debería situarse el acceso directo. Las secciones se hallan prefijadas por la política de empaquetado de Debian.

title="ZSNES" Éste es el texto que se mostrará en la entrada del menú.

command=/usr/bin/zsnes" Ésta es la orden exacta que será ejecutada.

icon=/usr/share/pixmaps/zsnes.xpm" Tal y como indica el grupo FreeDesktop.org, hemos instalado el icono en formato XPM y en la ruta `/usr/share/pixmaps`. Para convertir dicha imagen, podemos usar simplemente *Gimp*, o el programa **convert** del paquete `imagemagick`, de esta forma:

```
convert zsnes.png zsnes.xpm
```

Ahora hemos de centrarnos en que aparezca en los menús de KDE y GNOME. Para que sea así, hemos de crear otro fichero más, `debian/zsnes.desktop`, con el siguiente contenido:

```
[Desktop Entry]
Name=ZSNES
Type=Application
Comment=SNES (Super Nintendo) emulator that uses x86 assembly
Comment[es]=Emulador de SNES que emplea código ensamblador de x86
Exec=/usr/bin/zsnes %f
Icon=zsnes.png
MimeType=application/x-snes-rom;
Categories=Game;Emulator;
```

Dicho fichero ha de hallarse codificado tal y como indica el campo `Encoding`: en UTF-8. Además de indicar el nombre y el tipo de la aplicación, podemos especificar una descripción más larga. Adelanto un detalle para realizar la asociación de ficheros: el campo `Exec`, que contiene la orden a ejecutar, permite especificar sustituciones como

`%f`, que será sustituido por el primer fichero con el que se active el acceso directo. Si hay varios, se ejecutará el emulador tantas veces como ficheros haya, con un fichero distinto cada vez.

En `Categories` incluimos las categorías a las que pertenece esta aplicación: son más bien anotaciones semánticas que una descripción jerárquica de dónde debería colocarse exactamente. Debe de haber al menos una de las categorías principales según [26] (o si no no aparecerá en el menú), y pueden haber categorías adicionales para dar más información.

Los campos con texto descriptivo como `Name` y `Comment` se pueden personalizar según el idioma añadiendo al nombre de campo, entre corchetes, un identificador de idioma de ISO 639. He incluido un ejemplo para `Comment`. El campo `Icon` no requiere la ruta, ya que el directorio `/usr/share/pixmaps` es uno de los consultados de forma automática.

Otro detalle también importante para crear la asociación de fichero es especificar el tipo MIME de los ficheros que esta aplicación abre. Dado que no existe un tipo estándar MIME para ROM de Super Nintendo, definimos uno nosotros, cuidando de poner el prefijo `x-` en la segunda mitad, indicando que no es estándar.

Validaremos el fichero mediante **`desktop-file-validate`**, sin más problemas. En caso contrario habría que hacer las correcciones oportunas.

Ahora que tenemos los dos ficheros con la información necesaria, vamos a definir las acciones requeridas para instalarlos debidamente. En primer lugar, la llamada que necesitamos hacer a **`dh_installmenu`** ya la hace CDBS por nosotros, con lo que ya tendríamos la parte de instalar la entrada en el menú Debian. Aún tenemos que instalar el fichero `.desktop`, el icono, y añadir el icono al caché de GNOME, para mejorar la eficiencia. Añadimos estas líneas a `debian/rules`:

```
install/zsnes::
  dh_install -m 0644 $(CURDIR)/debian/zsnes.desktop  \
                     /usr/share/applications
  dh_install -m 0644 $(CURDIR)/debian/zsnes.xpm      \
                     /usr/share/pixmaps
  dh_desktop
  dh_iconcache
```

Cuando reconstruyamos el paquete, veremos que nos aparecerán dos ficheros nuevos en el directorio `debian`: `zsnes.postrm.debhelper` y `zsnes.postinst.debhelper`. Estos ficheros son guiones Bash generados automáticamente que se ejecutan después de la desinstalación e instalación. También hay guiones equivalentes para antes de la instalación y la desinstalación, llamados `preinst` y `prerm`, respectivamente. El sufijo `.debhelper` indica al mecanismo de construcción de paquetes que debe

concatenar su contenido con el que pudieran tener los ficheros sin tal sufijo, como `zsnes.postrm`, que nosotros mismos habríamos escrito manualmente.

En particular, **`dh_desktop`** ha añadido una llamada a `update-desktop-database`, y **`dh_installmenu`** otra llamada a `update-menu`. Con esto, nos aseguramos de que el menú sea actualizado debidamente y que el entorno de escritorio sepa que nuestro programa está disponible para abrir cualquier ROM de Super Nintendo.

Posteriormente, tras instalar el paquete, dichos guiones son guardados en `/var/-lib/dpkg/info`. Si alguna vez nos equivocamos al escribir alguno de ellos, es posible que no podamos ni terminar de desinstalar ni de instalar el paquete. Podemos forzar a que su ejecución termine con éxito añadiendo al inicio del guión en dicho directorio la línea:

```
exit 0
```

Así ya podremos retirar el paquete e instalar una versión con dicho fallo corregido.

Actualización de los tipos MIME

Ya nuestro sistema sabe que puede abrir ROM de Super Nintendo con ZSNES. Tenemos el acceso directo, y el icono. Sólo falta decirle al sistema cómo identificar una ROM de Super Nintendo.

En general, hay dos formas de identificar el tipo de un fichero: a través de una secuencia binaria específica en la cabecera (como en el caso de las imágenes en formato BMP, que incluyen siempre los caracteres «BM» al inicio del fichero), conocida como *magic cookie*, o a través de la extensión.

Aunque usar el contenido del fichero es mucho más robusto, no conozco realmente si siguen algún formato específico (probablemente no), así que nos limitaremos a la extensión, algo mucho más sencillo. En particular, nos centraremos en las dos más populares: `.sfc` y `.smc`.

Añadimos a nuestro paquete el fichero XML `debian/zsnes.sharedmimeinfo` siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<mime-info
  xmlns="http://www.freedesktop.org/standards/shared-mime-info">
   <mime-type type="application/x-snes-rom">
     <comment>SNES ROM</comment>
     <comment xml:lang="es">ROM de SNES</comment>
     <glob pattern="*.sfc"/>
```

```
      <glob pattern="*.smc"/>
    </mime-type>
</mime-info>
```

Tras la declaración XML, incluimos el elemento raíz `mime-info`, con la declaración del espacio de nombres para documentos de tipos MIME de FreeDesktop.org. Ahora tendríamos una serie de tipos MIME con `mime-type`, dando una línea descriptiva en posiblemente varios idiomas (de nuevo usando códigos de ISO 639), y posteriomente especificando patrones con los que identificar los ficheros que pertenecen a dicho tipo: `glob` usa un simple patrón de ficheros del shell (no se trata de una expresión regular, como podemos ver), y `magic` nos permitiría identificar por contenido.

Ahora deberíamos asegurarnos de que la línea que llama a **`dh_installmime`** se halle descomentada, si usáramos sólo *debhelper*, pero con CDBS no hay que hacer nada más. La llamada a **`update-mime-database`** será añadida automáticamente a los guiones de postinstalación y postdesinstalación por el guión anterior, y el fichero será instalado en su lugar correcto, `/usr/share/mime/packages`. Podemos comprobar, tras instalar el paquete, que la asociación se ha realizado bien inspeccionando `/usr/share/mime/globs`, que debería contener las líneas:

```
application/x-snes-rom:*.sfc
application/x-snes-rom:*.smc
```

A.4.5. Generación automática de paquetes

Módulos Perl

El guión **`dh-make-perl`** nos permite crear rápidamente versiones preliminares de paquetes Debian para módulos Perl del CPAN (Comprehensive Perl Archive Network) [32]. De esta forma, podemos integrar todos los módulos necesarios para nuestra aplicación Perl en paquetes, y que así el usuario pueda instalarla justo como cualquier otra aplicación.

Así, si queremos empaquetar el módulo `XML::Writer` del CPAN, escribiremos:

```
dh-make-perl --cpan XML::Writer
```

Es posible que tengamos que hacer algunos retoques a `debian/control` o a `debian/rules` si el sistema de compilación del paquete no es compatible con el habitual (MakeMaker), o si tiene dependencias que no puedan detectarse automáticamente. Por lo general, no habrá mucho que hacer: el CPAN impone ciertas cosas que hacen que la tarea sea bastante más sencilla que con otros programas.

Una nota: el servicio CPAN tiene enlaces a otro servicio muy útil que indica exactamente las dependencias de un paquete Perl cualquiera de forma recursiva, y nos indica cuáles módulos se hallan preconfigurados (y no requieren de un paquete por lo tanto) y cuáles no.

También es útil instalar el paquete *apt-file*, que es capaz de encontrar el paquete que contiene un determinado fichero, de tal forma que pueda ser añadido automáticamente a las dependencias del paquete generado. Es importante actualizar el caché cada cierto tiempo de esta forma:

```
sudo apt-file update
```

Adaptación de guiones de instalación

Un caso muy común es cuando nosotros mismos tenemos que compilar alguna aplicación, porque no dispongamos de un paquete apropiado. ¿Deberíamos directamente instalarla, y perder así las ventajas de una desinstalación limpia y segura, y la posibilidad de posteriormente actualizar a una versión posterior si finalmente aparece un paquete?

No hace realmente falta: *checkinstall* [20], cuyo paquete del mismo nombre tendremos que instalar, se ocupa de invocar **`make install`**, seguir todo el rastro de la instalación, y crear un rudimentario paquete Debian (también hay soporte para paquetes Slackware y RPM) a partir de él. Realiza la instalación automáticamente, y nos deja un paquete en el mismo directorio, que podemos pasar a cualquiera para que también lo instale. Por supuesto, el paquete no podremos subirlo a ningún repositorio serio: no tiene ninguna información de dependencias, por ejemplo, ni se halla firmado.

Si por ejemplo estamos compilando un programa basado en las *autotools* (unión de *autoconf*, *automake* y a veces *libtool*), sólo habría que cambiar el último paso:

```
./configure --prefix=/usr
make
sudo checkinstall
```

A.4.6. Otros formatos de paquete

Comparativa con otros formatos

Los paquetes Debian son sólo un tipo más de paquete que podemos encontrar. Otras distribuciones han desarrollado sus propios sistemas de empaquetado, con mayor o menor funcionalidad, y con mayor o menor éxito. En esta guía, mencionaremos los otros tres formatos más populares: RPM, Portage y Slackware.

RPM El formato RPM (RedHat Package Manager) es el empleado actualmente en las distribuciones SUSE y Fedora, entre otras, y es originario de la ahora desaparecida RedHat Linux. Tiene funcionalidad en el mismo nivel de los paquetes Debian, con guiones de pre/postinstalación y desinstalación, y metadatos, como por ejemplo las dependencias. Sin embargo, esta información de dependencias es menos rica que la de los paquetes Debian: sólo existe un tipo de dependencia, mientras que en los paquetes Debian tenemos recomendaciones (paquetes que casi siempre necesitaremos junto con el actual) y sugerencias (paquetes que mejorarían la funcionalidad del actual). Además, en los paquetes Debian podemos especificar alternativas en las dependencias, y usar paquetes virtuales, que indiquen la disponibilidad de una cierta funcionalidad, como «navegador web» o «entorno de ejecución Java», más que un software determinado.

Por otro lado, a diferencia de los paquetes Debian, incorpora guiones de verificación de la correcta instalación de un paquete, y disparadores al cambiar el estado de algún otro paquete. También incluye la capacidad de definir dependencias sobre ficheros, aunque a muchos no les parece realmente útil.

El principal problema de este formato era la falta de resolución automática de dependencias en la herramienta de gestión de paquetes **`rpm`** estándar: ésta simplemente informaba de las dependencias directas que no habían sido cumplidas, obligándonos a reintentar el proceso bajando un paquete tras otro hasta finalmente conseguir instalar el paquete deseado. Sin embargo, ya existen herramientas que añaden esta resolución de forma transparente, como **`yum`** de Fedora, *YaST* de SUSE, o **`urpmi`** de Mandriva.

Gracias a esas herramientas, hoy en día el problema es distinto, y se basa más en la forma y contenido de los paquetes en sí: no existe una política estricta y estándar de empaquetado como la que tienen los paquetes Debian [36], ni herramientas de validación. Por ello, un RPM no suele funcionar bien fuera de la versión específica de la distribución en que se desarrolló. Además, no hay tanto software empaquetado en formato RPM como lo hay en formato Debian, ni de forma tan accesible.

Portage El sistema Portage de Gentoo es una versión para Linux del sistema *ports* de FreeBSD: realmente, un paquete de Gentoo se halla formado por un fichero `ebuild`, que contiene las instrucciones de cómo obtener, compilar, instalar y configurar el software.

Normalmente, tras compilar el paquete, obtenemos también un paquete parecido a los que estamos acostumbrados a ver. Así, si tenemos varias máquinas con la misma versión de Gentoo, no tendremos que compilar en cada máquina. De la misma forma, existen también paquetes binarios precompilados si los necesitamos. De esa forma evitaremos tener que recompilar muchas aplicaciones grandes, como OpenOffice o KDE, por ejemplo.

Los ficheros `ebuild` incluyen información de dependencias, y la herramienta usada para la instalación, **`emerge`**, las sigue de forma transitiva. También disponemos de pa-

quetes virtuales, con la misma semántica que en Debian. El único problema se halla en las dependencias inversas: a la hora de retirar un paquete, Portage no comprueba si estamos rompiendo algún otro paquete. Existen herramientas como **`revdep-rebuild`** que hacen esta comprobación por nosotros, pero no se hallan integradas con el proceso de desinstalación.

Otro problemas incluyen el tener que realizar mantenimiento constante sobre los ficheros de configuración de los paquetes que actualicemos, o el ciclo de desarrollo más corto de los paquetes Gentoo frente a los de Debian. Aunque normalmente tendremos software mucho más reciente, no estará tan probado como lo podría estar un paquete Debian, y podemos llevarnos más de un disgusto.

Por lo demás, se trata de un sistema muy completo, con la capacidad de actualizar todo nuestro sistema comprobando nuestras dependencias de forma transitiva automáticamente, entre otras cosas. Obtenemos las actualizaciones sincronizando de forma periódica nuestro árbol de ficheros `ebuild` con un directorio remoto mediante **`rsync`**.

Slackware El formato `tgz`, de Slackware: es únicamente un `tar.gz` que será descomprimido directamente a `/`, para después ejecutar un guión de postinstalación. Se corresponde con la filosofía de dicha distribución: el usuario es el que sabe qué hay que hacer.

No incorpora metadatos de ningún tipo, ni control de dependencias. Es posiblemente el formato más sencillo de empaquetado que podríamos imaginarnos.

Conversión desde paquetes Debian

Puede que queramos que usuarios de SUSE, Slackware o Fedora, por ejemplo, empleen nuestro programa. Quizás no tengamos tiempo como para mantener un paquete para todos y cada uno de los otros muchos formatos disponibles.

La herramienta *alien* (como siempre, antes deberemos instalar su paquete) nos deja convertir entre paquetes Debian, Slackware, Solaris, RPM, LSB y Stampede. Es bastante experimental, y los paquetes generados no tendrán la misma calidad que uno hecho a mano, pero puede ser útil en muchos casos.

Así, para convertir cualquier paquete a formato Debian, usaremos:

```
alien (ruta al paquete)
```

Por otro lado, si queremos crear un paquete RPM a partir de un paquete Debian, podríamos usar, como en el caso de ZSNES:

```
alien --to-rpm \
  \~{}/packages/build-area/zsnes\_1.510.SVN5113-0ubuntu1\_i386.deb
```

Sólo hemos de tener cuidado de que la conversión se haya hecho lo bastante bien: por ejemplo, los guiones de preinstalación, postinstalación y demás no se convierten bien al formato RPM. Posiblemente tendremos que editar los ficheros generados para asegurarnos de que funcionen bien.

B GNU Free Documentation License

Version 1.2, November 2002

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers

to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing

tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title

equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of

peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Bibliografía

[1] Jon Allen. *Perl v5.10.0 Documentation*. http://perldoc.perl.org/, marzo 2008.

Esta web es la fuente principal de documentación de Perl. Su extensiva documentación en línea incluye una referencia completa de todas las palabras reservadas y funciones predefinidas de Perl, junto con tutoriales y listas de preguntas más frecuentes.

[2] Ken Arnold, James Gosling, y David Holmes. *El Lenguaje de Programación Java(TM)*. Addison-Wesley, Madrid, tercera edición, 2001.

Este libro, en el que participa uno de los creadores de Java, James Gosling, describe la sintaxis y el funcionamiento del lenguaje Java y de un pequeño subconjunto de su gran biblioteca estándar.

[3] *Assembla: About*. http://www.assembla.com/about, 2008.

Información acerca del espacio de trabajo Assembla, que ofrece espacios de desarrollo públicos o privados de forma gratuita, con la posibilidad de ampliar sus capacidades bajo pago. Los proyectos de código abierto reciben algo de tratamiento especial, consiguiendo mayor espacio en disco de forma gratuita. En cierta forma, Assembla podría verse como una forja comercial.

[4] Luciano Baresi y Michal Young. *Test Oracles*. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

Recopilación de los tipos de oráculos dirigidos a pruebas más conocidos hasta el momento, clasificados según las técnicas empleadas. Expande sobre el concepto de *oráculo* y muestra algunas posibilidades para los oráculos de sistemas conversores ("transducers" en el original).

[5] Kent Beck y Cynthia Andres. *Extreme Programming Explained - Embrace Change*. Addison-Wesley Professional, segunda edición, noviembre 2004.

La verdadera difusión de Extreme Programming fue iniciada a través del lanzamiento en 1999 de la primera edición de este libro, el primero acerca de dicha metodología, en la que se introducían los valores, principios y prácticas que formaban sus cimientos. En esta segunda

edición, los autores han revisado el planteamiento básico de Extreme Programming en base a las críticas realizadas por la comunidad de desarrolladores. Un ejemplo es la reestructuración de las prácticas (inicialmente todas obligatorias) en dos conjuntos: un núcleo obligatorio y un conjunto de prácticas opcionales, cuya aplicación no se recomienda hasta implementar efectivamente el núcleo obligatorio.

[6] Kent Beck y Martin Fowler. *Planning Extreme Programming*. Addison-Wesley, 2000.

Este libro explora en mayor profundidad la planificación de un proyecto basado en la metodología Extreme Programming, centrada fundamentalmente en las historias de usuario, y da recomendaciones sobre cómo escribirlas o realizar sus estimaciones.

[7] Oren Ben-Kiki, Clark Evans, y Ingy döt Net. *YAML Ain't Markup Language (YAML™) Version 1.2*. http://yaml.org/spec/1.2/, mayo 2008. Working Draft.

Esta es la versión 1.2 de la especificación de YAML, la más reciente a fecha de hoy. YAML es un lenguaje de serialización fácil de leer y escribir por humanos, e interoperable con un amplio número de lenguajes. Esta versión es un borrador prácticamente finalizado, hallándose en una última fase de discusión antes de su aprobación final.

[8] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, y François Yergeau. *Extensible Markup Language (XML) 1.0*. Informe técnico, World Wide Web Consortium, septiembre 2006. http://www.w3.org/TR/xml/.

Especificación oficial por el W3C del Extensible Markup Language.

[9] Eric M. Burke y Brian M. Coyner. *Java Extreme Programming Cookbook*. O'Reilly, marzo 2003.

Todos los libros de Extreme Programming hablan a nivel teórico, pero no dicen cómo deberían implantarse en la práctica. Esta obra es un catálogo de las herramientas disponibles libremente para la aplicación de las prácticas de Extreme Programming, y en particular las pruebas de unidad. Da una breve introducción a herramientas como el gestor de tareas Ant, el marco de pruebas de unidad JUnit, y extensiones a éste como JFCUnit (para interfaces gráficas Swing) o XMLUnit (para pruebas de unidad XSLT o XML).

[10] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, y Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd., primera edición, junio 2000.

Este libro es una de las principales referencias en patrones arquitectónicos. Generaliza el concepto de patrón del libro "Patrones de Diseño", restringido en principio al propio diseño de un subsistema a todos los

niveles. Así, se formulan desde patrones de arquitectura, que describen las partes del sistema completo, hasta patrones de implementación, soluciones a determinados problemas que se ajustan mejor al lenguaje de programación empleado.

[11] Paul A. Cairns y Jeremy Gow. *Integrating Searching and Authoring in Mizar*. *J. Autom. Reasoning*, 39(2):141–160, 2007.

En este artículo, los desarrolladores de Alcor, una interfaz gráfica sobre el sistema de demostraciones automatizadas Mizar, discuten acerca de posibles enfoques para la definición de aplicaciones que hagan de asistentes en la realización de estas demostraciones. En particular, describen la gran importancia que ha tenido el cambio de Mizar a XML para sus salidas.

[12] James Clark. *XSL Transformations (XSLT) Version 1.0*. Recomendación del W3C, W3C, noviembre 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

Estándar de la W3C para el lenguaje de transformaciones sobre XML XSLT, en su versión 1.0.

[13] James Clark y Steven DeRose. *XML Path Language (XPath) Version 1.0*. Recomendación del W3C, W3C, noviembre 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

Estándar de la W3C para el lenguaje declarativo de identificación de nodos XML XPath, en su versión 1.0.

[14] Ben Collins-Sussman, Brian W. Fitzpatrick, y C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, primera edición, junio 2004. Disponible gratuitamente en http://svnbook.red-bean.com/nightly/en/index.html.

Este texto publicado bajo la licencia Creative Commons a través de continua comunicación con la comunidad de usuarios instruye progresivamente al lector acerca del uso efectivo de la funcionalidad de Subversion, desde el uso básico de un repositorio hasta el control de acceso y reparación de repositorios o la creación de un repositorio accesible mediante un navegador web.

[15] Marc Dequènes y Arnaud Patard. *CDBS Documentation*. https://perso.duckcorp.org/duck/cdbs-doc/cdbs-doc.xhtml, 2007.

[16] Steve DeRose, Eve Maler, y Ron Daniel. *XPointer xpointer() Scheme*. http://www.w3.org/TR/xptr-xpointer, 19 diciembre 2002.

[17] Antonio García Domínguez. *Forja de XMLEye en RedIris*. https://forja.rediris.es/projects/csl2-xmleye/, julio 2008.

Forja del presente Proyecto, en la que los usuarios pueden informar de fallos, proponer mejoras, descargar el código y tener acceso a las

copias más recientes de la documentación y los paquetes Debian, entre otras cosas.

[18] Antonio García Domínguez. *Wiki de XMLEye*. http://wiki.shoyusauce.org/, julio 2008.

Wiki del Proyecto presente. A partir de ahora toda la documentación se irá condensando en esta web, para evitar tener que duplicar esfuerzos, y permitir la colaboración de otros participantes en el futuro.

[19] Marius Ducea. *Ubuntu package version naming explanation*. http://www.ducea.com/2006/06/17/ubuntu-package-version-naming-explanation/, 2006.

[20] Felipe Eduardo Sánchez Díaz Durán. *CheckInstall*. http://asic-linux.com.mx/~izto/checkinstall/index.php, 2007.

[21] Bruce Eckel. *Strong Typing versus Strong Testing*. http://mindview.net/WebLog/log-0025, mayo 2003.

Mensaje del weblog del autor en que expone su cambio de postura frente a los lenguajes con sistemas de tipos dinámicos. Sugiere que más que un sistema estático de tipos, es mucho más útil un conjunto de buenas pruebas de unidad automatizadas: esto nos deja utilizar lenguajes dinámicos que imponen menos restricciones y dan una productividad mucho mayor. Como ejemplo de la nueva flexibilidad obtenida, muestra que podemos distinguir entre tipos no por sus nombres, sino por las operaciones que proporcionan, consiguiendo así un nivel mucho más fino de granularidad sin las restricciones que impone la herencia tradicional.

[22] Peter Seibel et al. *History of lexical scoping in Scheme and other Lisps?* http://groups.google.com/group/comp.lang.lisp/browse_thread/thread/12f458dd1e931338/9c1c8061eb6804fa?q=closures+scheme&lnk=ol&, octubre 2003.

Hilo en el grupo Usenet `comp.lang.lisp` abierto por Peter Seibel en el que se discute cómo Scheme no fue tanto el primer dialecto de Lisp en tener alcance léxico, como el primero en implementarlo al 100 %: envolver el código con `FUNCTION` parecía funcionar sin problemas en el Lisp 1.5 original, pero sólo iba a medias en todos los dialectos restantes.

[23] *Extreme Programming Roadmap*. http://www.c2.com/cgi/wiki?ExtremeProgrammingRoadmap, noviembre 2007.

Este sitio web fue el lugar en el que se gestó la metodología Extreme Programming. Aunque hoy en día no está muy activo, en comparación con el grupo Yahoo! en la dirección http://groups.yahoo.com/

group/extremeprogramming/, contiene una gran cantidad de información muy valiosa acerca de qué es Extreme Programming y cómo se puede implementar. Los propios usuarios dan sus experiencias y opiniones de la metodología, incluyéndose entre ellos el creador de la metodología ágil de desarrollo Crystal, Alistair Cockburn.

[24] Amy Fowler. *A Swing Architecture Overview*. http://java.sun.com/products/jfc/tsc/articles/architecture/, abril 2003.

Amy Fowler da una breve explicación en este artículo de la web de Sun acerca de las razones tras la arquitectura "de modelo separable" de la biblioteca Swing para interfaces de usuario de Java. Dicha arquitectura es un caso simplificado del patrón arquitectónico Modelo-Vista-Controlador donde se han unido la Vista y el Controlador. También explica cómo se consigue la implementación de los Look & Feel intercambiables a través del uso de un delegado instalado por el Look & Feel activo para el dibujado del control.

[25] Martin Fowler. *GUI Architectures*. http://martinfowler.com/eaaDev/uiArchs.html, julio 2006.

Expone y aclara los patrones arquitectónicos más comúnmente utilizados en los últimos años para estructurar las aplicaciones basadas en interfaces gráficas, pasando por los patrones MVC, MVP, y por el conocido Formularios y Controles, propio de herramientas RAD como C++ Builder. Es particularmente interesante el enfoque que utiliza respecto a las pruebas de unidad en interfaces gráficas: no es tanto cuestión de mejorar las herramientas de pruebas existentes, como definir un diseño que las facilite.

[26] FreeDesktop.org. *FreeDesktop.org Desktop Menu Specification*. http://standards.freedesktop.org/menu-spec/latest/, 2008.

[27] FreeDesktop.org. *FreeDesktop.org Specifications*. http://www.freedesktop.org/wiki/Specifications, julio 2008.

[28] Freedesktop.org. *Software/shared-mime-info*. http://freedesktop.org/wiki/Software/shared-mime-info, 2008.

[29] Neal Gafter. *A Definition of Closures*. http://gafter.blogspot.com/2007/01/definition-of-closures.html, enero 2007.

Mensaje del blog de Neal Gafter en el cual expone la historia de las clausuras lambda, y cómo podrían suponer una importante mejora para Java. Actualmente, participa en la definición de una JSR para su implementación y en la creación del prototipo asociado, como integrante del proyecto "Closures for the Java Programming Language" (http://www.javac.info/).

[30] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Patrones de Diseño*. Addison-Wesley, 2003.

Traducción al español del libro que popularizó el diseño basado en patrones. Se listan patrones de comportamiento, estructurales y de creación de objetos. Aunque el concepto de "patrón" como solución genérica a un tipo de problema en un contexto determinado fue creado por un arquitecto, Christopher Alexander, la disciplina donde ha tenido más éxito e implantación ha sido en la ingeniería de software.

[31] Charles F. Goldfarb. *The Roots of SGML – A Personal Recollection*. http://www.sgmlsource.com/history/roots.htm, 1996.

Recolección personal de las experiencias del creador de GML, el antecesor de SGML, el primer metalenguaje de marcado estructurado ampliamente usado.

[32] Jarkko Hietaniemi. *Comprehensive Perl Archive Network*. http://www.cpan.org, 2008.

[33] *IcedTea: Main Page*. http://icedtea.classpath.org/wiki/Main_Page, mayo 2008.

Proyecto iniciado por Red Hat que provee un entorno de compilación basado en herramientas libres para el código del proyecto OpenJDK, y reemplaza algunas partes propietarias con código de otros proyectos, como el GNU Classpath. Ha superado recientemente el TCK (Technology Compatibility Kit) de Sun, pasando a ser el primer entorno 100 % libre de J2SE 6.0 en conseguirlo (véase la dirección http://blog.softwhere.org/archives/196).

[34] Sun Microsystems Inc., editor. *Java(TM) Look and Feel Design Guidelines*. Addison-Wesley Professional, segunda edición, marzo 2001.

Este libro consiste en una guía de estilo para el diseño de interfaces gráficas de usuario bajo Java, y en particular bajo el Look & Feel Metal. Sin embargo, es lo suficientemente rico en prácticas recomendadas para usabilidad y accesibilidad como para poder extender sus consejos al diseño de interfaces gráficas en general.

[35] Sun Microsystems Inc., editor. *Java(TM) Look and Feel Design Guidelines: Advanced Topics*. Addison-Wesley Professional, primera edición, diciembre 2001.

Este libro extiende a su primera parte "Java(TM) Look and Feel Design Guidelines" sobre tópicos más especializados y complejos, como el diseño de ventanas, menús y asistentes.

[36] Ian Jackson y Christian Schwarz. *Debian Policy Manual*. http://www.debian.org/doc/debian-policy/, junio 2008.

[37] Ron Jeffries. *Misconceptions about XP*. http://www.xprogramming.com/xpmag/Misconceptions.htm, enero 2002.

Ron Jeffries desbanca algunos prejuicios acerca de Extreme Programming en este artículo de su web http://wwww.xprogramming.

com, como la idea de que en Extreme Programming no se genera documentación en absoluto (se crea sólo la necesaria) o que mediante la frase "siempre buscar la solución más simple" nos conformemos con soluciones inaceptables.

[38] Matt Kauffmann y J Strother Moore. *A Brief ACL2 Tutorial*. http://www.cs.utexas.edu/users/moore/publications/tutorial/rev3.html, 2000.

Esta demostración acerca del funcionamiento de ACL2, realizada en formato de diapositivas HTML, introduce de manera práctica lo que los autores llaman "El Método", la forma usual de trabajar con ACL2. El ejemplo consiste en demostrar que un algoritmo más avanzado de inversión de una lista es equivalente al algoritmo trivial de inversión.

[39] Matt Kauffmann y J Strother Moore. *Industrial Proofs with ACL2*. http://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms/intro-to-acl2.pdf, febrero 2005.

En este informe los creadores de ACL2 mencionan algunas de sus aplicaciones en el mundo real, como la demostración del cumplimiento del estándar IEEE (Institute of Electrical and Electronics Engineers) 758 de la unidad de coma flotante de los Athlon K5. También se explica de manera superficial qué es exactamente ACL2 y se dan recomendaciones para iniciar su aprendizaje.

[40] Matt Kaufmann y J Strother Moore. *ACL2 Version 3.3: BOOK-MAKEFILES*. University of Texas, Austin, EEUU, noviembre 2007.

Disponible electrónicamente bajo http://www.cs.utexas.edu/users/moore/acl2/v3-3/BOOK-MAKEFILES.html, esta página del manual de ACL2 describe cómo se organizan los libros de ACL2, y nos da una serie de convenciones con las que estructurar los proyectos multifichero.

[41] LispWorks Ltd. *Common Lisp HyperSpec: 1.1.2 History*, 2005. Disponible electrónicamente en http://www.lispworks.com/documentation/HyperSpec/Body/01_ab.htm.

Página de la documentación de Common Lisp en versión HTML generada automáticamente a partir del estándar oficial, que trata sobre la historia de Lisp, y menciona a Scheme directamente como el primer lenguaje que, entre otras cosas, introdujo la primera implementación completa de las clausuras lambda con alcance léxico.

[42] Alex Martelli. *polymorphism (was Re: Type checking in python?)*. http://groups.google.com/group/comp.lang.python/msg/e230ca916be58835?hl=en&, julio 2000.

Mensaje original del grupo de noticias `comp.lang.python` de Alex Martelli, a partir del cual se acuñó el término "duck typing".

[43] Gerard Meszaros. *xUnit Test Patterns*, capítulo Using Test Doubles, páginas 521–579. The Addison-Wesley Signature Series. Addison-Wesley, Westford, Massachussetts, EEUU, 1ª edición, mayo 2007.

Capítulo dedicado a los patrones de Dobles para Pruebas de un libro completo dedicado a técnicas y patrones relacionadas con pruebas de unidad basadas en el marco xUnit. Los Dobles para Pruebas sustituyen a un objeto del que depende nuestro código a probar, pudiendo realizar pruebas sobre las entradas y salidas desde y hacia esta dependencia.

[44] J Strother Moore. *Recursion and Induction*. http://www.cs.utexas.edu/~moore/publications/how-to-prove-thms/notes-version-1.pdf, mayo 2008.

Este texto, más que un tutorial, consiste en una colección de ejercicios diseñados para enseñar a razonar en una lógica formal, más concretamente la lógica de ACL2. Empieza por los fundamentos de la sintaxis y tipos de datos de Lisp, y finaliza con conceptos relativamente avanzados como los números ordinales.

[45] *OpenJDK*. http://openjdk.java.net/, 2008.

Proyecto liderado por Sun Microsystems que impulsa la publicación bajo licencias libres de la edición estándar de la Java Platform, y otros proyectos relacionados. Incluirá el código del JDK, JRE y de la JVM. Actualmente se halla en un estado muy avanzado.

[46] *Premios del II Concurso Universitario de Software Libre*. http://www.concursosoftwarelibre.org/noticias, mayo 2008.

Noticia del II CUSL en que se da un listado de los premios finales del CUSL, en los que XMLEye tiene el honor de aparecer como Primer Premio al Mejor Proyecto Educativo.

[47] *Python Tutorial: Glossary*. http://docs.python.org/tut/node18.html#l2h-46, febrero 2008.

Definición de Duck Typing por el glosario del tutorial oficial de Python. Esta técnica es la forma que se emplea comúnmente en lenguajes como Python o Perl para identificar un tipo no por quién es, sino por lo que hace.

[48] Ellie Quigley. *Perl By Example*. Prentice-Hall, tercera edición, 2002.

Este libro, a diferencia de otros, procura enseñar Perl de manera general a través de ejemplos prácticos comentados línea a línea. Da una visión amplia de los usos más avanzados de Perl, como la conexión a bases de datos, su uso para diseño de páginas web o el empleo de la parte orientada a objetos de Perl.

[49] Josip Rodin y Osamu Aoki. *Debian New Maintainer's Guide*. http://www.debian.org/doc/maint-guide/index.html, 2007.

[50] Gerarld Jay Sussman y Jr. Guy Lewis Steele. *Scheme: An Interpreter for Extended Lambda Calculus*. AI Lab Memo AIM-349, MIT AI Lab, Disponible electrónicamente en http://library.readscheme.org/page1.html, diciembre 1975.

Este artículo presenta la primera versión de Scheme, el primer intérprete de un dialecto de Lisp en utilizar por defecto alcance léxico y no dinámico. Así, al invocar a una función anónima (o *expresión lambda* en Lisp) devuelta que hiciera referencia a variables no locales, se usarían no las variables disponibles en el momento de la llamada, sino las disponibles en el momento de la definición de la expresión lambda. De esta forma se podrían mantener los axiomas del cálculo lambda propuesto por Alonzo Church. Este concepto de función anónima con ligaduras a las variables no locales disponibles en el momento de su definición, y que perduran más allá de la ejecución de la función que devolvío la función anónima, se conoce hoy en día como *clausura lambda*, y es una construcción de uso muy extendido entre los usuarios de Lisp y otros lenguajes funcionales, pero también presente en lenguajes como JavaScript o Python, en formas más limitadas.

[51] *SwingWiki*. http://www.swingwiki.org/, abril 2007.

Esta web creado por la comunidad de desarrolladores Swing describe diversas prácticas recomendadas y soluciones a diversos defectos de la biblioteca Swing. También lista ciertas características no documentadas de Swing, como la disponibilidad de fuentes con anti-aliasing.

[52] Josef Urban. *XML-izing Mizar: Making Semantic Processing and Presentation of MML Easy*. En Michael Kohlhase, editor, *MKM*, tomo 3863 de *Lecture Notes in Computer Science*, páginas 346–360. Springer, 2005. ISBN 3-540-31430-X.

En este artículo, uno de los desarrolladores de Mizar describe las motivaciones, las experiencias y los resultados obtenidos durante el cambio del formato de salida de Mizar de un formato binario *ad hoc* a XML. Indica que se ha producido un aumento considerable, pero no crítico, de los costes en espacio y tiempo, pero consideran que merece la pena, ya que facilitará la creación de nuevas herramientas para Mizar.

[53] Gary V. Vaughan, Ben Elliston, Tom Tromey, y Ian Lance Taylor. *Autoconf, Automake and Libtool*. http://sourceware.org/autobook/, 2006.

[54] Rodney Waldhoff. *Implementing the Singleton Pattern in Java*. http://radio.weblogs.com/0122027/stories/2003/10/20/implementingTheSingletonPatternInJava.html, diciembre 2003.

En este artículo del blog de Rodney Waldhoff, se nos muestra una implementación del patrón de diseño *Singleton* en Java que difiere de las sugeridas por sus creadores, empleando clases separadas para implementar el punto de acceso global y la funcionalidad. La clase que re-

presenta el punto global puede ser configurada en tiempo de ejecución para modificar la clase con la funcionalidad que se debe instanciar.

[55] Scott Walters. *Perl Design Patterns Wiki*. http://perldesignpatterns.com/, 2006.

Mantenida por Scott Walters, esta web acumula años de experiencia en la implementación del paradigma orientado a objetos en Perl y el uso de los conocidos patrones de diseño. Se mencionan también los anti-patrones, actividades o situaciones indeseables.

[56] why the lucky stiff (seudónimo). *YAML in Five Minutes*. http://yaml.kwiki.org/index.cgi?YamlInFiveMinutes, julio 2003.

Guía introductoria a YAML que da los aspectos más importantes en sólo 5 minutos. La fecha de la cita fue extraída del anuncio original en la lista de correo de YAML. He aquí una versión acortada de su dirección: http://tinyurl.com/5pjhvb.

[57] Ubuntu Wiki. *PackagingGuide/PatchSystems*. https://wiki.ubuntu.com/PackagingGuide/PatchSystems, 2007.

[58] Ubuntu Wiki. *PackagingGuide/Complete*. https://wiki.ubuntu.com/PackagingGuide/Complete, 2008.

[59] *WikiWikiWeb: Front Page*. http://www.c2.com/cgi/wiki?, junio 2008.

Sitio web del primer wiki de todos. Se halla dedicado a patrones de diseño y aspectos humanos del desarrollo de software.

[60] *YAXML, the (draft) XML Binding for YAML*. http://yaml.org/xml.html, 2006.

Borrador de la correspondencia de XML a YAML que he refinado y revisado para definir YAXML::Reverse.

[61] Bill Young. *The Towers of Hanoi Example*. http://www.cs.utexas.edu/users/moore/acl2/v3-3/TUTORIAL1-TOWERS-OF-HANOI.html, noviembre 2007.

Tutorial de la web oficial de ACL2 en que se ilustra su uso para demostrar que se requieren $2^n - 1$ movimientos para mover todos los n discos de una pila a otra en el problema de las Torres de Hanoi.