UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

xCvs. Un manejador de versiones extensible

TESIS

que para obtener el título de:

Licenciado en Ciencias de la Computación

presenta:

Mauricio Aldazosa Mariaca

Director de tesis: Licenciado Manuel Alberto Sugawara Muro

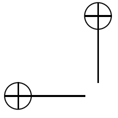
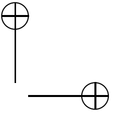
2006

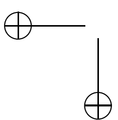
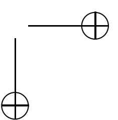
UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

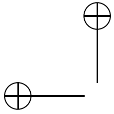
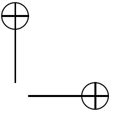
DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

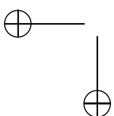
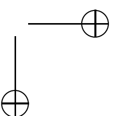



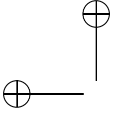
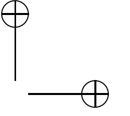



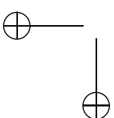
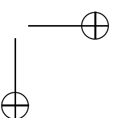



A mis padres, por todo su apoyo.

A Fer, por todos estos años conmigo.

VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"XCVS: Un manejador de versiones extensible"

realizado por Mauricio Aldazosa Mariaca

con número de cuenta 9954381-9 , quien cubrió los créditos de la carrera de:

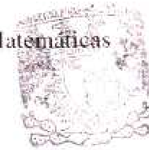
Lic. en Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

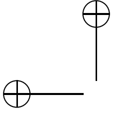
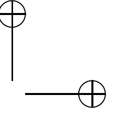
Director de Tesis Propietario	Lic. en C. C. Manuel Alberto Sugawara Muro
Propietario	M. en C. José de Jesús Galaviz Casas
Propietario	Dra. Elisa Viso Gurovich
Suplente	Lic en C. C. Karla Ramírez Pulido
Suplente	Lic. en C. C. Francisco Lorenzo Solsona Cruz

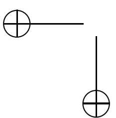
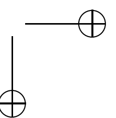
Consejo Departamental de Matemáticas



Dr. Francisco Hernández Quiroz

CONSEJO DEPARTAMENTAL DE MATEMATICAS

Índice general

1. Introducción **1**
1.1. Antecedentes . . . 1
1.1.1. Problemas del control de versiones . . . 3
1.2. Herramientas Actuales . . . 7
1.2.1. *SCCS (Source Code Control System)* y *CSSC* . . . 7
1.2.2. *RCS (Revision Control System)* . . . 7
1.2.3. *CVS (Concurrent Versions System)* . . . 7
1.2.4. *Aegis* . . . 8
1.2.5. *Subversion* . . . 8
1.3. Problemas de las herramientas actuales . . . 8
1.4. Propuesta . . . 9
1.5. Fundamentos . . . 10
1.5.1. Detección de cambios . . . 10
1.5.2. Gramáticas . . . 13
1.5.3. Clasificación de gramáticas . . . 15
1.5.4. Reconocimiento sintáctico . . . 22
1.5.5. JavaCC y JJTree . . . 22
1.5.6. XML y DiffXML . . . 23

2. Modelo y arquitectura **25**
2.1. Filosofía . . . 25
2.2. El servidor . . . 25
2.2.1. Elementos básicos del servidor . . . 26
2.2.2. Capa de Instrucciones . . . 30
2.2.3. Instrucciones a nivel usuario . . . 32
2.2.4. Instrucciones a nivel administrativo . . . 34
2.3. Repositorio . . . 40

2.3.1. La base de datos 40
2.3.2. El esquema de autorización 42
2.3.3. El esquema de trabajo 43
2.4. Excepciones 43

3. El manejador sintáctico 45
3.1. Transformación de un árbol de parseo a un JavaTree 45
3.1.1. Información relevante 46
3.1.2. El patrón *Visitor* 50
3.1.3. Análisis detallado de la información 52
3.2. Serialización de un JavaTree a un archivo XML 59
3.3. Información modificada 60

4. XcvsClient, Un cliente gráfico para xCVS 61
4.1. Administración de xCVS 61
4.1.1. Módulos 61
4.1.2. Tipos MIME 63
4.1.3. Usuarios y grupos 63
4.1.4. Manejadores 67
4.2. Utilización de xCVS 67
4.2.1. Inico de sesión 68
4.2.2. Creación de un módulo 68
4.2.3. Obtención de una copia local 69
4.2.4. Modificación de la estructura del módulo 71
4.2.5. Sincronición con el repositorio 72
4.2.6. Envió de cambios 72

5. Conclusiones y Trabajo futuro 75
5.1. Conclusiones 75
5.2. Trabajo futuro 75

A. Tablas en la base de datos 77
A.1. Esquema auth 77
A.1.1. Manejo de módulos 81
A.1.2. Manejo de archivos 82
A.1.3. Manejo de diferencias 85
A.2. Manejadores de entrada/salida 89
A.3. Esquema xcvs 90

Bibliografía 97

Índice de figuras

1.1. Un documento de texto visto como un árbol 5
1.2. El árbol de una abstracción de bajo nivel. 6
1.3. Una gramática que define expresiones aritméticas simples . . 14
1.4. La gramática anterior siguiendo las convenciones 15
1.5. Una derivación de la gramática 1.4 18
1.6. Una derivación alterna para la gramática 1.4 18
1.7. El árbol de derivación para 1.6 20
1.8. Una derivación para la cadena 1 + 2 * 3 20
1.9. Una derivación alterna para la cadena 1 + 2 * 3 21
1.10. Dos árboles de derivación para la misma cadena 21

2.1. Estructura de un módulo en xCVS 26
2.2. Un módulo muy simple 27
2.3. Un módulo en desarrollo 28
2.4. El patrón Command 31
2.5. Creando un nuevo usuario 41
2.6. El patrón DAO 41

3.1. Una operación aritmética con precedencia 46
3.2. El patrón Visitor 51

4.1. El panel de módulos 62
4.2. El panel de tipos MIME 63
4.3. El panel de extensiones 64
4.4. El panel de usuarios 65
4.5. El panel de grupos 66
4.6. El panel de manejadores 67

4.7. El diálogo para iniciar sesión 68
4.8. El diálogo para crear módulos 69
4.9. El menú de archivos 70
4.10. El menú de instrucciones 71
4.11. El diálogo para sincronizarse con el repositorio 72

Capítulo 1

Introducción

1.1. Antecedentes

El control de versiones consiste en el manejo de varias revisiones de una misma unidad de información. El control de versiones es una práctica que se desarrolló en ingeniería como un proceso formal basado en el rastreo de planos. El proceso de creación de estos planos es iterativo y en el caso en el que se haya seguido una ruta que llevó el diseño a un callejón sin salida, es necesario regresar a una versión anterior del mismo. Implícito en esta forma de control esta la posibilidad de lograr esto, pues simplemente es necesario obtener una copia anterior del diseño. De forma similar en la ingeniería de software se puede conocer como *control de versiones* a cualquier práctica que nos permita controlar la evolución sobre una unidad de información digital, como lo pueden ser archivos de código fuente, imágenes, diagramas o cualquier tipo de información en la que trabaja un grupo de personas. Aun cuando el control de versiones puede ser aplicado a cualquier tipo de información, en la práctica las herramientas y técnicas más sofisticadas raramente se utilizan fuera de los círculos de desarrollo de software (aunque por supuesto este tipo de prácticas podrían ser benéficas para otras áreas).

Si bien lo discutido en el párrafo anterior es la característica más utilizada del control de versiones no es lo único que puede ofrecer; existen otros problemas que pueden ser solucionados mediante el uso de estas prácticas. Durante el desarrollo e instalación de software, es muy común que múltiples versiones del mismo software sean instaladas en diferentes servidores, y que varios desarrolladores trabajen de forma privada en las actualizaciones al código fuente. Errores y otro tipo de problemas pueden ocurrir en solo algunas de esas distintas versiones (dado que la corrección de un error específico puede generar nuevos errores conforme el programa evoluciona). Es por esto que para el desarrollador encargado de localizar y corregir errores

es vital tener la capacidad de obtener y ejecutar diferentes versiones del software para poder determinar en cual versión es donde se encuentra el problema. Un ejemplo común de lo mencionado anteriormente es cuando durante el desarrollo de software es necesario implementar dos versiones paralelas del software, por ejemplo en una se quieren corregir errores plenamente identificados, mientras que en la otra rama se experimenta con nuevas características.

En la versión más sencilla del control de versiones todas las tareas son realizadas por los desarrolladores, los que pueden simplemente mantener varias copias de diferentes versiones de un programa y numerarlas apropiadamente. Esta aproximación inocente puede funcionar en proyectos pequeños, pero es ineficiente (dado que probablemente habrá muchas versiones casi iguales o idénticas de los archivos), requiere mucha disciplina por parte del desarrollador y es propensa a errores. Por estas razones es mejor automatizar algunos o todos los pasos involucrados en el control de versiones.

Tradicionalmente los sistemas de control de versiones utilizan un modelo centralizado, en el que todas las funciones son realizadas en un servidor compartido. Desde hace algunos años han surgido nuevos sistemas que hacen uso de un modelo distribuido, en el que cada desarrollador trabaja con su propia copia local del repositorio, y los cambios se comparten entre repositorios en un paso separado. Esta forma de trabajo permite a los desarrolladores sin una conexión de red permanente realizar cambios, además de que les otorga control completo sobre el desarrollo, pues no es necesario el permiso de una autoridad central.

En la mayoría de los proyectos de desarrollo de software varios desarrolladores trabajan en un programa al mismo tiempo y esto puede generar problemas de concurrencia. Si dos desarrolladores intentan modificar un archivo al mismo tiempo, sin ningún método para controlar el acceso bien podría darse el caso en el que un desarrollador sobreescriba el trabajo de otro. Uno de los métodos que utilizan algunos de los sistemas de control actuales es el uso de candados, esto es, cada archivo puede ser accedido sólo por un desarrollador a la vez (en cuanto a permiso de escritura se refiere). La otra forma en la que suele resolverse el problema es permitiendo que cualquier número de desarrolladores edite el archivo, y se provee de algún mecanismo para combinar todos los cambios realizados. La decisión sobre qué mecanismo utilizar para evitar estos problemas es fundamental en los sistemas de control, si bien el poner un candado a los archivos evita el uso de algoritmos de combinación complejos (incluso puede ser necesario que los usuarios deban resolver las diferencias entre los archivos), si un usuario retiene el candado a un archivo por demasiado tiempo los además usuarios

podrían verse tentados a ignorar el sistema de control de versiones y editar su copia del archivo, provocando esto más problemas.

Las características más comunes en los sistemas de control de versiones actuales son las siguientes:

- Arquitectura centralizada (cliente - servidor).
- Manejo de concurrencia mediante combinación de diferencias.
- Usualmente distinguen entre archivos de texto y binarios.

A lo largo de este trabajo se pretende desarrollar un manejador de versiones que sea extensible, sobre todo en el tipo de archivos que identifica y maneja. Esto implica que el programa debe tener una arquitectura que sea adecuada para agregar nuevos módulos a cualquier parte del proceso de manejo de versiones. Mediante esas extensiones se pretende obtener un manejador en el cual el proceso de la información y el manejo de diferencias sea óptimo, dependiendo del tipo de documento que se esté controlando. Este trabajo está dividido en cinco capítulos. En el primer capítulo se presenta una revisión rápida de las herramientas actuales y a las herramientas clásicas. Posteriormente vemos cuáles son algunos de los problemas que esas herramientas presentan. Para terminar el capítulo se analiza nuestra propuesta y se señalan las principales diferencias con lo revisado anteriormente. En el capítulo dos se presenta el corazón del programa, el modelo que se utiliza y la arquitectura que se escogió para la implementación. En el capítulo tres veremos cómo se construye desde el inicio un módulo de extensión que sea capaz de analizar código fuente en Java de una forma semántica. En el cápitulo cuatro se muestra cómo se realizan las operaciones más comunes con el programa desarrollado; finalmente en el capítulo cinco se encuentran las conclusiones y trabajo futuro.

1.1.1. Problemas del control de versiones

Los manejadores de versiones actuales tienen dos grandes propósitos, el primero es mantener un registro sobre cierto trabajo, registrar todas las versiones que han existido de él. El segundo propósito es ayudar en la colaboración entre varias personas que desarrollan un proyecto. De estos dos propósitos, el primero surgió como respuesta al problema que encuentran las personas al tratar de comparar dos versiones de un trabajo para ver cómo se ha modificado con el paso del tiempo, o en el caso de desarrolladores de software regresar a un versión estable de un programa, cuando el agregar el último detalle de tecnología a la versión actual termina por lograr que

el programa no funcione más. El segundo de estos propósitos es facilitar el trabajo en equipo. Cuando se está llevando a cabo un proyecto de software o escribiendo un artículo o libro generalmente hay más de una persona involucrada. En estos casos es necesario que cada una de ellas tenga una copia funcional del trabajo que se está llevando a cabo, y más importante aún es que cuando se lleve a cabo un cambio todas las personas interesadas en él (todos los que trabajan en el proyecto) se enteren de que algo cambió.

Es así pues que el primer problema con el que debemos tratar es el de la detección de cambios en archivos. Pensemos que tenemos un documento escrito, el cual acaba de ser enviado a un corrector de estilo. Para el momento en que el corrector de estilo regresa el borrador con sus correcciones nosotros hemos agregado más secciones al documento y hemos hecho algunas correcciones nosotros mismos. En este momento nos encontramos con dos versiones del mismo documento que necesitan combinarse en un solo documento. La mejor manera de lograr esto (rápidamente) es detectar en qué partes difieren la versión del corrector de estilo y la nuestra (partes del documento que entran en conflicto), para luego decidir cuales de esas partes son las que hay que agregar al documento final. Precisamente eso es lo que hacen las herramientas de control de versiones al cumplir con el primer propósito que mencionamos al principio de esta sección. La forma en la que trabajan la mayoría de estas herramientas es mediante el cálculo de la subsecuencia común más larga (LCS[1]) entre las dos versiones, para nuestra discusión actual basta saber que para trabajar con ese algoritmo lo que tomamos son dos cadenas de texto y las comparamos. Esto consigue que dicho algoritmo sea muy general, puesto que puede trabajar sobre cualesquiera dos cadenas, sobre cualquier alfabeto. Pero volviendo al ejemplo del documento y el corrector de estilo nos damos cuenta de que si bien, el documento es una enorme cadena de texto, el escritor y el corrector no lo ven así. Para ellos el documento presenta una *estructura*. Esta puede ser una abstracción de alto nivel, como la presentada en el índice, o de bajo nivel, como la que nos dan los párrafos, las oraciones, etc. Supongamos que el índice de nuestro documento es el siguiente:

1. Introducción
    a) Historia.
    b) Fundamentos.
2. Capítulo 1

[1]Longest Common Subsequence

Esto lo podemos ver como una estructura arborescente como la que se ve en la Figura 1.1.

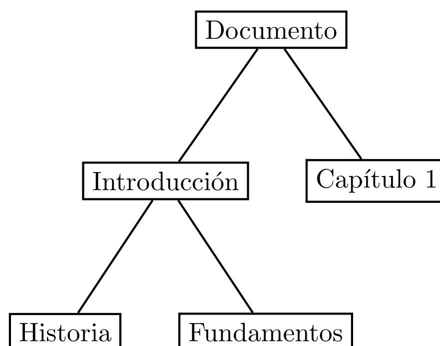


Figura 1.1: Un documento de texto visto como un árbol

Incluso si nos fijáramos en la abstracción de bajo nivel de párrafos y oraciones podríamos verlo como una estructura similar, como se aprecia en la figura 1.2.

Una vez que nos percatamos de que los documentos escritos tienen una estructura jerárquica podemos preguntarnos si el método de comparación mediante LCS es adecuado para estos documentos. La verdad es que al calcular diferencias de esa manera no estamos aprovechando información que tenemos a la mano, y con la cual se puede hacer un mejor trabajo de comparación. La desventaja que tenemos es que la estructura del documento (la de alto nivel) es diferente para cada tipo de documento. Una carta no tiene secciones, un artículo no tiene capítulos, mientras que un libro tiene ambas. Así pues perdemos la generalidad que teníamos al utilizar la comparación basada en LCS. Hay que tener cuidado al escoger el nivel de abstracción adecuado, ya que hay elementos en las abstracciones que no son comunes entre diferentes tipos de documentos, sin embargo en el otro extremo tenemos la separación del documento en caracteres, con lo que terminamos en la posición inicial y por lo tanto no ganamos información.

Como hemos señalado anteriormente, la parte más importante es decidir la abstracción adecuada para cada documento, pero en la actualidad las herramientas disponibles no presentan una gran variedad con respecto a los algoritmos que utilizan para determinar las diferencias entre 2 versiones distintas de un mismo archivo. La parte fundamental de este trabajo será crear una plataforma en la cual el agregar nuevas formas para manejo de diferencias sea una tarea sencilla, logrando así que el manejador de

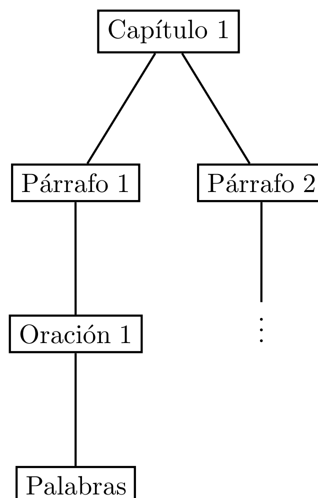


Figura 1.2: El árbol de una abstracción de bajo nivel.

versiones pueda trabajar con la mayor cantidad de tipos de archivo de la mejor manera. Además se presentará una de dichas extensiones al sistema que se encargará de aprovechar la estructura de un archivo de código fuente en Java.

Para ver cuál es el nivel de abstracción que utilizaremos en estos archivos es necesario conocer cómo es que se especifica la sintaxis de un lenguaje de programación (y otros tipos de lenguajes). En lo que resta de este capítulo revisaremos los fundamentos necesarios para esta tarea. En la sección 1.2 haremos un breve repaso de las herramientas que han existido y existen actualmente para el manejo de versiones. En 1.3 se presentan los problemas existentes en las herramiantas que se utilizan actualmente. En la sección 1.4 veremos que es lo que nosotros proponemos como una alternativa a las herramientas y métodos utilizados hoy en día. Finalmente en la sección 1.5 haremos un repaso por las herramientas y tecnología que utilizaremos para implementar nuestra propuesta, esto incluye:

1. Detección de cambios en árboles.
2. Gramáticas.
3. Reconocimiento Sintáctico.
4. Generadores de reconocedores sintácticos.

5. XML y DiffXML.

1.2. Herramientas Actuales

Actualmente existen varias herramientas que permiten el manejo de versión en archivos de código fuente. La mayoría de ellas han surgido como mejoras a herramientas anteriores tratando de agregar funciones que a los usuarios les parecían esenciales, mejorar características deficientes en las herramientas y en otros casos un cambio de filosofía. A continuación se presentan las herramientas que han sido las más relevantes.

1.2.1. *SCCS (Source Code Control System)* y *CSSC*

SCCS fue durante muchos años la única opción para el control de versiones de código fuente, hasta que llego RCS. CSSC es el remplazo de GNU para la herramienta propietaria SCCS. La página actual de CSSC es: `http://cssc.sourceforge.net/index.shtml`

1.2.2. *RCS (Revision Control System)*

Fue pensado para manejar múltiples versiones de archivos. RCS automatiza el almacenamiento, recuperación, registro, identificación y combinación de versiones. Fue desarrollado por Walter Tichy en la Universidad de Purdue al principio de la década de 1980, fue una mejora sobre Source Code Control System (SCCS). Entre las mejoras que presentó se encontraban; una interfaz más sencilla y un almacenamiento de versiones orientado a lograr un acceso más rápido. Estas mejoras las obtuvo mediante el almacenamiento de un copia completa de la versión más reciente y una archivo de diferencias con la actualización, este archivo conocido como *delta*. Utilizó las utilidades de diff de GNU para encontrar las diferencias. Se puede encontrar la última versión en: `www.gnu.org/software/rcs/rcs.html`

1.2.3. *CVS (Concurrent Versions System)*

Probablemente la más conocida de todas las herramientas. Este es el software de control de versiones que utiliza el movimiento de software libre, es bastante popular y podemos decir que es el estándar de dicho movimiento. Entre sus principales características se encuentran el uso de un modelo de cliente/servidor que permite que desarrolladores que se encuentren separados geográficamente funcionen como un equipo. Además permite el uso

de etiquetas para destacar un conjunto de archivos, y de esta forma poder acceder, por ejemplo, a una versión estable de los archivos sujetos al control de versiones. Fue uno de los primeros en diferenciar los archivos de texto de los archivos binarios. CVS tiene su página en: `https://www.cvshome.org`

1.2.4. *Aegis*

Según su propia página *Aegis* es un "sistema de manejo de configuraciones basado en transacciones". La diferencia principal entre las demás herramientas y Aegis es que esta última permite especificar ciertas condiciones a los cambios que se agregarán al repositorio. El nombre mismo del programa indica cuál fue su intención al crearlo, un escudo diseñado para proteger el código fuente funcional de los cambios que pudieran llevarlo a un estado inestable. Esta herramienta se concentra en el control de calidad del software. Se puede obtener en: `http://aegis.sourceforge.net/`

1.2.5. *Subversion*

Subversion busca convertirse en el remplazo de CVS, por lo cual la mayoría de las diferencias que tiene son cambios a este último. Entre las mejoras más relevantes se encuentran la manera en que se realizan las actualizaciones al repositorio, el protocolo que se utiliza para comunicar el servidor con el cliente y la forma de enviar los cambios en los archivos. Se puede obtener la última versión de `http://subversion.tigris.org/`

1.3. Problemas de las herramientas actuales

El mayor problema que tienen las herramientas que se usan hoy en día es la generalidad con la que tratan a los archivos. Tomemos como ejemplo a CVS, el estándar de la industria de software libre. CVS utiliza a GNU diff para calcular las diferencias entre dos archivos; este programa tiene un algoritmo basado en LCS para calcular las diferencias. Como ya mencionamos la comparación se realiza carácter por carácter por lo que si tomamos la cadena original "\n" (un salto de línea) y la cadena modificada " \n\n\n" (tres saltos de línea) puede que el significado de esta diferencia sea significativo o no. Eso depende del tipo de documento. Puede ser que en un archivo binario aparezca esta cadena, en donde seguramente representa un cambio significativo. Sin embargo si aparece en un archivo fuente entre dos funciones no importa si éstas están separadas por uno o por diez saltos de línea, ese

cambio en el texto no representa nada para nosotros. Otro ejemplo que manifiesta un problema más grave, es cuando se comparten archivos de código fuente editados en sistemas operativos diferentes. Mientras que en sistemas de la familia UNIX el separador de líneas es ``\n'' en los sistemas de Microsoft la cadena que representa a un salto de línea es ``\r \n''. Éste es un ejemplo de un cambio que un reconocedor de diferencias basado en texto detectaría, mientras que un reconocedor de diferencias semántico (como el presentado en el Capítulo 3) no lo marcaría como un cambio significativo.

Además de estos problemas tenemos el caso en el que un usuario puede modificar el programa mismo de control de versiones, esto es, permitir que el usuario conecte las partes que crea necesarias para cierto tipo de documentos. Algunas de las herramientas mencionadas anteriormente (*Subversion*) planean en el futuro incluir un sistema de *plugins* que les permita hacer precisamente eso. Nosotros proponemos un sistema similar, salvo que en este momento nosotros planeamos incluir los *plugins* del lado del servidor, mientras que en *Subversion* lo harán del lado del cliente.

1.4. Propuesta

Como ya se señaló anteriormente, este sistema de manejo de versiones pretende lograr que se tenga un mejor aprovechamiento de la metainformación y la estructura de los documentos para lograr comparaciones más significativas. Además se presentará un ejemplo en el cual se creará un manejador de diferencias para archivos de código fuente de Java 1.4. A continuación se presenta una breve discusión sobre el funcionamiento de este manejador.

En este trabajo se presenta una propuesta con un enfoque distinto al que utilizan la mayoría de las herramientas actuales; la principal diferencia radica en reconocer que los métodos utilizados hasta ahora no aprovechan la estructura de los documentos que están comparando. Como vimos en la sección anterior las herramientas actuales utilizan LCS (o algún algoritmo similar) para encontrar las diferencias entre dos textos. Esto es una de sus principales ventajas, pues puede utilizarse para comparar cualesquiera dos cadenas de texto. Pero esto también es una desventaja, pues en un contexto más específico (como es nuestro caso) hay información que se desperdicia. Esta información es la estructura específica de nuestro documento. Un documento de texto tiene una estructura jerárquica (de árbol) en la cual los elementos que la componen pueden ser capítulo, sección, título, párrafo, oración y palabra. Esto permite que la comparación sea un poco más "inte-

ligente", puesto que podemos revisar los cambios en términos más abstractos que simplemente caracteres de texto. En el caso que se estudió durante la realización de este trabajo, se escogió analizar archivos de código fuente de Java. Éstos también tienen una estructura jerárquica, diferente a los documentos de texto, que está definida por la gramática del lenguaje. Un poco más adelante revisaremos con cuidado cómo es que se define una gramática, y cómo se utiliza.

1.5. Fundamentos

1.5.1. Detección de cambios

Introducción

El problema de la detección de cambios en estructuras jerárquicas (como son los árboles con los que trabajaremos) es más útil que el de detección de cambios en archivos planos (texto simple), pero esto implica un costo mayor en la complejidad de los métodos utilizados. La detección de cambios en este trabajo se lleva a cabo mediante DiffXML 1.5.6, el cual implementa 2 algoritmos para la detección: FMES (Fast Match Edit Script.) y un algoritmo propio xmdiff. A continuación revisaremos brevemente el algoritmo FMES.

Fast Match Edit Script

Fue desarrollado por Chawathe, Rajaraman, García-Molina y Widom en la universidad de Stanford [CRGMW96]. Está diseñado para detectar cambios en datos estructurados de manera jerárquica; este algoritmo tiene cuatro características clave, las cuales son:

- Cambios en las relaciones entre nodos. Además de detectar cambios en los valores de los nodos es importante detectar también cambios en las relaciones entre éstos, por ejemplo saber si un nodo cambió de padre.

- No se asumen identificadores estables. Para lograr que el algoritmo sea lo más general posible no se asume que los nodos tengan algún tipo de identificadores (llaves) que casen información entre versiones.

- Diferencias entre una versión vieja y una nueva.

- Buen desempeño. Los algoritmos fueron desarrollados pensando en manejar grandes volúmenes de datos, por lo cual el desempeño es vital aún cuando se deban comprometer otras características. Si definimos *delta* como una transformación que lleva un árbol a otro, tenemos que este el algoritmo siempre encuentra deltas correctas, aunque no necesariamente mínimas. Existen otros algoritmos que siempre encuentran deltas mínimas pero corren en $O(n^2 log^2 n)$, entre ellos se encuentran los reportados en [ZS89].

La información jerárquica se puede representar como un árbol ordenado, un árbol en el cual cada uno de los nodos tiene un orden designado. Además, para el desarrollo de este algoritmo se toma un modelo en el cual los nodos se representan mediante una *etiqueta* y un *valor*. También se supone que cada nodo tiene un identificador único (si éste no se especifica en el árbol, el algoritmo genera uno), mismos nodos en distintas versiones del árbol no necesitan tener el mismo identificador.

Una vez establecidos estos requisitos es momento de atacar el problema de la detección de cambios entre dos árboles. Lo que se está buscando es una transformación apropiada que convierta el árbol T_1 en T_2, dicha transformación es la *delta* mencionada anteriormente. El primer paso para lograr esto es encontrar qué nodos en T_1 corresponden a nodos en T_2. La correspondencia entre nodos que tienen valores idénticos o similares es lo que se conoce como un *apareamiento*. Un apareamiento es una relación uno a uno; además se dice que un apareamiento es *parcial* si sólo algunos de los nodos en ambos árboles participan, mientras que un apareamiento es *total* si todos los nodos participan.

Así pues, el primero de los problemas que se deben resolver es encontrar un apareamiento apropiado para los árboles que se están comparando. A este problema se le llama el problema del *Buen Apareamiento* (Good Matching Problem). Al resolver este problema se intenta aparear, no sólo nodos que sean idénticos, sino también los que sean semejantes.

Cuando se tiene un buen apareamiento (parcial) M entre T_1 y T_2 el siguiente paso es encontrar una secuencia de operaciones que transformen a T_1 en un árbol T_1' que sea isomorfo[2] a T_2. Las operaciones que se realizan a T_1 incluyen *insertar* nodos, *borrar* nodos, *actualizar* valores y *mover* subárboles. Intuitivamente mientras T_1 se transforma en T_1', el apareamiento parcial M se transforma en un apareamiento total M' entre los nodos de T_1' y T_2. Esta secuencia de operaciones es lo que llamamos un guión de edición, y decimos que se conforma al apareamiento original M siempre que $M' \supseteq$

[2] Se dice que dos árboles son *isomorfos* si son idénticos salvo identificadores.

M. Ahora bien, se quiere que ese guión transforme a T_1 lo menos posible para obtener un árbol isomorfo a T_2, para obtener estas transformaciones mínimas se introduce el concepto de costo de un guión de edición. Así es que el segundo problema con el que se debe tratar es el de encontrar un guión de edición de costo mínimo, este problema se conoce como el problema del Guión de Edición Conformante Mínimo (*Minimum Conforming Edit Script*, MCES).

Minimum Conforming Edit Script

Este problema se define formalmente de la siguiente manera. Dado un árbol T_1 (el árbol viejo), un árbol T_2 (el árbol nuevo), y un apareamiento (parcial) M entre sus nodos, generar un guión de edición de costo mínimo que sea conformante con M y que transforme a T_1 en T_2. Este algoritmo empieza con un guión vacío E y conforme aplica una operación a T_1 la agrega a E. Así pues al terminar obtenemos un árbol que es isomorfo a T_2. Además, este algoritmo extiende el apareamiento parcial M en un apareamiento total, pues conforme agrega operaciones a E agrega nodos a M. A continuación se presenta una breve descripción del algoritmo, para revisarlo a fondo se puede consultar [CRGMW96]. Consideremos que los

El algoritmo se divide en cinco fases:

Actualización. En esta fase se buscan parejas de nodos $(x, y) \in M$ en las que los valores de los nodos difieran. Por cada una de esas parejas se agrega la operación de actualización a E, y se aplica esa misma operación a T_1. Al final de esta fase se ha transformado a T_1 de forma que $v(x) = v(y)$ (i.e. el valor de x es igual al valor de y) para toda pareja de nodos $(x, y) \in M$.

Alineación. Definamos el *compañero* de un nodo como el nodo al que esta apareado (con el apareamiento establecido). Supóngase que $(x, y) \in M$. Se dice que los hijos de x y y están *desalineados* si x tiene hijos apareados u y v de forma que en T_1, u está a la izquierda de v pero en T_2 el compañero de u está a la derecha de v. En esta fase se revisan todas las parejas de nodos internos $(x, y) \in M$ para revisar si sus hijos están desalineados, y de ser así se agregan operaciones de movimiento a E.

Inserción. En esta fase se busca un nodo no apareado $z \in T_2$ para el cual su padre sí está apareado. Supóngase que $y = p(z)$ (i.e. y es el padre de z) y además que el compañero de y en T_1 es x. Después se crea un

nuevo nodo w con la misma etiqueta y valor que z, y se inserta como el k-ésimo hijo de x. La posición k se determina en base a los hijos de x y z que ya han sido alineados. Esta inserción se realiza también en T_1 y se agrega la pareja (w, z) a M.

Movimiento. Se buscan parejas de nodos $(x, y) \in M$ tales que $(p(x), p(y)) \notin M$. Supóngase que $v = p(y)$. Se sabe que al final de la fase de inserción v tiene un compañero u en T_1, entonces lo que se hace es colocar a x como el k-ésimo hijo de u. Se agrega esa operación a E y se aplica a T_1. Al final de esta fase T_1 es isomorfo a T_2 excepto por nodos no apareados en T_1.

Borrado. Se buscan nodos hoja no apareados $x \in T_1$. Por cada uno de esos nodos se agrega la operación de borrado en E y se aplica a T_1. Al final de esta fase T_1 es isomorfo a T_2, E es el guión final y M es el apareamiento total al cual se conforma E.

1.5.2. Gramáticas

Otro de los elementos que debemos conocer antes de revisar la construcción de la aplicación es cómo es que reconocemos un lenguaje. Si bien éste puede parecer un problema sencillo, en una inspección más cercana nos damos cuenta de que no es así. La dificultad principal radica en encontrar una forma finita de representar conjuntos que bien pueden ser infinitos. Las personas que tengan un conocimiento básico de teoría de la computación podrán pensar en reconocer un lenguaje mediante un autómata; si bien éste es un método viable los lenguajes que se pueden reconocer de esta manera son muy limitados por lo que se necesita una herramienta más poderosa. Es aquí en donde entran en juego las gramáticas.

Definición de Gramáticas

Una gramática se compone de cuatro elementos, los cuales son:

1. Un conjunto de *símbolos terminales.*
2. Un conjunto finito de símbolos *no terminales.*
3. Un conjunto de *producciones.*
4. Una variable que representa el *símbolo inicial.*

Para obtener una frase en el lenguaje de la gramática debemos iniciar con el símbolo inicial, y mediante la aplicación de producciones reemplazamos variables pertenecientes al conjunto de no terminales por elementos del conjunto de símbolos terminales. A este proceso se le conoce como *derivación*, el cual analizaremos más a fondo un poco más adelante.

Representamos una gramática mediante sus cuatro componentes, i.e. $G = (V, T, P, S)$ en donde V es el conjunto de variables, T es el alfabeto de terminales, P es el conjunto de producciones y S es el símbolo inicial.

Veamos un breve ejemplo de una gramática que define expresiones aritméticas simples.

Sea $G = (V, T, P, S)$ con $V = \{expr, op\}$, $T = \{id, +, -, *, /, (,)\}$ y $S = expr$. Las producciones de la gramática se muestran en la figura 1.3.

$$\begin{aligned}
&expr \rightarrow expr\ op\ expr \\
&expr \rightarrow (expr) \\
&expr \rightarrow id \\
&op \rightarrow + \\
&op \rightarrow - \\
&op \rightarrow * \\
&op \rightarrow /
\end{aligned}$$

Figura 1.3: Una gramática que define expresiones aritméticas simples

Convenciones de notación

Existen varias convenciones que se suelen adoptar para no tener que especificar en cada paso qué símbolos son terminales, cuáles son no terminales y demás aclaraciones necesarias. A continuación se presentan las convenciones que utilizaremos.

1. Letras en minúsculas cerca del principio del alfabeto como *a, b, c* representan símbolos terminales. Símbolos de operadores como + y -, símbolos de puntuación y dígitos también son considerados símbolos terminales.

2. Letras en mayúsculas cerca del inicio del alfabeto como *A, B, C* son variables.

3. Letras en minúsculas cerca del final del alfabeto, como u o v son cadenas de terminales.
4. Letras griegas minúsculas como α y β son cadenas de terminales y/o variables.
5. Si tenemos las siguientes producciones de A, $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_n$, las podemos escribir como $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$

Utilizando estas convenciones la gramática de la figura 1.3 quedaría como se ve en la figura 1.4.

$$E \to E\ A\ E|(E)| - E|d$$
$$A \to +| - | * |/$$

Figura 1.4: La gramática anterior siguiendo las convenciones

1.5.3. Clasificación de gramáticas

Ahora bien, dependiendo de ciertas restricciones que podemos imponer sobre cómo son las producciones obtenemos una clasificación de gramáticas en cuatro tipos. Esta clasificación es conocida como *clasificación de Chomsky*. Las cuatro categorías que se definen son las siguientes:

Gramáticas no restringidas

En este caso no se restringe de forma alguna a las producciones. Estas pueden ser de la siguiente forma:

$$\alpha A\beta \to \gamma$$

Se permite que α, β y γ sean la cadena vacía. Por el hecho de que γ pueda ser la cadena vacía a estas gramáticas se les denomina *gramáticas contraibles*.

Gramáticas sensitivas al contexto

En este caso las producciones son de la forma:

$$\alpha A\beta \to \gamma$$

Pero se pide que γ sea distinto de la cadena vacía. Con excepción de la producción $S \rightarrow \epsilon$ (con ϵ representando la cadena vacía). Una producción $\alpha \rightarrow \beta$ tal que $\mid \alpha \mid \leq \mid \beta \mid$ se dice que es *no contraíble*. Se exige que todas las producciones (con excepción de $S \rightarrow \epsilon$) de una gramática sensitiva al contexto sean no contraíbles.

Gramáticas libres de contexto

En estas gramáticas todas las producciones son de la forma:

$$A \rightarrow \alpha,\ \alpha \in (V \cup T)^{+}\ y\ A \in N \cup \{S\}$$

Con excepción de la producción $S \rightarrow \epsilon$

Gramáticas regulares

En estas gramáticas las producciones tiene de su lado derecho a lo más 2 símbolos, uno terminal y otro no terminal. Si las producciones son de la forma:

$A \rightarrow aB\ o\ A \rightarrow a,\ A\ y\ B \in N \cup \{S\},\ a \in T$

Se trata de una gramática *lineal a la derecha*, y si son de la forma:

$A \rightarrow Ba\ o\ A \rightarrow a$

Se trata de una gramática *lineal a la izquierda*.

De las gramáticas discutidas anteriormente las que son de interés para este trabajo son las gramáticas libres de contexto; por lo que ahora nos concentraremos solo en este tipo de gramáticas. De los cuatro tipos que presenta las clasificación de Chomsky; éstas son las que mejor nos permiten representar a la mayoría de los lenguajes de programación y generar algoritmos eficientes para que dada una cadena se pueda determinar si es que pertenece al lenguaje de la grámatica y cuál es la derivación con la que se obtiene.

Derivaciones

Es mediante las producciones de una gramática que podemos inferir que ciertas cadenas pertenecen al lenguaje de una variable, pero existen dos formas de llegar a esa inferencia. La manera más común es utilizar las reglas del cuerpo a la cabeza, i.e. buscamos en la cadena subcadenas que coincidan con el cuerpo de algunas de las producciones y sustituimos a esa subcadena por la cabeza de la producción identificada. De esta manera inferimos que la cadena con la que empezamos está en el lenguaje producido a partir de la cabeza.

Existe otro procedimiento para definir el lenguaje de una gramática, en el cual se utilizan las producciones de la cabeza al cuerpo. En este método lo primero que se hace es expandir el símbolo inicial mediante una de sus producciones. Después se expanden cada una de las variables con alguna de sus producciones correspondientes, y continuamos repitiendo este proceso hasta que obtengamos una cadena formada sólo por símbolos terminales. El lenguaje definido por la gramática son todas las cadenas que se pueden formar siguiendo este procedimiento. A este uso de las gramáticas se le conoce como *derivación.*

En el proceso de derivación de cadenas se hace necesaria la introducción de un nuevo símbolo de relación $\Rightarrow$. Supóngase que $G = (V, T, P, S)$ es una gramática. Sea $\alpha A \beta$ una cadena de terminales y variables, con A una variable. Esto es α y β son cadenas en $(V \cup T)^*$, y A está en V. Sea $A \rightarrow \gamma$ una producción de G. Entonces decimos que $\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$. Si se da por sentada a G simplemente decimos que $\alpha A \beta \Rightarrow \alpha \gamma \beta$. Nótese que un paso en una derivación sustituye cualquier variable en cualquier parte de la cadena por el cuerpo de una de sus producciones. Se puede extender la relación $\Rightarrow$ para representar cero, uno o más pasos de derivación, en cuyo caso el símbolo utilizado es $\overset{*}{\Rightarrow}$.

Veamos como ejemplo una derivación de la gramática 1.4. La cadena $-(d + d)$ tiene la siguiente derivación:

$$
\begin{aligned}
E &\Rightarrow -E &(1.1)\\
&\Rightarrow -(E) &(1.2)\\
&\Rightarrow -(E\ A\ E) &(1.3)\\
&\Rightarrow -(E+E) &(1.4)\\
&\Rightarrow -(E+d) &(1.5)\\
&\Rightarrow -(d+d) &(1.6)
\end{aligned}
$$

Figura 1.5: Una derivación de la gramática 1.4

$$
\begin{aligned}
-\,(E\ A\ E) &\Rightarrow -(d\ A\ E) &(1.7)\\
&\Rightarrow -(d+E) &(1.8)\\
&\Rightarrow -(d+d) &(1.9)
\end{aligned}
$$

Figura 1.6: Una derivación alterna para la gramática 1.4

Como vemos en el ejemplo anterior hay dos elecciones que se deben tomar en cada paso de la derivación. La primera es decidir cuál es el no terminal que se va a sustituir, y la segunda con cuál de sus producciones se va a reemplazar ese no terminal. Por ejemplo, en la derivación anterior, en 1.4 podíamos haber continuado como se ve en la figura 1.6.

Aquí lo se cambió fue el orden en el que se sustituyeron los no terminales. Para poder analizar el reconocimiento sintáctico que llevan a cabo los compiladores y otras herramientas (como javacc, que utilizaremos más adelante) es conveniente restringir el número de elecciones que debemos tomar. La restricción que utilizaremos es sustituir la variable que se encuentre más a la izquierda en cada paso. Dicha derivación se conoce como una *derivación por la izquierda*. De hecho la derivación 1.6 es una derivación por la izquierda.

Árboles de parseo

Existe una representación de las derivaciones como árboles la cual es sumamente útil para problemas como el que se ataca en este trabajo así co-

mo para los compiladores. Esta representación nos proporciona de forma concisa cómo es que los símbolos de una cadena terminal se agrupan en subcadenas, en donde cada una de ellas pertenece al lenguaje de una de las variables de la gramática. El uso de esta estructura en compiladores y herramientas similares, se debe a la facilidad que representa el poder traducir el código fuente, representado por un árbol de parseo, mediante el uso de funciones recursivas.

Veamos cómo se construye un árbol de parseo; supongamos que tenemos una gramática $G = (V, T, P, S)$. Los *árboles de parseo* de G son árboles con las siguientes características:

1. Cada nodo interior está etiquetado por una variable V.

2. Cada hoja está etiquetada ya sea por una variable, un terminal, o ϵ (la cadena vacía). Si la hoja está etiquetada con ϵ entonces debe ser el único hijo de sus padre.

3. Si un nodo interior está etiquetado con A, y sus hijos están etiquetados:

$$X_1, X_2, \ldots, X_k$$

de izquierda a derecha, entonces $A \rightarrow X_1 X_2 \ldots X_3$ es una producción en P. Se debe notar que la única ocasión en la que una de las X's puede ser ϵ es si esa es la etiqueta del único hijo, y $A \rightarrow \epsilon$ es una producción de G.

El árbol de derivación de 1.6 lo podemos ver en la figura 1.7.

Existe un problema más con el que debemos tratar, y este es el de la ambigüedad. Supongamos que tenemos la cadena $1 + 2 * 3$ perteneciente a la gramática 1.4. Esta cadena tiene dos derivaciones que se muestran en las figuras 1.8 y 1.9

La diferencia radica en que en la Figura 1.8 se reemplaza la segunda E por $E * E$ en 1.14, mientras que en la Figura 1.9, se reemplaza la primera E por $E + E$ en 1.24. Los árboles de derivación se presentan en la Figura 1.10.

Si usamos estos árboles para evaluar la expresión obtenemos resultados distintos. La derivación (1.8) indica que $1 + 2 * 3$ se agrupa $1 + (2 * 3) = 7$, mientras que la derivación (1.9) indica que debemos agrupar $(1 + 2) * 3 = 9$. Es evidente que en el sentido que nosotros manejamos la aritmética la derivación correcta es la primera. Con este ejemplo en mente podemos decir que una gramática $G = (V, T, P, S)$ es *ambigua* si existe al menos una cadena

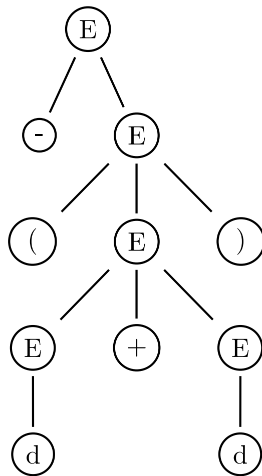


Figura 1.7: El árbol de derivación para 1.6

$$
\begin{aligned}
E &\Rightarrow E\ A\ E && (1.10)\\
&\Rightarrow E + E && (1.11)\\
&\Rightarrow E + E\ A\ E && (1.12)\\
&\Rightarrow E + E * E && (1.13)\\
&\Rightarrow d + E * E && (1.14)\\
&\Rightarrow d + d * d && (1.15)\\
&\Rightarrow 1 + d * d && (1.16)\\
&\Rightarrow 1 + 2 * d && (1.17)\\
&\Rightarrow 1 + 2 * 3 && (1.18)\\
& && (1.19)
\end{aligned}
$$

Figura 1.8: Una derivación para la cadena 1 + 2 * 3

$$
\begin{aligned}
E &\Rightarrow E\ A\ E && (1.20)\\
&\Rightarrow E * E && (1.21)\\
&\Rightarrow E\ A\ E * E && (1.22)\\
&\Rightarrow E + E * E && (1.23)\\
&\Rightarrow d + E * E && (1.24)\\
&\Rightarrow d + d * d && (1.25)\\
&\Rightarrow 1 + d * d && (1.26)\\
&\Rightarrow 1 + 2 * d && (1.27)\\
&\Rightarrow 1 + 2 * 3 && (1.28)
\end{aligned}
$$

Figura 1.9: Una derivación alterna para la cadena 1 + 2 * 3

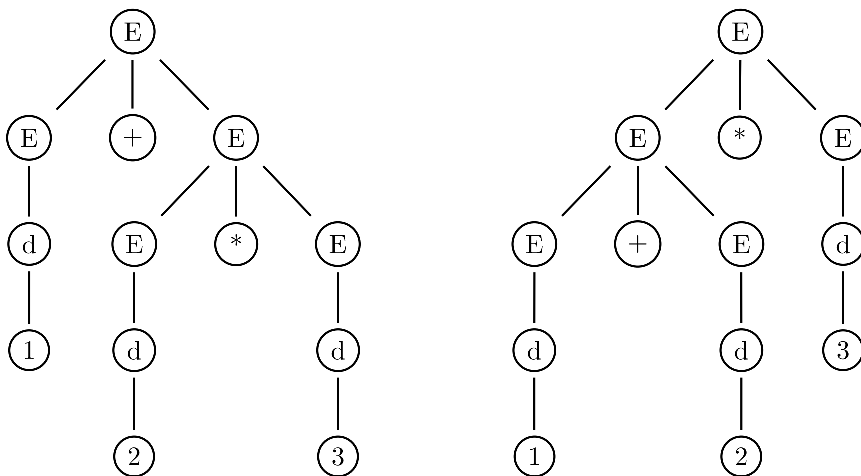


Figura 1.10: Dos árboles de derivación para la misma cadena

en T^* para la cual podemos encontrar dos árboles de análisis sintáctico distintos, ambos con raíz en S. Si todas las cadenas en T^* tienen solo un árbol de análisis sintáctico entonces decimos que la gramática es *no ambigua*.

1.5.4. Reconocimiento sintáctico

El reconocimiento de un archivo de código fuente lo llevamos a cabo mediante las herramientas JavaCC y JJTree (1.5.5). Éstas se encargan de reconocer el archivo y entregarnos un árbol de análisis sintáctico. Para ello es necesario proporcionar un archivo que contenga la gramática de Java 1.4. Es a partir de este árbol, que nosotros obtenemos un árbol de XML con el que podemos calcular las deltas para realizar actualizaciones al repositorio.

1.5.5. JavaCC y JJTree

JavaCC

JavaCC (Java Compiler Compiler) es un generador de analizadores sintácticos para el lenguaje de programación Java. JavaCC genera un analizador sintáctico para una gramática en BNF, el cual se encuentra escrito en Java. En 1996 Sun MicroSystems liberó un generador de analizadores sintácticos llamado *Jack*. Poco después los responsables de ese proyecto dejaron Sun y formaron su propia compañía llamada Metamata, y cambiaron el nombre de Jack a *JavaCC*. Metamata luego fue adquirida por WebGain, la cual después dejo de operar por lo cual JavaCC paso a su hogar actual `https://javacc.dev.java.net/`.

JavaCC presenta varias características deseables; la primera de ellas que discutiremos es el hecho de que es un analizador sintáctico descendente. La mayoría de los métodos de análisis sintáctico están comprendidos en dos clases, llamadas métodos *descendente* y *ascendente*. Estos términos hacen referencia al orden en el que se construyen los nodos del árbol de análisis sintáctico. En el primero, la construcción se inicia en la raíz y avanza hacia las hojas, mientras que en el segundo la construcción se inicia en las hojas y avanza hacia la raíz. Entre las ventajas que ofrece JavaCC sobre herramientas como YACC[3] (basadas en análisis ascendente) están el uso de gramáticas más generales, es más fácil de depurar, se puede reconocer cualquier no terminal en la gramática, y tiene la capacidad de pasar atributos hacia arriba y abajo del árbol durante la etapa de análisis. Una de las posibles desventajas que tiene JavaCC contra YACC es que estrictamente hablando existen más lenguajes que se pueden describir con gramáticas

[3] Yet Another Compiler Compiler

LALR(1) (las usadas por YACC) que con gramáticas LL(1), más aún casi cualquier problema que surge en lenguajes de programación se puede solucionar con una gramática LALR(1), y no podemos decir lo mismo de las gramáticas LL(1). Como ya se menciono antes la principal diferencia entre los tipos de analizadores es como construyen el árbol. Otra manera de ver esto es que Yacc toma sus decisiones después de haber consumido todos los terminales asociados con una elección, mientras que JavaCC toma sus decisiones antes de consumir alguno de los tokens asociados con la elección. Para solucionar este problema JavaCC puede adelantarse (lookahead) y leer en la cadena tan adelante como lo necesite (sin consumir los terminales). Esta característica permite aliviar la mayoría de los problemas que surgen al utilizar un analizador descendente.

JJTree

JJTree es un preprocesador para JavaCC. Como mencionamos en la sección 1.5.5 JavaCC es un generador de analizadores sintácticos, pero por sí mismo no genera el árbol de análisis sintáctico, por lo que necesitamos de una herramienta que le diga cómo hacerlo. Es aquí en donde entra JJTree; esta herramienta es parte de la distribución de JavaCC, y es su trabajo generar las clases necesarias para el manejo de árboles. Para hacer uso de ella es necesario pasarle el archivo de la gramática que vamos a utilizar, JJTree agrega reglas sintácticas para generar el árbol, y ese archivo de salida es el que debemos pasarle a JavaCC.

1.5.6. XML y DiffXML

XML

XML(eXtensible Markup Language) es una recomendación de la W3C para crear lenguajes de marcado[4] de propósito específico. Es un subconjunto simplificado de SGML(Standard Generalized Markup Language). Fue desarrollado por el XML Working Group (originalmente conocido como el Editorial Review Board) bajo el auspicio del World Wide Web Consortium (W3C). El trabajo fue iniciado en el año de 1996 y finalmente se concluyó con una recomendación del W3C en 1998. Los objetivos de XML son los siguientes:

1. XML debe poder utilizarse de forma directa en Internet.

[4] Markup Languages

2. XML debe soportar una gran variedad de aplicaciones.
3. XML debe ser compatible con SGML.
4. Debe ser fácil escribir programas que procesen documentos en XML.
5. El número de características opcionales en XML debe mantenerse en un mínimo absoluto, idealmente cero.
6. Los documentos en XML deben ser legibles por los humanos y razonablemente claros.
7. El diseño de XML debe ser preparado rápidamente.
8. El diseño de XML debe ser formal y conciso.
9. Los documentos de XML deben ser fáciles de crear.
10. La brevedad del marcaje de XML es de importancia mínima.

Como podemos ver de los puntos anteriores se desprenden varias de las fortalezas de XML, entre las que se encuentran compatibilidad con internet, lectura fácil sin uso de herramientas especiales, la habilidad de representar estructuras de computación básicas (árboles, listas, registros) y tiene una sintaxis estricta (a diferencia de HTML). Es por eso que el ver a un árbol de parseo como un documento XML es una representación "natural" para nuestra aplicación, pues nosotros tenemos una estructura arborescente que debe ser comparada, y transmitida a través de la red.

DiffXML

Es un trabajo creado por Adrian Mouat disponible en `http://diffxml.sourceforge.net/` que pretende proveer una herramienta Open Source que permita hacer comparaciones entre estructuras jerárquicas. Presenta dos opciones para el algoritmo que se encarga de encontrar las diferencias entre los árboles. Entre los usos que se le pueden dar a esta herramienta están el control de versiones, comparación y actualización de documentos (por ejemplo en un navegador) y en bases de datos.

Pasamos ahora a revisar la arquitectura y el modelo que se siguieron para la implementación de xCVS.

Capítulo 2

Modelo y arquitectura

2.1. Filosofía

Como se mencionó en el capítulo anterior, la base de este trabajo es crear un manejador de versiones que sea fácil de extender, en cualquier sentido y dirección. Así es que podemos agregar objetos que traten de forma distinta archivos que son fundamentalmente diferentes y como una consecuencia de esto sea más sencillo aprovechar la meta-información de los documentos que se comparan.

A lo largo de este capítulo se presentará la arquitectura del servidor y del cliente. A través de las subsecciones se presentarán las diferentes capas que conforman el sistema y se expondrán los motivos y consideraciones que se tomaron en cuenta para llegar al diseño final, así como posibles rutas alternas que finalmente fueron descartadas.

Se describirá a detalle la base de datos que está detrás del repositorio, cuáles son las tablas que se utilizan, los objetos a los que se traducen y las peticiones que se utilizan a través de las instrucciones del servidor.

Se contemplarán también las medidas de seguridad adoptadas y su implementación, así como el uso de *plugins* para diferentes tipos de archivos. En las últimas secciones se presentará el API [1] que se provee para generar futuras expansiones; finalmente se presentarán ejemplos de uso para familiarizar al lector con el sistema.

2.2. El servidor

El servidor está dividido en dos capas, la primera es la capa de persistencia implementada mediante una base de datos (PostgreSQL) y la segunda

[1] Application Program Interface.

es una capa de servicios que contiene las instrucciones con las que los clientes podrán comunicarse con el servidor. Para poder apreciar cómo es que está implementado el servidor primero será necesario revisar cuáles son los bloques básicos con los que opera xCVS.

2.2.1. Elementos básicos del servidor

Lo primero que es necesario para entender el funcionamiento del servidor es el modelo con el que se va a trabajar. Para esto analizaremos los objetos (objetos como unidades, no en el sentido de la programación Orientada a Objetos) con los que operamos y cuáles son las funciones que desempeña cada uno de ellos.

El objeto de más alto nivel dentro del manejo de revisiones es el de un proyecto particular, o en la jerga de xCVS un *módulo*. Un módulo está formado por una o varias ramas, las cuales a su vez contienen los archivos que los usuarios modifican. Así pues cada proyecto que es parte del repositorio dentro de xCVS lo podemos ver como una estructura jerárquica como la que se ve en la figura 2.1

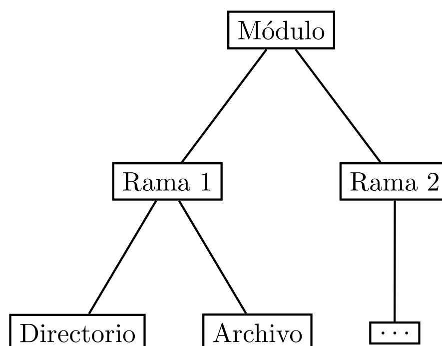


Figura 2.1: Estructura de un módulo en xCVS

xCVS no establece ninguna restricción sobre lo que debe contener cada una de las ramas; bien puede ser que ambas ramas no tengan ningún archivo en común, aunque en la práctica generalmente comparten varios archivos. La información necesaria para crear un módulo es muy poca (esto es información propia del módulo). Soló necesitamos un par de datos:

- El identificador del usuario que será el dueño del módulo.
- El nombre del módulo.

Cada módulo puede estar en varios estados al mismo tiempo. Por ejemplo, puede ser necesario tener una versión del proyecto que esté lista para producción y ser entregada a clientes, mientras que también es necesario tener copias de los archivos en las que se reparan errores que se van encontrando y tal vez una más en la que se agregan nuevas características al programa. Todas estas versiones pertenecen al mismo proyecto y como tal deben estar relacionadas de alguna forma. Esto es, todas estas versiones son *ramas* del mismo proyecto. En xCVS cada módulo tiene al menos una rama la cual se conoce como el tronco del módulo. Esta rama es la que se genera cuando se crea por primera vez el módulo. Conforme vaya siendo necesario durante la evolución del proyecto, se pueden crear nuevas ramas con las versiones actuales del proyecto. A este proceso generalmente se le conoce como una *bifurcación*. Supongamos que se tiene un módulo muy simple en el que la estructura de directorios es la de la figura 2.2.

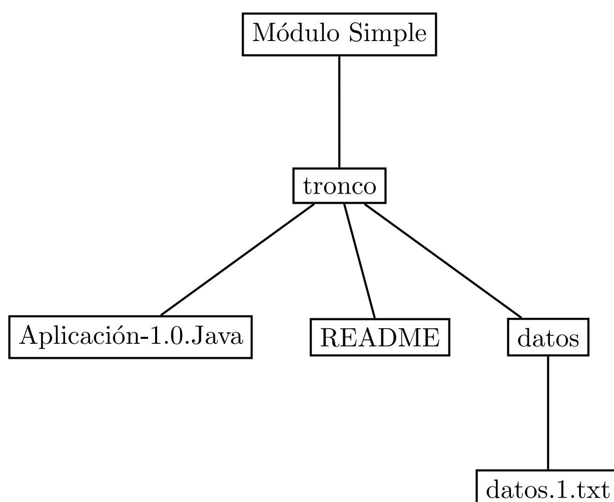


Figura 2.2: Un módulo muy simple

Veamos un ejemplo de lo mencionado anteriormente. En algún momento del desarrollo de esa aplicación se decide que es necesario implementar nuevas características y documentarlas en el archivo `README`. Así pues se decide crear una nueva rama en el módulo, en la que se realizarán las modificaciones necesarias, mientras que en la rama actual no se realizará ningún cambio más y se mantendrá como la versión estable de la aplicación. Así ahora el módulo se ve como en la figura 2.3.

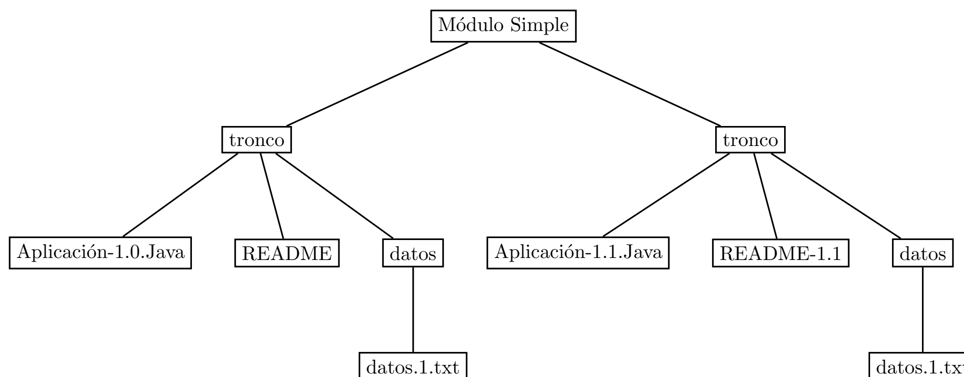


Figura 2.3: Un módulo en desarrollo

Al igual que con los módulos es poca la información que se requiere para crear una nueva rama dentro de xCVS:

- El identificador del módulo al que pertenece la rama.
- El nombre de la rama.

Finalmente, dentro de las ramas es dónde están los objetos centrales del manejo de versiones, los archivos. Necesitamos tener objetos que reflejen de una forma fidedigna los archivos en el sistema operativo de la máquina en la que se está realizando la edición de archivos. En xCVS el repositorio mantiene una tabla en la base de datos, la cual contiene los archivos que pertenecen a un módulo. En este caso la información necesaria para crear un nuevo archivo es mucho mayor a la de los dos casos anteriores:

- El identificador del usuario que es dueño del archivo.
- A qué rama pertenece.
- Cuál archivo es su padre en la jerarquía de directorios.
- Su manejador de diferencias.
- Su filtro de entrada.
- Su filtro de salida.
- Su tipo MIME [2].

[2]Multipurpose Internet Mail Extensions.

- Su nombre.
- Los datos del archivo.

En la descripción que se ha hecho hasta el momento sobre los objetos con los que trabaja xCVS se ha mencionando repetidamente el concepto de usuario; veamos ahora con mayor detalle cómo funciona el esquema de autenticación en el sistema.

Para el esquema de autenticación de xCVS se intentó presentar un sistema de permisos similar al encontrado en sistemas UNIX, si bien el nuestro es mucho más pequeño y sencillo, dado el hecho de que nosotros no nos enfrentamos a todos los problemas que debe resolver un sistema operativo. Central a este esquema es el concepto de *usuario*. Un usuario es simplemente una persona (o un programa) al que se le conceden ciertos permisos para acceder a algún recurso. En xCVS a un usuario se le pueden conceder los siguientes tres tipos de permisos:

- Permiso para crear un nuevo grupo.
- Permiso para crear un nuevo usuario.
- Permiso para crear un nuevo módulo.

El primero de estos permisos hace que sea necesario definir otro de los elementos con los que trabaja xCVS. Hay ocasiones en las que es más fácil otorgar permisos a un conjunto de desarrolladores que hacerlo individualmente con cada uno de ellos. Por ejemplo podemos tener un equipo de usuarios de prueba de una aplicación y queremos que todos ellos se rijan bajo un mismo conjunto de permisos, digamos todos ellos deben ser capaces de leer pero no de modificar los archivos. Los *grupos* nos permiten hacer eso precisamente, son simplemente un conjunto de usuarios que mantienen un conjunto de permisos. A un grupo se le otorgan permisos sobre una de las ramas de algún módulo existente. Es importante señalar que los permisos sobre una rama también pueden ser otorgados a un usuario. Los permisos que se pueden otorgar sobre una rama son los siguientes:

- Permiso para leer los archivos de la rama.
- Permiso para agregar archivos al contenido de la rama.
- Permiso para crear una rama nueva a partir de esa.
- Ser declarado el dueño de una rama.

Cabe ahora una aclaración cómo funciona el sistema de permisos. Para que un usuario pueda otorgar cualquiera de los permisos mencionados anteriormente es necesario que el mismo tenga ese permiso. Cuando se instala el sistema por vez primera éste se encarga de crear un usuario con características especiales (él mismo se otorga todos los permisos). Este usuario especial o super usuario es el encargado de crear a los primeros usuarios de uso común. Así pues cada vez que a un usuario se le otorga un permiso, el sistema guarda un registro de quién se lo dio. Esto permite que si en algún momento a un usuario se le revoca uno de sus permisos todos los usuarios que obtuvieron un permiso igual a partir de él también lo pierdan.

Los últimos objetos a los que se les puede conceder permisos son los archivos. Éstos son los elementos con permisos más básicos. Los siguientes permisos se pueden otorgar ya sea a un usuario o a un grupo.

- Permiso para leer un archivo.
- Permiso para escribir (modificar) un archivo.

Cabe señalar que en todos los permisos descritos anteriormente no se menciona ningún permiso para borrar algún elemento; eso es porque en xCVS el único usuario (o grupo) que puede borrar un elemento es el mismo que lo creó, esto es, el que está marcado como su dueño.

2.2.2. Capa de Instrucciones

Ahora que se tiene una idea general de los elementos que componen al servidor podemos ver cuáles son las instrucciones que se brindan al programador para poder crear, modificar y eliminar estos elementos. La capa de servicios para interactuar con el servidor nos presenta dos tipos de instrucciones. La primera categoría de instucciones son las que podriamos denominar de nivel *administrativo*. Estas son todas las instrucciones con las que podemos modificar el estado del servidor; podemos dar de alta nuevos usuarios, crear grupos o tal vez asociar algunos archivos a nuevos manejadores. El segundo tipo de instrucciones son las de nivel *usuario*, es mediante estas que podemos utilizar xCVS para el manejo de diferencias; permiten obtener copias locales, actualizarlas o enviar cambios al repositorio.

Estas instrucciones están contenidas en el paquete `mx.unam.fciencias.conexa.xcvs.commands`, todas las clases dentro de este paquete están basadas en el patrón de conducta *Command* [GHJV95].

Mediante este patrón obtenemos varias ventajas en el diseño del servidor. Hay ocasiones en las que un objeto necesita hacer una petición a otro

objeto sin conocer los detalles sobre cómo se realiza la misma y una buena forma de hacer esto es encapsular esa petición en un objeto. La forma más común de implementar este patrón es haciendo uso de una clase abstracta (en Java generalmente se utiliza una interfaz), y con ella podemos separar la implementación concreta de la función de los objetos que la realizan, como se muestra en la figura 2.4. Al encapsular las funciones dentro de objetos obtenemos varias ventajas, en nuestro caso agregamos varias de estas instrucciones a una transacción y luego la llevamos a cabo, permitiendo hacer *rollback* en caso que alguna de las llamadas a las funciones no pueda ser completada exitosamente. Otra ventaja es que se puede tener una bitácora exacta de en qué orden se realizaron las operaciones. Finalmente, si cada uno de los objetos es capaz de realizar una operación, se podría incluir dentro de ese objeto una función capaz de revertir lo realizado y si se almacenan los parámetros con los que se ejecutó la operación podríamos revertir un conjunto de instrucciones.

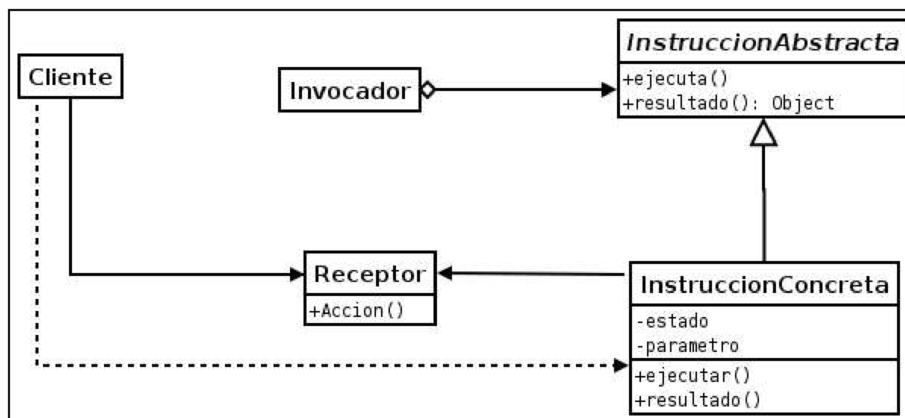


Figura 2.4: El patrón Command

La interfaz mencionada anteriormente con la que se cumple con el patrón *Command* es la clase de nombre `AbstractCommand`. Esta interfaz contiene los siguientes métodos:

- `public abstract Object execute () throws IllegalStateException, XcvsException`
  Es la encargada de llevar a cabo la instrucción, si el objeto no contiene parámetros validos deberá lanzar una `IllegalStateException`, si la función no puede completarse exitosamente entonces se deberá lanzar

una excepción de tipo `XcvsException`.

- `public Object getResult() throws IllegalStateException`
  Deberá regresar el valor que resultó después de ejecutar la instrucción, o `null` en el caso en el que la instrucción no establece ningún valor como resultado. En caso de que se intente obtener el resultado antes de ejecutar la instrucción (mediante el método execute) el objeto deberá lanzar una excepción de tipo `IllegalStateException`.

2.2.3. Instrucciones a nivel usuario

Como ya mencionamos anteriormente las clases concretas dentro de este paquete podemos dividirlas en dos categorías, la primera de ellas contiene todos las instrucciones encargadas de modificar el estado de un módulo y su copia local. Este conjunto contiene las siguientes instrucciones:

Checkout

Esta es la instrucción encargada de obtener una copia local de alguna rama dentro del repositorio, con el fin de obtener los archivos para editarlos y mantenerlos bajo control de versiones. Para realizar esto es necesario generar una estructura de directorios en la máquina del cliente, que sea igual a la contenida dentro de la base de datos. Además de estos archivos es necesario crear archivos de control para xCVS los cuales le permitirán conocer el estado de los archivos en la copia local. Más adelante detallaremos cómo funcionan estos archivos de control.

Copy

Debe copiar un archivo dentro de la base de datos o otro directorio; se conservan todos los permisos y atributos del archivo original, con la excepción del dueño. El dueño del nuevo archivo es el usuario que realizó la copia.

Export

Esta instrucción es casi idéntica a *Checkout*, con la diferencia de que no genera los archivos de control de xCVS. Es la instrucción utilizada cuando se quiere distribuir una copia de alguna rama pero no se espera que esa copia se mantenga bajo control de versiones.

History

Es la encargada de presentar al usuario la evolución de un archivo desde que es sujeto al control de versiones. Presenta un listado de los cambios que ha sufrido el archivo, incluyendo:

- El nombre del usuario que realizó el cambio.
- La fecha en la que se llevó a cabo.
- Un comentario que describe los cambios que ocurrieron.

Move

Mueve un archivo (o directorio) de un directorio a otro, es necesario que el usuario que realiza esta operación tenga permisos de escritura en el directorio destino y ser el dueño del archivo origen (para poder borrarlo después de realizar la copia). Al ejecutar esta instrucción el nuevo archivo preserva todos sus campos, en contraste con otros manejadores como CVS, en donde una operación de este tipo implica perder la historia del archivo.

Rename

Permite que se le cambie el nombre a un archivo dentro de la base de datos. Para que un usuario pueda llevar a cabo esta operación es necesario que sea el dueño del archivo que se va a renombrar.

Tag

En ocasiones es necesario mantener una etiqueta a varios archivos para poder tener una versión estable del conjunto de archivos que están bajo el control de versiones. Esto implica que mediante un solo nombre accedemos a cierto conjunto de archivos. En xCVS esto se logra mediante el comando *Tag*. Este comando se encarga de crear una nueva rama que contiene los archivos que queremos conocer bajo cierto nombre. Se provee principalmente para la comodidad de usuarios de otros manejadores de versiones.

Es importante señalar que de las instrucciones anteriores las que modifican la estructura de directorios y archivos (i.e. *Copy, Move, Rename*) también generan deltas. Esto es importante pues si por alguna razón un usuario cambia de lugar un archivo (en su copia local) es importante que los demás usuarios de ese módulo también se enteren de ese cambio. Si bien esta operación podría llevarse a cabo eliminando el archivo de un directorio y agregándolo en el directorio destino hay una diferencia fundamental en

el resultado de ese proceso. El archivo nuevo no tiene historia alguna, para el manejador de versiones es un archivo completamente desconocido, aun cuando los usuarios saben que se trata del mismo archivo. Al realizar esta operación mediante la instrucción *Copy* mantenemos toda la historia que el archivo había acumulado previamente.

2.2.4. Instrucciones a nivel administrativo

Estas instrucciones son las encargadas de crear, eliminar o modificar entidades dentro del sistema. Estas entidades pueden ser objetos, grupos, tipos MIME, etc. Todas las modificaciones que se hacen a las entidades almacenadas en la base de datos son mediante funciones implementadas en el lenguaje `PL/pgSQL` de PostgreSQL.

Funciones encargadas de los usuarios

Crear usuario: Permite agregar nuevos usuarios al sistema; el usuario que invoca la función es asignado como el *creador* del nuevo usuario.

Eliminar usuario: Permite eliminar un usuario del sistema. Sólo el usuario que es el creador de este usuario puede eliminarlo.

Actualizar usuario: Permite modificar los valores de un usuario como lo son el nombre y los permisos intrínsecos que tiene asociados (crear usuarios, crear grupos y crear módulos). El único usuario que puede hacer estas modificaciones es el creador del usuario.

Funciones encargadas de los permisos para los usuarios

Para el manejo de los permisos de los usuarios se proveen funciones para otorgar o revocar cualquiera de los permisos mencionados anteriormente.

- Funciones para conceder permisos:
  - auth.grant_add_user
  - auth.grant_add_group
  - auth.grant_add_module
- Funciones para cancelar permisos:
  - auth.revoke_add_user
  - auth.revoke_add_group
  - auth.revoke_add_module

Funciones encargadas de los grupos

Crear grupo: Permite crear nuevos grupos de usuarios en el sistema. Cuando un usuario crea un grupo se convierte en su dueño, además es el agregado como el primer miembro del mismo. Él es el único usuario del sistema que más adelante podrá agregar nuevos miembros al grupo.

Eliminar grupo: Permite eliminar un grupo del sistema. Sólo el usuario que creó el grupo (y por lo tanto es el dueño) puede realizar esta operación.

Actualizar grupo: Permite modificar el nombre asignado a un grupo.

Agregar un usuario a un grupo: Permite agregar a un usuario como miembro de un grupo. El usuario que intente realizar esta operación debe contar con el permiso adecuado. Además deberá indicar si este nuevo usuario puede a su vez agregar nuevos miembros al grupo.

Eliminar un usuario de un grupo: Permite quitarle a un usuario la membresía a un grupo. Solamente el usuario que lo agregó inicialmente al grupo puede realizar esta operación.

Funciones encargadas de archivos

Crear archivo: Permite crear un nuevo archivo en el sistema; es necesario indicar cuáles son los manejadores por omisión (de diferencias, de entrada y de salida) que se asociarán a este archivo. Para poder crear un archivo dentro de un módulo es necesario tener permisos ya sea como usuario o bien pertenecer a un grupo que tenga los permisos adecuados. El usuario que invoca esta función es el dueño del archivo recién creado y tiene permisos de lectura y escritura.

Eliminar archivo: Permite eliminar un archivo del repositorio; si este archivo es un directorio entonces se borran los archivos dentro de él de forma recursiva.

Actualizar archivo: Permite modificar los valores de un archivo, como lo son el nombre del archivo, cualquiera de sus tres manejadores (diferencias, entrada, salida), el tipo MIME o los datos del archivo. Es necesario que el usuario que invoca la función tenga permisos de escritura.

Ramificar un archivo: Mediante el uso de esta función es posible ramificar un archivo, esto es, nos permite copiar los datos y todos los valores asociados a un archivo y colocarlos en un nueva rama de desarrollo. El usuario que ejecuta esta operación es el dueño de la nueva copia y tiene todos los permisos de lectura y escritura sobre él.

Funciones encargadas de permisos para archivos

Para el manejo de los permisos de archivos se tienen las siguientes funciones, utilizadas para otorgar o revocar los permisos que hemos mencionado anteriormente.

- Funciones para usuarios:
  - xcvs.grant_user_file_can_read
  - xcvs.grant_user_file_can_write
- Funciones para grupos:
  - xcvs.grant_group_file_can_read
  - xcvs.grant_group_file_can_write

Funciones encargadas de manejadores de entrada/salida

Las siguientes funciones nos permiten manejar la creación y eliminación de manejadores de entrada y salida. Además de actualizar los manejadores.

Crear manejador: Permite la creación de un manejador de entrada/salida. Es necesario proveer el nombre del manejador así como el nombre de la clase que es el manejador per se. Es necesario que esta clase, así como el archivo jar que la contiene, sean agregados a la base de datos antes de utilizar esta función.

Actualizar manejador: Permite cambiar el nombre o la clase que se utiliza para comparar o actualizar archivos.

Eliminar manejador: Permite eliminar un manejador de la base de datos.

Funciones encargadas de tipos MIME

Estas funciones permiten manipular los tipos MIME reconocidos por xCVS, los cuales son utilizados para determinar los manejadores (entrada, salida y diferencias) que se usarán para un archivo particular.

Crear tipo: Encargada de la creación de un nuevo tipo MIME; es necesario proveer el nombre y los manejadores (entrada, salida y diferencias) que se asociarán a él.

Actualizar tipo: Permite cambiar el nombre de tipo MIME, además de los manejadores que tiene asociados.

Eliminar tipo: Permite eliminar un tipo de la base de datos; sólo es necesario proveer el identificador del tipo a eliminar.

Funciones encargadas de extensiones de tipos MIME

Es mediante las extensiones que xCVS determina el tipo MIME de un archivo, por lo que es necesario asociar un tipo MIME a las extensiones que identifican a un archivo.

Crear extensión: Encargada de crear una nueva extensión asociada a un tipo MIME; es necesario proveer la extensión como una cadena de texto (por ejemplo ".txt") y el tipo MIME al que se asociará.

Eliminar extensión: Permite eliminar una extensión asociada a un tipo MIME; para esto sólo es necesario que la función reciba cuál es la extensión que debe eliminar.

Actualizar extensión: Permite cambiar los valores de una extensión, como lo son la cadena asociada a la extensión, o el tipo MIME al que está asociada la extensión.

Funciones encargadas de ramificaciones

Crear una rama: Permite la creación de nuevas ramas dentro de un módulo. La información necesaria se compone de el nombre de la nueva rama, el identificador de la rama de la cual se originará y un booleano que indica si esta rama es o no una etiqueta.

Eliminar una rama: Permite eliminar una rama de la base de datos; sólo es necesario el identificador de la rama que se desea borrar.

Funciones encargadas de los permisos de las ramas

- Funciones para conceder permisos a usuarios:
  - grant_user_branch_can_read(int, int, boolean)
  - grant_user_branch_can_add(int, int, boolean)
  - grant_user_branch_can_branch(int, int, boolean)
  - grant_user_branch_can_owner(int, int, boolean)
- Funciones para conceder permisos a grupos:
  - grant_group_branch_can_read(int, int, boolean)
  - grant_group_branch_can_add(int, int, boolean)
  - grant_group_branch_can_branch(int, int, boolean)
  - grant_group_branch_can_owner(int, int, boolean)

Funciones encargadas de manejadores de diferencias

Las siguientes funciones nos permiten manejar la creación y eliminación de manejadores de diferencias. Además nos permiten actualizar los valores con los que trabajan

Crear manejador: Permite la creación de un manejador de entrada/salida. Es necesario proveer el nombre del manejador así como el nombre de la clase que es el manejador per se. Es necesario que esta clase, así como el archivo jar que la contiene sean agregados a la base de datos antes de utilizar esta función.

Actualizar manejador: Permite cambiar el nombre o la clase que se utiliza para comparar o actualizar archivos.

Eliminar manejador: Permite eliminar un manejador de la base de datos.

Funciones encargadas de módulos

Estas son las funciones encargadas de la creación, eliminación o modificación de los módulos, una de las partes fundamentales de xCVS.

Crear módulo: Permite la creación de un nuevo módul;, para esto sólo es necesario el nombre que éste llevará.

Eliminar módulo: Permite eliminar un módulo de la base de datos; nuevamente sólo es necesario el nombre del módulo.

Funciones encargadas de los miembros de un grupo

Agregar miembro: Agrega un usuario a un grupo; es necesario proveer el identificador del usuario y el identificador del grupo. Además es necesario indicar si el nuevo miembro tendrá el permiso para él a su vez agregar nuevos usuarios.

Eliminar miembro: Elimina a un usuario de un grupo; es necesario el identificador de usuario y el identificador de grupo.

Como ya mencionamos antes, todas estas instrucciones están basadas en el patrón *Command*, así que veamos cómo es que funcionan todas ellas. Antes de ahondar más en este proceso es necesario revisar otra parte fundamental en la arquitectura de xCVS. El paquete `xfcdb-DBI`. Este paquete desarrollado en la facultad de Ciencias como parte del proyecto XFC provee a programas en Java una interfaz con una base de datos. Si bien permite el uso de varios objetos que cumplen con el estándar de JavaBeans para la modificación de los datos, xCVS no hace uso de esa funcionalidad; los datos son cambiados directamente haciendo uso de `JDBC`. Hacemos uso de `xfcdb-DBI` para tener acceso desde Java a las funciones en `PL/pgSQL` que llevan cabo las operaciones centrales de xCVS. Así este paquete se encarga de la conversión de tipos entre `PL/pgSQL` y Java, pasando digamos todos los argumentos de tipo `bytea` en `PL/pgSQL` a un arreglo de bytes `byte[]` en Java. Este proceso es realizado de manera automática lo cual libra al programador de una tarea bastante tediosa; cuando se agrega una nueva función a la base de datos el programador tendría que crear una función en Java que se encargará de hacer la llamada adecuada mediante JDBC a la base de datos. En lugar de eso basta con ejecutar las clases necesarias de `xfcdb-DBI` y esto generará las interfaces y clases concretas para poder utilizarlas en las clases propias de xCVS. Este proceso tampoco tiene porqué realizarlo el programador a mano, basta con ejecutar el objetivo `compile` del archivo build.xml incluido en la distribución de xCVS para que todo eso ocurra. Éste es el primero de dos paquetes (el otro es JavaCC que revisaremos con calma más adelante) que se utilizaron en el desarrollo de xCVS que reflejan una excelente idea, librar al programador de tareas largas y tediosas, permitiendo que la computadora genere el código necesario.

Sabiendo pues que mediante las interfaces DBI de `xfcdb-DBI` es como se accede a las funciones en nuestra base de datos, revisemos cual es el proceso que sigue cada una de las instrucciones que hemos mencionado. Dentro del método `execute()` de cada una de estas instrucciones lo primero que se realiza es un chequeo sobre los parámetros con los que se intentará modificar

la base de datos; todos estos parámetros son objetos de Java (i.e. no son primitivos), para así poder propagar el valor de `null` a la base de datos. Se revisa que los valores fundamentales sean válidos (que estén establecidos) y finalmente se propaga la llamada al DAO; éste realiza cualquier operación (lógica de negocios) que sea necesaria con los datos y finalmente hace la llamada adecuada al paquete `xfcdb-dbi`.

La forma a *grosso modo* en la que operan todas estas funciones es la siguiente:

1. El cliente crea la nueva instrucción.
2. Se establecen los parámetros necesarios para llevarla a cabo.
3. El cliente agrega la instrucción a una transacción.
4. Se ejecuta la transacción y por lo tanto la instrucción.
5. La instrucción propaga la llamada al DAO.
6. El DAO realiza lógica de negocios y propaga la llamada al DBI.
7. El DBI ejecuta la función almacenada dentro de la base de datos.

En la figura 2.5 se muestra un diagrama de secuencia, en el que algún cliente intenta crear un nuevo usuario (sin el uso de transacciones).

2.3. Repositorio

2.3.1. La base de datos

Para lograr una separación clara entre la capa que contiene la lógica de la aplicación y la capa encargada de la persistencia de datos se utilizó el patrón de "Data Access Object (DAO)" [CW03] como se muestra en la figura 2.6 . Haciendo uso de este patrón podemos separar el código que accede a la base de datos del código que hace uso de los datos. De esta manera el resto de la aplicación delega toda la responsabilidad del acceso a la base de datos al DAO. [3]

El DAO permite pues que si en algún momento se decide hacer uso de una base de datos distinta a la proporcionada por omisión (PostgreSQL) el cambio es lo más sencillo posible, incluso podría hacerse uso de un método de

[3] En realidad nuestro DAO no es un DAO puro pues lleva a cabo operaciones que contienen lógica de negocios, principalmente validación.

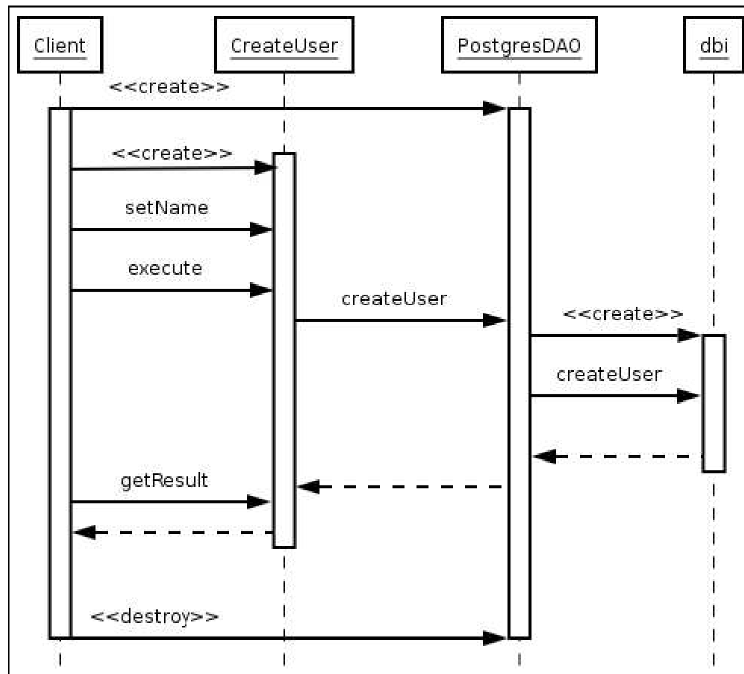


Figura 2.5: Creando un nuevo usuario

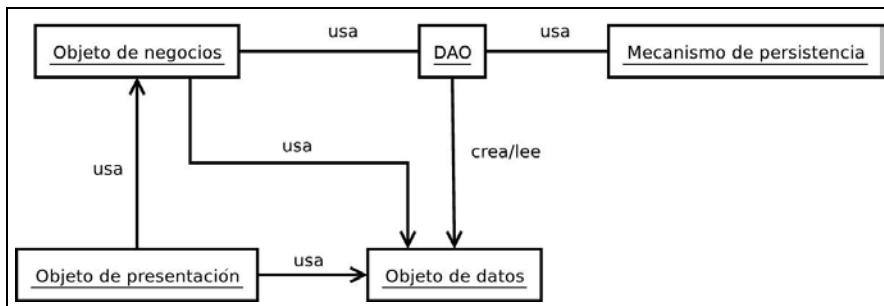


Figura 2.6: El patrón DAO

persistencia completamente distinto como un sistema de archivos (aunque esta última opción implicaría que el programador tendría que lidiar con problemas de concurrencia).

La primera aproximación hacia la capa de persistencia en el desarrollo de xCVS, fue crear un DAO que permitiera hacer uso de cualquier mecanismo de persistencia, para lograr mayor independencia del sistema, pero esta opción fue abandonada pues el compromiso costo-beneficio entre utilizar una tecnología probada como lo son las bases de datos, en contra de una tecnología desarrollada en casa, la cual en un futuro pudiera ser modificada por la comunidad, requeriría de un DAO demasiado general. Esto implica que no se podrían presuponer características tales como un sistema de transacciones adecuado, o la integridad de los datos (aunque esta última tampoco puede ser garantizada en las bases de datos, pues depende de las tablas, el sistema nos asegura que con las tablas adecuadas se mantiene la información correcta).

Finalmente se decidió mantener el patrón del DAO, pero en un nivel más concreto aplicado solamente a las bases de datos relacionales. Para la presentación final de este trabajo se creó una interfaz `Repository` que contiene los métodos que un objeto necesita implementar para interactuar con la base de datos y una clase concreta `PostgresDAO` que como su nombre lo indica está implementada para trabajar con el manejador de bases de datos *PostgreSQL* (>= 8.0).

La base de datos esta dividida en dos esquemas, uno encargado de la autorización del sistemas (usuarios y grupos) y otro encargado de la persistencia de las modificaciones. A continuación se presenta una vista rápida de estos esquemas, si se desea verlos a un nivel más técnico se puede consultar el Apéndice A.

2.3.2. El esquema de autorización

El esquema de autorización (`auth`, en la base de datos) es el encargado de permitir el acceso de un usuario a las funciones del sistema ya sea para llevar a cabo funciones a las que la mayoría del público tiene acceso (como podría ser el obtener una copia local de algún módulo) o tal vez alguna función administrativa (crear un nuevo usuario o grupo). Para realizar cualquier función dentro de xCVS es necesario ser un usuario registrado e iniciar una sesión. Al instalar el sistema se crea un usuario a partir del cual se deben crear los demás usuarios y grupos. Dentro de xCVS un usuario solo se puede tener una sesión abierta en un momento dado, con lo cual se evitan problemas de seguridad. Después de que un usuario inicia una sesión se le

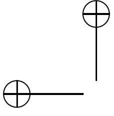
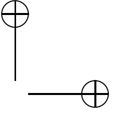
regresa una llave con la cual puede autenticarse ante al sistema. Si otro usuario inicia una sesión es sólo mediante esta llave que el usuario anterior podrá recobrar su sesión.

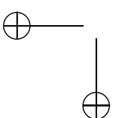
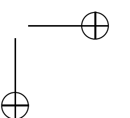
2.3.3. El esquema de trabajo

Este esquema (`xcvs` en la base de datos) contiene la mayoría de las tablas con las que trabaja un usuario comúnmente. Aquí se encuentran los archivos, los tipos MIME y sus extensiones, los manejadores, módulos y ramas. Si se desea hacer alguna extensión a xCVS los más probable es que sea aquí en donde se deban realizar los cambios. Aquí se encuentran también todas las funciones encargadas de los cambios a estas tablas, esto quiere decir que la mayoría de las instrucciones de xCVS terminarán haciendo uso de alguna ahí definida.

2.4. Excepciones

El manejo de excepciones intenta ser lo mas fino posible; así pues dentro del paquete `mx.unam.fciencias.conexa.xcvs.exceptions` se tienen clases que se mapean a los posibles errores que ocurran mientras se intenta ejecutar una instrucción. Esto es si mientras se ejecuta una función en `PL/pgSQL` esa excepción es recuperada en el DAO como una `SQLException` y dentro de ese método se envuelve en una excepción de Java más significativa. Por ejemplo, supongamos que un usuario intenta crear un nuevo manejador de entrada/salida, pero el nombre que quiere utilizar ya existe dentro de esa tabla. Esto violaría los requerimientos en la base de datos, por lo que se genera una `SQLException`. En vez de propagar esta excepción, ésta se envuelve en una `CreateIOHandlerException` (preservando el stack de la excepción de SQL) y ésta se lanza en su lugar. De esta forma las capas superiores (clientes) que utilicen al DAO pueden atraparlas y lidiar con ellas de una forma adecuada.

Capítulo 3

El manejador sintáctico

En este capítulo veremos de manera exacta cómo es que se construyó el manejador sintáctico para archivos de Java. En la sección 3.1 veremos cómo es que se lleva a cabo la transformación de un árbol de parseo, como los que vimos en la introducción, a un árbol en Java que represente la información relevante para nosotros. En la Sección 3.2 veremos los pasos necesarios para poder escribir los árboles en un archivo XML. Posteriormente en 3.3 revisaremos la información que es modificada por los dos pasos anteriores.

3.1. Transformación de un árbol de parseo a un JavaTree

Como primer paso para construir el árbol es necesario tener una estructura jerárquica; para este trabajo no se necesita ninguna característica especial para el árbol así que no se presentará a fondo la implementación que se utilizó. Basta con decir que se creó una clase `CollapsedTreeNode` que representa un nodo del árbol.

La transformación de un árbol de parseo de Java 1.4 a un árbol que nosotros utilizamos para representar el código está definida por los símbolos no terminales en la gramática utilizada para construir el árbol y por la información que éstos representan para nosotros. Es por ello que el árbol que nosotros construimos, al que llamamos *JavaTree*, es una versión colapsada del árbol de parseo completo. Por colapsada nos referimos a que muchos de los nodos hijos transfieren su información a su padre y desaparecen. Esto es necesario hacerlo porque muchos de los nodos en la gramática no aportan información relevante para el cálculo de diferencias. Por ejemplo en Javacc, la herramienta utilizada para generar el reconocedor sintáctico de Java, no existe el concepto de precedencia para los símbolos no terminales. Esto hace

que sea imposible decirle a Javacc que el operador aritmético `*` tiene una precedencia más alta que `+`. Esto ocasionaría que la evaluación de operaciones aritméticas no fuera la adecuada. Para evitar eso, la precedencia de operadores es establecida mediante las producciones. En el caso anterior las producciones encargadas de expandir las operaciones aritméticas expanden primero el símbolo `*` y posteriormente `+`. Así pues un subárbol que represente una operación aritmética de multiplicación y suma se ve como en la figura 3.1. Es decir para conseguir que se evalúe primero la multiplicación tenemos una producción para la cual su cabeza es la suma y su cuerpo es la multiplicación. Es claro que para xCVS estos nodos que utilizados para representar la precedencia no modifican el cálculo de diferencias, para nosotros sería exactamente igual si toda la expresión estuviera contenida en un solo nodo, es decir podemos colapsar ese subárbol en un solo nodo.

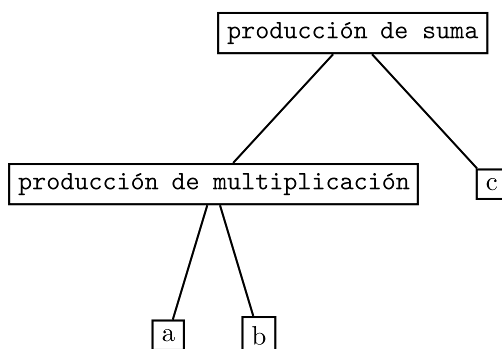


Figura 3.1: Una operación aritmética con precedencia

3.1.1. Información relevante

Habiendo entendido el problema anterior se debe hacer un análisis para determinar cuáles son los nodos que deben ser colapsados y cuáles son los que debemos conservar. Para lograr eso pensemos en cuáles son los elementos que componen una clase o interfaz en Java.

Algunos de los nodos que es evidente debemos conservar son los que dividen el archivo en distintas clases e interfaces ya sean de primer nivel o anidadas. Otros de los nodos que definen elementos estructurales importantes son los enunciados, entre los más conocidos se encuentran los encargados de control de flujo (if, while, for, switch).

Lo primero que necesitamos es el nodo raíz de un archivo; en la gramática

de Java este es el símbolo inicial y lleva el nombre de CompilationUnit; este nodo contiene la declaración de paquete de un archivo. Después del nodo raíz, ¿qué cosas se pueden definir en un archivo de Java? Lo que sigue después del nombre del paquete es la importación de clases que se van a utilizar en el archivo. Finalmente tenemos una declaración sobre si el archivo contiene una clase o una interfaz. A este nivel la información que nos es relevante es:

- La raíz y el paquete al que pertenece (CompilationUnit).
- Declaraciones de importación.
- Declaraciones de clase.
- Declaraciones de interface.

Ahora bien, dentro del cuerpo de una clase, ¿cuáles son las cosas que nos interesan? Pensemos en las cosas que un programador usualmente coloca al inicio de una clase. Lo primero que uno suele ver son campos estáticos para representar constantes y tal vez un bloque inicializador. Después uno puede encontrar los constructores de la clase y los métodos. En ocasiones en un archivo hay más de una clase, o alguna interfaza anidada. Así que la información relevante a este nivel es:

- Inicializadores (pueden o no ser estáticos).
- Declaración de variables miembro.
- Declaración de constructores.
- Declaración de métodos.
- Declaración de clases anidadas.
- Declaración de interfaces anidadas.

Por parte de las interfaces la información relevante es muy similar a la de las clases. Éstas pueden tener una clase anidada, una interfaz anidada, declaraciones de métodos y declaraciones de campos.

El inicializador de una clase puede contener declaraciones de variables locales, enunciados, o declaraciones de clases o interfaces. Esto agrega a nuestra lista:

- Declaración de variables locales.

Las clases e interfaces anidadas no agregan ningún elemento nuevo.

Veamos ahora el caso de los constructores. El constructor de una clase está formado por una llamada explícita a otro constructor de la misma clase o de una superclase. Esta llamada debe ser lo primero que aparezca en el cuerpo del constructor. Esta llamada es seguida por un bloque de enunciados. Agregamos entonces:

- Constructor explícito (una llamada a `super()` o `this()`).
- Un bloque de enunciados.

Las declaraciones de métodos, de campos, de variables locales y constructores explícitos no agregan nada nuevo.

Dentro de las cosas que componen el código en Java se encuentran los enunciados, los cuales definen la mayoría de las cosas que se pueden escribir en nuestro lenguaje. Con ellos obtenemos la siguiente información:

- Enunciado.
- Expresión.
- Lista de expresiones.
- Enunciado etiquetado.
- Enunciado vacío.

Además tenemos también los enunciados con los que controlamos el flujo del programa, que son los siguientes:

- `switch`
- `if`
- `while`
- `do while`
- `for`
- `break`
- `continue`

Cuando se está ejecutando un método puede terminar de dos formas: todo se lleva a cabo normalmente y el método termina, u ocurre algo excepcional y esto debe ser manejado de una forma particular, fuera de lo común. Esto hace necesarios los siguientes enunciados:

- `return`
- `throw`
- `try`

Finalmente tenemos un par de enunciados más. El primero de ellos se encarga de permitir la sincronización y el segundo nos permite asegurarnos de que los valores que utilizamos cumplan con ciertas características. Estos enunciados son:

- `synchronized`
- `assert`

Dentro de los enunciados hay algunos elementos que necesitan más producciones para reconocer la sintaxis adecuada. Cuando estamos utilizando un enunciado `for` es necesario establecer tres aspectos; cómo se inicializan las variables, cómo se actualizan y la condición para continuar dentro del ciclo. La condición de terminación es un enunciado normal (usualmente con un operador relacional), por decirlo de alguna manera, pero los otros dos son especiales. Por lo que también necesitamos de:

- Inicialización de un `for`.
- Actualización de un `for`.

Siguiendo con los enunciados de control de flujo, cuando se utiliza un `switch` es necesario contemplar el caso de las etiquetas. Éstas también necesitan de un nuevo enunciado. Así pues necesitamos la siguiente información:

- Etiqueta de un `switch`.

Todos los elementos e información que hemos mencionado hasta el momento son relevantes para el compilador, pero también es necesario comtemplar el uso de información que sólo tiene sentido para el programador, esto es, los comentarios. En Java existen tres tipos de comentarios. Los comentarios en línea que empiezan con `//`, los de varias líneas que están delimitados por `/*` y `*/` y por último los comentarios Javadoc delimitados por `/**` y `*/`. Es necesario entonces que xCVS se percate también de esta información:

- Comentario en línea.
- Comentario multi-línea.
- Comentario Javadoc.

Cada uno de los elementos que se han listado se mapea a una clase en Java que pertenece al paquete JavaTree. Todas ellas extienden a la clase `CollapsedTreeNode` y así podemos recorrer el árbol de reconocimiento sintáctico y generar nuestro árbol colapsado.

3.1.2. El patrón *Visitor*

Para poder convertir el árbol de reconocimiento sintáctico que obtiene `Javacc` se hace uso del patrón *visitor* [GHJV95], el cual describiremos brevemente a continuación. El patrón de conducta *visitor* representa una operación que será realizada en los elementos de una estructura de objetos. Mediante su uso es posible definir nuevas operaciones sin la necesidad de cambiar las clases de los elementos en los que opera.

En este caso tenemos el árbol que obtuvimos analizando el código fuente, pero es necesario transformalo en un nuevo árbol cuyos nodos estén colapsados. Como vimos en las secciones anteriores esto implica eliminar nodos o transferir la información que ellos tienen a sus padres, los cuales hemos identificado como nodos con información relevante. Lo que el patrón visitor nos permite es tener un objeto que sabe cómo colapsar los nodos del árbol de reconocimiento sintáctico. Este objeto recorrerá el árbol desde la raíz y conforme va visitando los nodos va construyendo un árbol JavaTree el cual contiene solo la información que nos interesa. Es necesario que los nodos del árbol de reconocimiento sintáctico esten preparados para aceptar al visitante, y hacer la llamada pertinente para que el visitante realice las operaciones que él define.

En la figura 3.2 se muestra el diagrama de este patrón. Es necesario definir una interfaz `Visitor` para que los elementos que queremos sean recorridos (`Elemento` en la figura 3.2) puedan indicarle al visitante que debe realizar una operación `VisitaElementoConcreto()`. Los clases que implementan esta interfaz, `VisitorConcreto`, definen las operaciones que se realizarán en la estructura que se está visitando. Los elementos sólo deben definir la operación con la que se asocia el visitante con ellos, `acepta(Visitor v)`. En el diagrama, `Estructura` representa la estructura de elementos que serán visitados. Finalmente `Cliente` es el programa encargado de iniciar la visita.

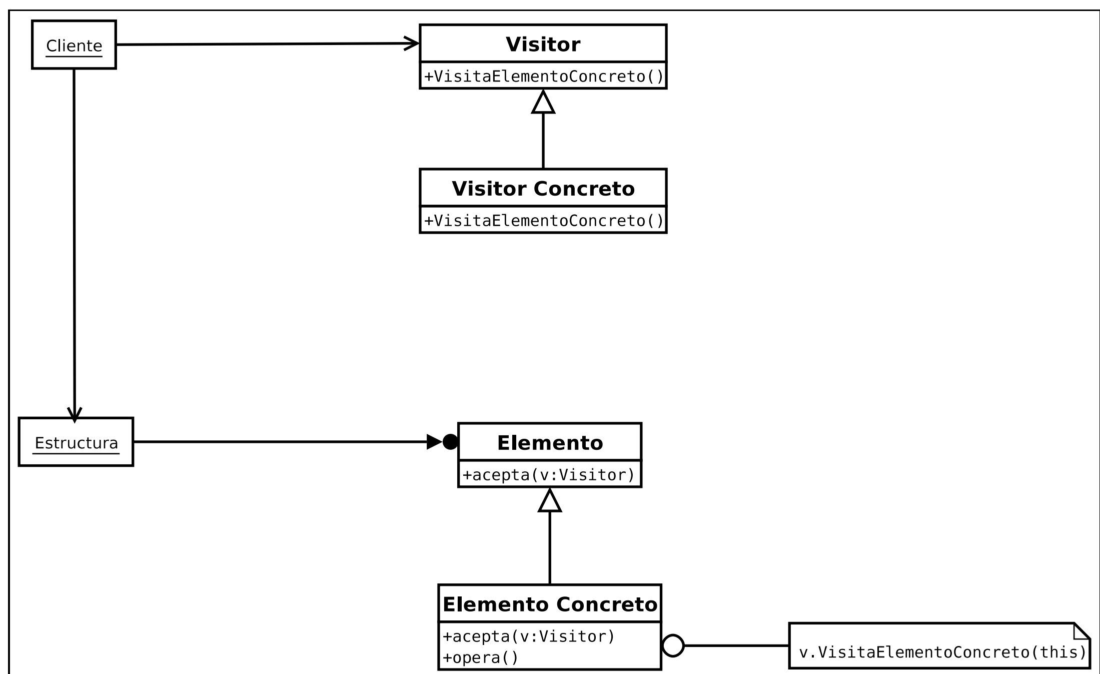


Figura 3.2: El patrón Visitor

Como ya mencionamos xCVS hace uso de este patrón para convertir el árbol obtenido por el reconocedor sintáctico, a un árbol con el cual se pueden obtener diferencias. Otras aplicaciones podrían ser la impresión de una estructura, se podría tener un *visitor* que lo despliegue como un archivo HTML, otro que facilite la transmisión de la información y lo serialice a un archivo XML.

3.1.3. Análisis detallado de la información

Veamos con más detalle dónde están los aspectos que faltan por describir de Java. Lo haremos en orden alfabético, así que empecemos con las aseveraciones.

Aseveraciones

La única modificación que hubo en la versión 1.4 de Java fue el uso de aseveraciones (assertions). Para revisar el uso de esta característica se puede consultar
`http://Java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`. Esta expresión puede aparecer en dos formas, la más simple es:

`assert: Expresion booleana;`

En la que se evalúa la expresión booleana y en caso de ser falsa, se lanza un error sin mensaje alguno. La segunda forma es:

`assert: Expresion booleana; Expresion`

En este caso nuevamente se evalúa la expresión booleana, pero en caso de ser falsa al error se le asocia como mensaje el valor de la segunda expresión.

Break

Representa un enunciado `break` dentro de un ciclo. Su único campo es el identificador (etiqueta) a donde se debe pasar el control del programa.

Declaración de Clase

Representa una clase de primer nivel (top-level class) en un archivo. Contiene el nombre de la clase, la superclase que extiende, una lista de interfaces que implementa y sus modificadores que pueden ser:

- `abstract`
- `final`

- `public`
- `strictfp`

Compilation unit

La raíz de todo programa en Java. Aquí es donde empieza el reconocimiento sintáctico. La única propiedad colapsada que tiene es el nombre del paquete al que pertenece el archivo fuente.

Constructores

La declaración de un constructor. Contiene el nombre del constructor, una lista de excepciones que puede arrojar, una lista de parámetros y los modificadores de acceso, que pueden ser los siguientes:

- `public`
- `private`
- `protected`

continue

Representa un enunciado para continuar en un ciclo. En código esto se ve de la siguiente forma:

```
continue: Etiqueta
```

De forma que lo único que necesitamos es una etiqueta para saber a dónde debemos transferir la ejecución del programa.

do ... while

Representa un enunciado do-while como el siguiente:

```
do{
...
} while( expresion );
```

Por lo que sólo se necesita de la expresión que será evaluada para determinar si se sale del ciclo o no.

Enunciado vacío

Un enunciado vacío. Representado como un ';'.

Constructor explícito

Una llamada explícita a otro constructor; ésta puede ser una llamada a un constructor de una superclase (`super()`) o una llamada a otro constructor de la misma clase (`this()`). Como campos contiene el nombre del constructor y una lista de parámetros. Permite asegurarnos que estas llamadas sean lo primero que aparece dentro de un constructor.

Miembros

Los campos dentro de una clase. Contiene el tipo de las variables y una lista con los nombres de las variables. Además están los modificadores de acceso que son los siguientes:

- `public`
- `protected`
- `private`
- `static`
- `final`
- `transient`
- `volatile`

Inicialización de un ciclo `for`

Cuando declaramos un ciclo `for` la primera parte que se escribe es la inicialización. Ésta es una lista de expresiones, en la que podemos declarar tantas variables como necesitemos, siempre que éstas sean del mismo tipo.

for(***int i = 0, j = 20***; i $\leq$ j; i++, j- -)

Actualización de un ciclo `for`

Siguiendo con el ejemplo anterior, despues de inicializar las variables que vamos a utilizar en el ciclo, es necesario colocar una lista de expresiones que indique cómo se actualizan las variables para en algún momento poder salir del ciclo.

for(int i = 0, j = 20; i ≤ j; ***i++, j- -***)

if

Un enunciado de control de flujo selectivo. Puede presentarse de dos formas, en la primera se evalúa una condición booleana y en caso de resultar verdadera se ejecuta una expresión (que puede ser sencilla o un bloque de expresiones)

```
if(condición){
...
}
```

En su segunda forma, el `if` está apareado con una segunda expresión (que nuevamente puede ser un bloque) `else`, que se evalúa si la condición resulta falsa.

```
if(condición){
...
}
else{
...
}
```

import

Una declaración de importación de clases/paquetes. Puede presentarse de dos maneras, la primera es una importación explícita. En esta forma es necesario indicar el nombre del paquete y la clase que se quiere importar.

```
import mi.paquete.de.Java.LaClase;
```

En su forma alterna se puede importar todas las clases dentro de un paquete, aun cuando no se expresen los nombres de las clases.

```
import mi.paquete.de.Java.*;
```

Inicializador

Es un bloque que se ejecuta cuando se crea el objeto. Puede utilizarse, por ejemplo, para asegurarse que una condición para un ciclo sea correcta, o inicializar variables estáticas. Puede o no ser estático.

Declaración de interfaces

Una declaración de interfaz de primer nivel (top-level). Contiene campos para el nombre de la interfaz, un lista para las interfaces que implementa y los siguientes modificadores de acceso:

- `abstract`
- `public`
- `strictfp`

CollapsedInlineComment

Representa un comentario en línea.

CollapsedJavadocComment

Un comentario para generar documentación JavaDoc.

CollapsedMultilineComment

Representa un comentario de varias líneas.

CollapsedLabeledStatement

Representa una etiqueta dentro del código. Las etiquetas son de la forma `etiqueta: enunciado`. Contiene un campo con la etiqueta.

CollapsedLocalVariableDeclaration

Representa una declaración local de una variable, esto es el tipo de las variables, el nombre y además indica si la variable es o no final. Dado que estas declaraciones se pueden utilizar dentro de un for puede no terminar con un `;` por lo que la clase contiene un booleano que indica si es así o no.

CollapsedMethodDeclaration

La declaración de un método. Contiene el tipo que regresa el nombre de método y una lista de excepciones que puede arrojar. Además puede tener un bloque si es una implementación, o un `;` si es una declaración. Los modificadores de acceso que puede tener son los siguientes:

- `public`
- `protected`
- `private`
- `static`
- `abstract`
- `final`
- `native`
- `synchronized`
- `strictfp`

CollapsedNestedClassDeclaration

Una declaración de una clase anidada. Los modificadores de acceso de estas clases son distintas a las de una clase de primer nivel. Extiende a `CollapsedClassDeclaration`. Los nuevos modificadores son los siguientes:

- `private`
- `protected`
- `static`

CollapsedNestedInterfaceDeclaration

Representa una declaración anidada de una interfaz, tiene modificadores de acceso distintos a los de una interfaz de primer nivel. Extiende a `CollapsedInterfaceDeclaration`, los nuevos modificadores son los siguientes:

- `static`

- `abstract`
- `final`
- `public`
- `protected`
- `private`
- `strictfp`

CollapsedReturnStatement

El enunciado de retorno de un método, contiene la expresión que se debe regresar.

CollapsedStatement

Representa un enunciado dentro de Java.

CollapsedStatementExpression

Representa una expresión y puede o no terminar en `;` por lo que tiene un booleano que indica este hecho.

CollapsedStatementExpressionList

Una lista de expresiones de enunciados.

CollapsedSwitchLabel

Una etiqueta dentro de un enunciado `switch`. Puede ser una etiqueta de la forma `case Expression :` o de la forma `default:`. En el primer caso su único campo contiene la expresión, en el segundo el campo es vacío.

CollapsedSwitchStatement

Un enunciado `switch`. Su único campo contiene la expresión que se evaluará.

CollapsedSynchronizedStatement

Un enunciado de sincronización, contiene la expresión sobre la cual se realizará la sincronización.

CollapsedThrowStatement

Un enunciado para arrojar una excepción. Contiene la excepción que se arrojará.

CollapsedTreeNode

La implementación de nodo para representar el *JavaTree*.

CollapsedTreeVisitor

El encargado de hacer la conversión de un árbol a otro; en 3.1.2 veremos a detalle cómo funciona.

CollapsedTryStatement

Un enunciado para tratar de llevar a cabo una operación, estando preparado para lidiar con una excepción. Contiene una variable que indica si el `try` tiene un bloque `finally`, además tiene una lista con los nombres de las excepciones que pueden ocurrir.

CollapsedWhileStatement

Un enunciado de control de flujo iterativo. Contiene la expresión que se evaluará para determinar si se ejecuta el ciclo o no.

3.2. Serialización de un JavaTree a un archivo XML

Ahora que ya se tiene una representación en un árbol del código en Java de un programa, es necesario entonces encontrar las diferencias entre dos de estos árboles para poder calcular la delta entre ellos y generar el parche correspondiente. Para eso se utilizó la herramienta DiffXML, la cual trabaja con el algoritmo definido en la sección 1.5.1.

Para poder convertir cada una de nuestras estructuras arborescentes se hace uso del paquete XStream, el cual es una biblioteca utilizada para serializar objetos a XML y leerlos nuevamente; entre las características que presenta se encuentran las siguientes:

- Facilidad de uso.
- No se requieren mapeos.

- Desempeño.
- Los objetos no necesitan ser modificados.
- Soporta referencias circulares.

Mediante el uso de XStream es muy sencillo serializar los JavaTree. Por conveniencia se implementó una interfaz que inicializa el reconocedor que utiliza XStream, y le provee el objeto que se desea serializar y el archivo a donde se desea escribir, o bien regresa la cadena con el archivo XML.

3.3. Información modificada

Mediante la conversión definida anteriormente se logra conservar toda la información del código fuente, con una sola excepción. Esta excepción consiste en la colocación de los comentarios. Después de que se construyó el árbol colapsado los comentarios (de cualquier tipo) aparecen como el primer hijo de cada uno de los nodos. Esto quiere decir que los comentarios se adelantan a la expresión que los contiene. Entonces una expresión con un comentario que originalmente aparecía como:

```
i = 0; // Se inicializa la variable i;
```

Ahora se ve como:

```
// Se inicializa la variable i;
i = 0;
```

Terminamos así la revisión del modelo y la arquitectura, por lo que ahora pasamos a ver como se implementó XcvsClient, un cliente gráfico para interactuar con xCVS.

Capítulo 4

XcvsClient, Un cliente gráfico para xCVS

Ahora que se ha entendido cómo es que funciona el servidor y las ideas detrás de xCVS, es el momento de presentar el cliente mediante el cual se puede interactuar con el servidor. La distribución actual de xCVS tiene un cliente llamado *XcvsClient*, el cual presenta una GUI diseñada en Java Swing. Hay que aclarar que el cliente puede ser desarrollado en cualquier lenguaje que se desee, no necesariamente en el mismo en el que se programó el servidor. A continuación se presentarán las tareas más cotidianas de XcvsClient.

4.1. Administración de xCVS

4.1.1. Módulos

XcvsClient está dividido en varios paneles desde los cuales el usuario puede controlar diferentes aspectos del servidor, o de sus copias locales. El primero de estos paneles es el encargado de los módulos y de los archivos que éstos contienen. Este panel es el que se muestra en la figura 4.1. Además de consultar los archivos que pertenecen a una ramificación, dentro de este panel el usuario tiene las siguientes opciones:

- Agregar o eliminar un módulo.
- Agregar, eliminar o modificar ramificaciones.
- Agregar, eliminar o modificar archivos.

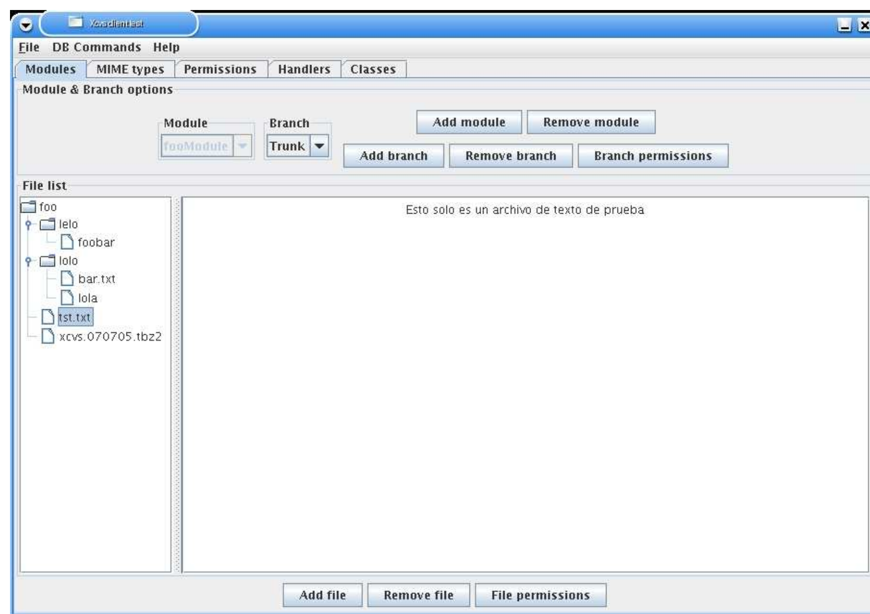


Figura 4.1: El panel de módulos

4.1.2. Tipos MIME

Los siguientes paneles que debemos revisar son en los que se pueden consultar los tipos MIME y las extensiones asociadas a los mismos. El panel para los tipos MIME se muestra en la figura 4.2. Ahí el usuario puede consultar (o modificar) cuáles son los manejadores que un tipo MIME tiene asociados, así como agregar nuevos tipos o eliminar otros que ya no sean requeridos. En la figura 4.3 se muestra el panel en el que se indica cuáles son las extensiones que un tipo MIME tiene asociadas.

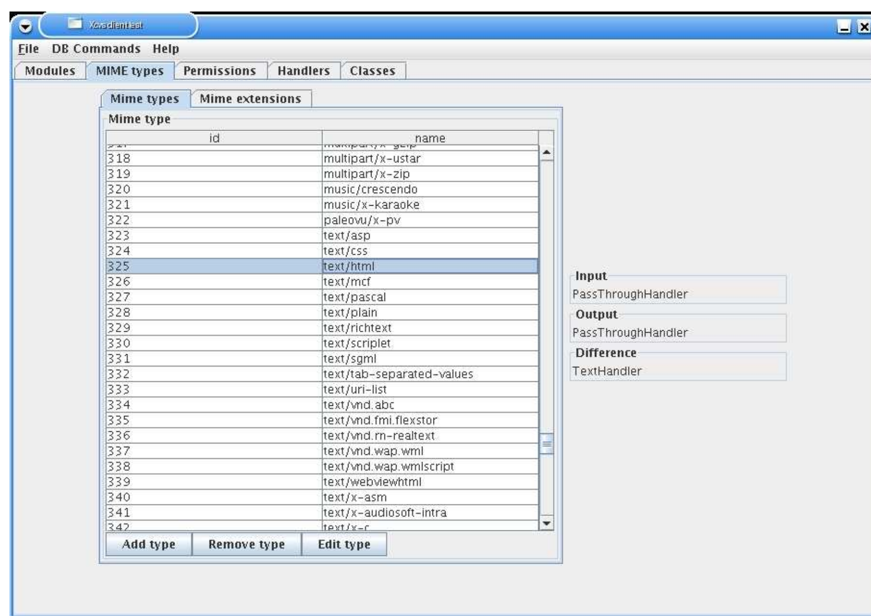


Figura 4.2: El panel de tipos MIME

4.1.3. Usuarios y grupos

En el panel de permisos es en donde el usuario puede crear nuevos usuarios, eliminarlos y modificar los permisos que se le han concedido. Esas son acciones que también se pueden realizar con los grupos de xCVS. Además se presenta la posibilidad de agregar nuevos miembros a un grupo o eliminar a un miembro de un grupo. Estos dos paneles se muestran en las figuras 4.4 y 4.5.

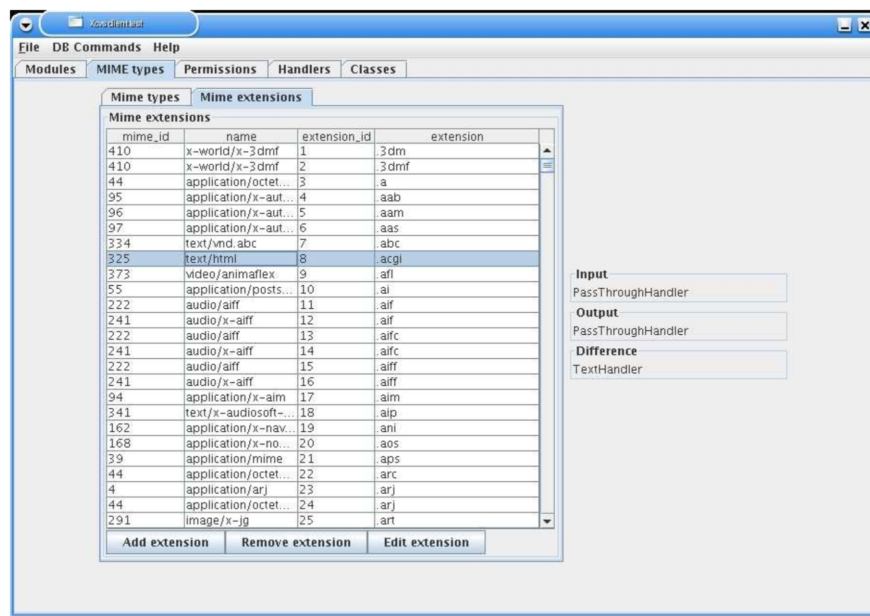


Figura 4.3: El panel de extensiones

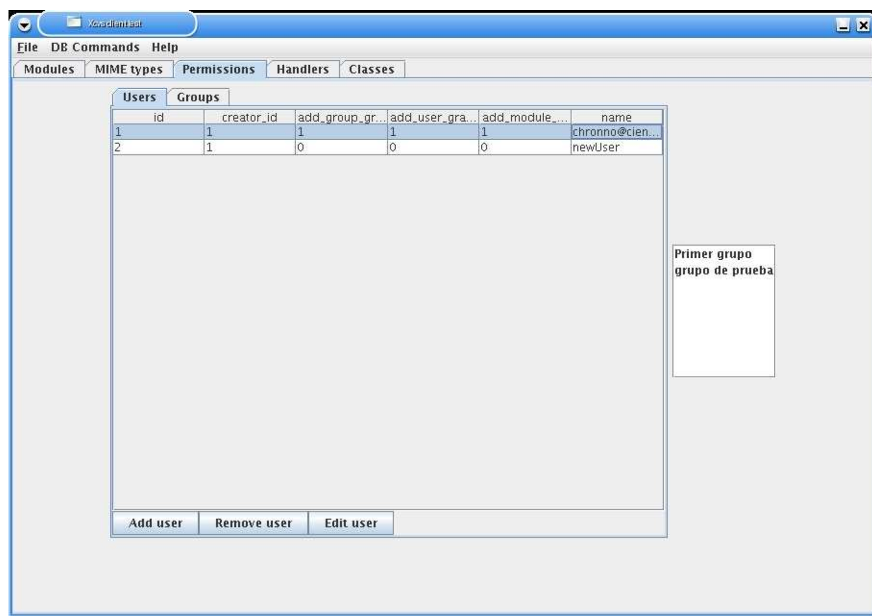


Figura 4.4: El panel de usuarios

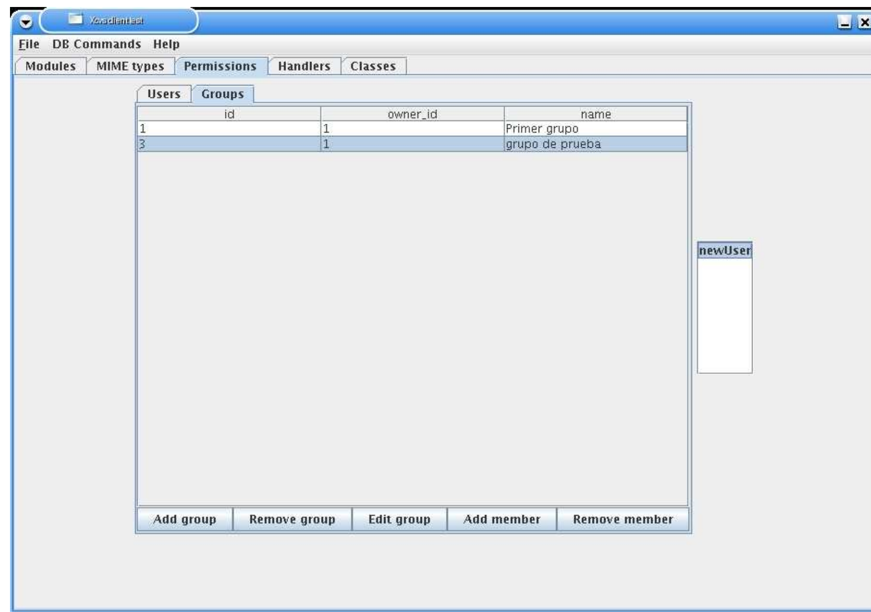


Figura 4.5: El panel de grupos

4.1.4. Manejadores

Se tiene también un panel en el que es posible consultar los manejadores tanto de entrada/salida como de diferencias que se tienen instalados, y la clase de Java a la cual están asociados. Así es posible modificar el comportamiento de xCVS sin alterar las clases asociadas. Más adelante veremos cómo se pueden cambiar estas asociaciones. En la figura 4.6 se muestra este panel.

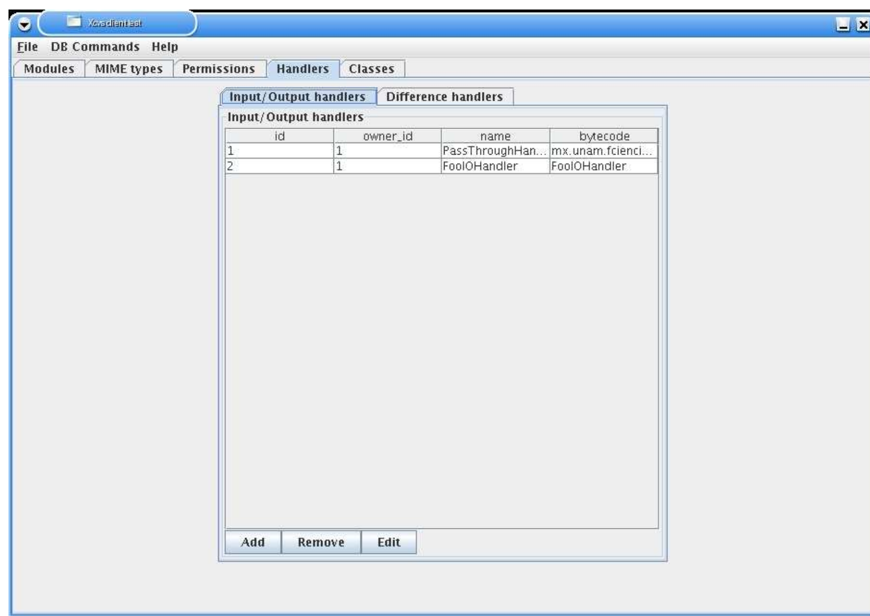


Figura 4.6: El panel de manejadores

4.2. Utilización de xCVS

Ahora que conocemos las diferentes opciones que nos presenta Xcvs-Client, veremos cómo se llevan a cabo las operaciones más comunes en el uso de esta herramienta.

4.2.1. Inico de sesión

Siempre que queramos modificar la información que está en el repositorio es necesario iniciar una sesión dentro del sistema; para eso es necesario proporcionar al cliente nuestro nombre de usuario y contraseña las cuales serán autenticadas de forma segura por el servidor. Para iniciar una sesión es necesario en la barra de menús de XcvsClient seleccionar la opción `DB Commands` → `Open Session`. Así se le presentará al usuario un diálogo para que el introduzca su nombre de usuario y contraseña, como se muestra en la figura 4.7.

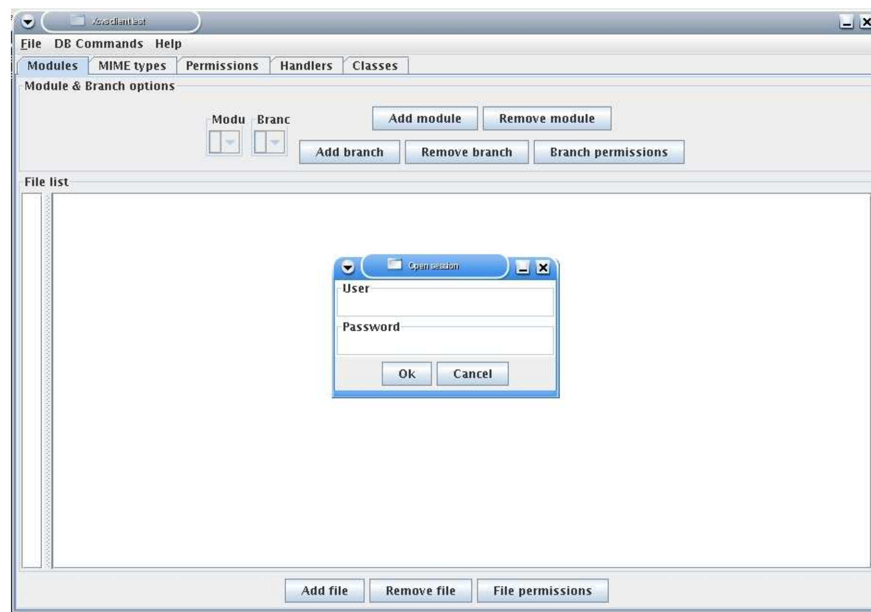


Figura 4.7: El diálogo para iniciar sesión

4.2.2. Creación de un módulo

Una vez iniciada la sesión el usuario puede crear un módulo para que éste sea sometido al control de versiones. Esto puede lograrlo de dos maneras, una es crear el módulo y dejarlo vacío, para después agregar archivos conforme vaya siendo necesario. La otra opción es si ya se tiene un conjunto de archivo que se quiere controlar. En ese caso el usuario puede seleccionar crear el

nuevo módulo a partir de un directorio en el sistema de archivos local. El diálogo para crear módulos se presenta en la figura 4.8.

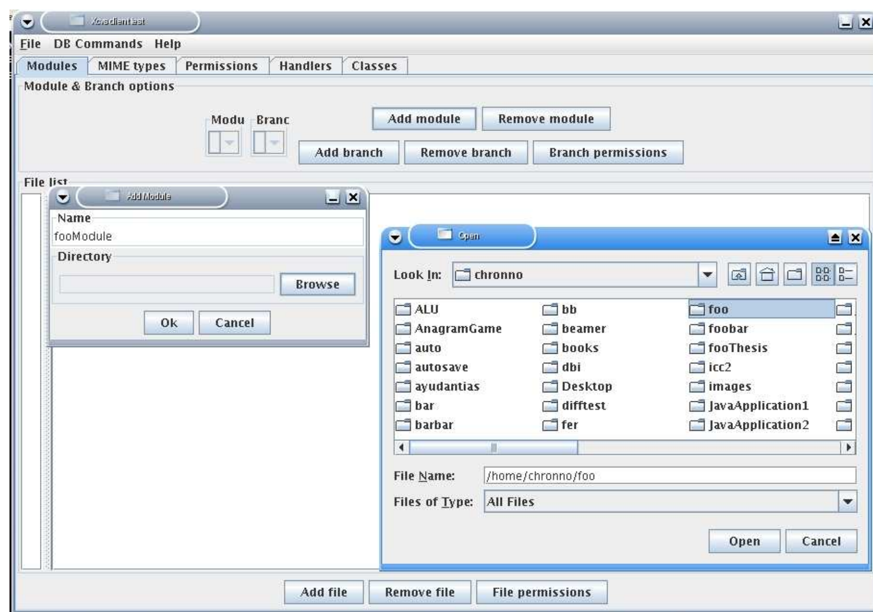


Figura 4.8: El diálogo para crear módulos

4.2.3. Obtención de una copia local

Una vez que se tiene un módulo en el repositorio es necesario que otros usuarios obtengan copias de esos archivos para poder modificarlos. En esta ocasión también existen dos formas de hacerlo. En la primera de ellas se hace uso de la instrucción `Checkout` que se revisó en capítulos anteriores. En este caso se obtiene una copia en la que se incluyen archivos de control necesarios para poder realizar operaciones de control de versiones, esto nos permite tener una copia con la cual se continuará el desarrollo del módulo. La otra opción es obtener una copia lista para la distribución del módulo, es decir una copia en la cuál no se reincorporarán cambios al repositorio. Para hacer uso de la primera opción es necesario ejecutar `File` → `Checkout`, y para la segunda será necesario `File` → `Export Module`. Estas instrucciones las podemos apreciar en la figura 4.9.

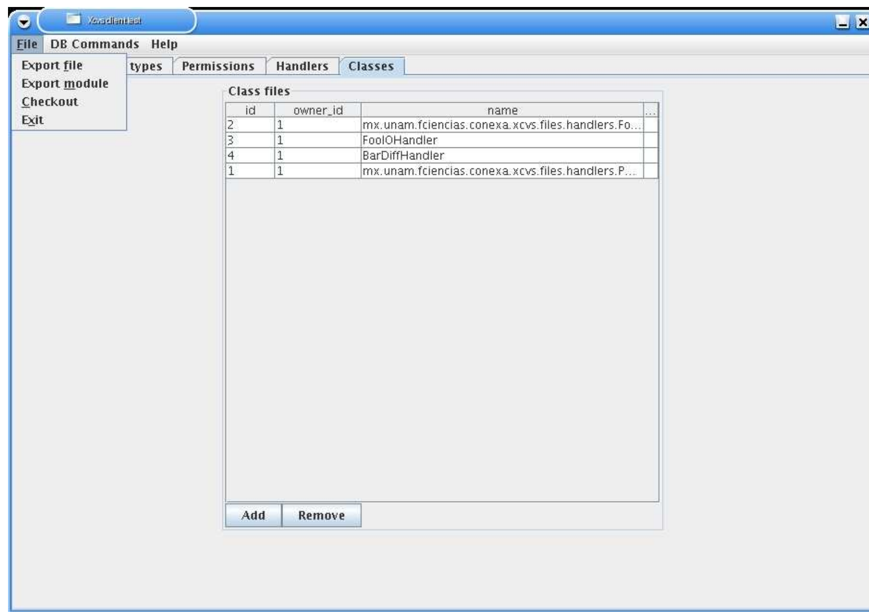


Figura 4.9: El menú de archivos

4.2.4. Modificación de la estructura del módulo

Es posible que haya ocasiones en las que se decida que un archivo debe de cambiar su posición relativa (en la estructura de directorios). Si bien esto es posible hacerlo desde el sistema de archivos, xCVS permite que estas operaciones sean sujetas a control. Es decir se presentan operaciones equivalentes a las presentadas por el sistema operativo, que además permiten que estos cambios se propaguen. Las posibles operaciones son:

- Remove
- Rename
- Move
- Copy

Estas instrucciones se encuentran todas dentro del menú `DB Commands`, como se ve en la figura 4.10.

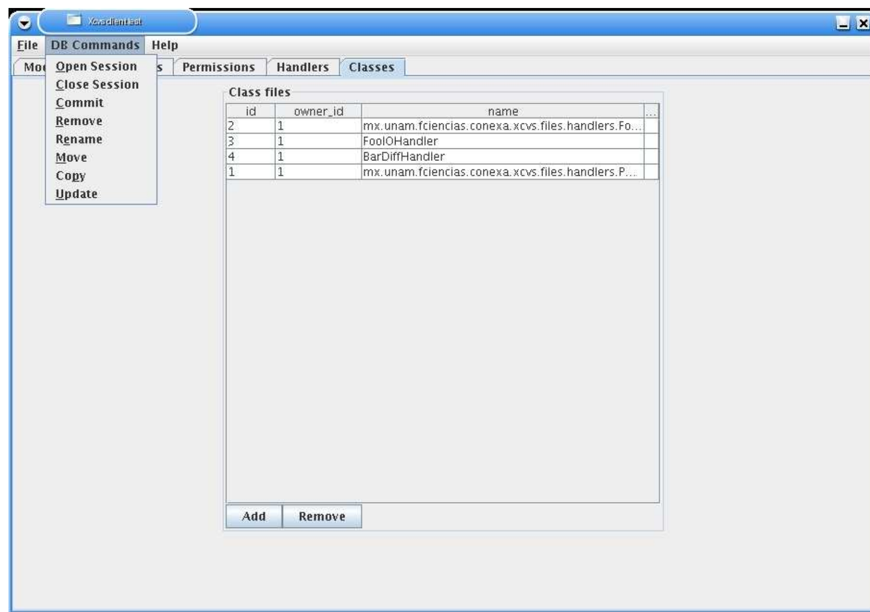


Figura 4.10: El menú de instrucciones

4.2.5. Sincronición con el repositorio

Ésta es la instrucción que se debe utilizar para obtener la versión más reciente de un archivo desde el repositorio; se accede a ella mediante `DB Commmands` → `Update`. Nuevamente se le presenta al usuario un diálogo para seleccionar el archivo, como se muestra en la figura 4.11.

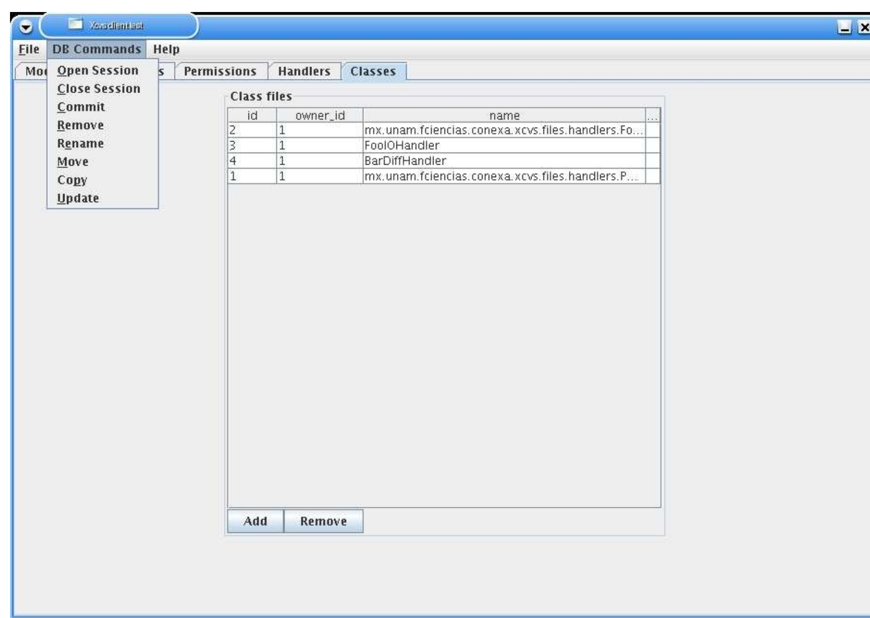


Figura 4.11: El diálogo para sincronizarse con el repositorio

4.2.6. Envió de cambios

La instrucción complementaria a la anterior, es la encargada de enviar un archivo modificado al servidor para que éste calcule las diferencias y genere las deltas adecuadas. Para utilizarla es necesario seleccionar mediante el menú `DB Commmands` → `Commit`. El usuario deberá seleccionar el archivo mediante un diálogo como el que se mostró para la instrucción anterior (figura 4.11).

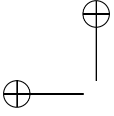
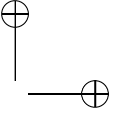
Estas son las instrucciones con las que el usuario puede interactuar con el servidor de xCVS. Cabe aclarar que estas son las instrucciones usuales para la mayoría de los sistemas de manejo de versiones, por lo que usuarios con experiencia previa utilizando otros sistemas podrán adaptarse rápidamente al uso de xCVS.

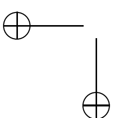
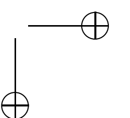
El programa tanto en distribución compilada, como el código fuente se encuentra disponible en línea en la dirección:

`http://sourceforge.net/projects/xtcvs`

El código fuente se encuentra disponible bajo la licencias GPL [1], por lo cual se permite hacer modificaciones al mismo, siempre que estas permanezcan bajo la misma licencia. Para poder hacer uso de xCVS, es necesario tener instalado en el sistema una versión de la máquina virtual de Java (>= 1.4) y un servidor de bases de datos PostreSQL (>= 8,0). El resto de las dependencias para compilar xCVS se incluyen en la distribución de código fuente.

[1] General Public License

Capítulo 5

Conclusiones y Trabajo futuro

5.1. Conclusiones

En el desarrollo de este trabajo se hizo evidente que la convergencia de diversas tecnlogías, conjugadas con el uso de patrones de diseño adecuados, puede dar lugar a soluciones mucho más adecuadas, eficientes y robustas para problemas que, hasta ahora, solo habían podido ser resueltos en parte y deficientemente.

Es claro también que el problema abordado aquí se puede resolver de manera mucho más eficiente y completa en la medida en que se aprovecho la meta-información que, hasta la fecha muy pocas aplicaciones similires explotan. Es decir a través del uso de la meta-información fue posible reducir un problema complejo en un problema más sencillo, para el cuál tenemos una mejor solución.

Además se hizo patente el hecho que el uso de herramientas sencillas, open source, disponibles en la red se puede lograr una herramienta útil para la mayoría de los programadores. Constatamos también que aprovechando la meta-información se obtiene un sistema de versiones más inteligente y mucho más eficiente ya que se obtienen guiones de edición pequeños con los que es posible minimizar la información que es necesario transmitir entre cliente y servidor. El uso de este tipo de herramientas podría utilizarse también en otras aplicaciones como lo son navegadores.

5.2. Trabajo futuro

El trabajo a futuro de xCVS es muy amplio, pues esa fue una de las ideas principales al desarrollarlo. Entre las opciones para seguir extendiéndolo están las siguientes:

- Soporte para nuevos lenguajes de programación.
- Nuevos filtros para el pre / post procesamiento de los archivos.
- Soporte para detección de tipos MIME mediante números mágicos.
- Nuevos algoritmos de comparación.
- Definición de las gramáticas por medio de un lenguaje propio que permita compilarse y generar los serializadores y manejadores correspondientes.

Apéndice A

Tablas en la base de datos

A.1. Esquema auth

Sesiones

Para iniciar una sesión en el sistema es necesario proveer el nombre del usuario y su contraseña. Una vez que se autentica al usuario se le regresa la llave con la cual puede abrir su sesión cada vez que lo necesite. Esta llave debe ser almacenada por su cliente pues sin ella no podrá continuar con su sesión.

La información sobre las sesiones se almacena en la tabla `auth.session` que contiene los siguientes campos:

- **id**
  Un identificador para la sesión.
- **user_id**
  El identificador del usuario al que pertenece la sesión.
- **skey**
  La llave de la sesión.
- **host**
  El servidor.
- **start_time**
  La hora a la que se inició la sesión.
- **end_time**
  La hora a la que terminó la sesión.

Para iniciar una sesión se utiliza la función `auth.open_session(text user, text password, text host)` la cual se encarga de verificar que exista el usuario y que la contraseña sea la correcta (mediante la función `auth.check_passphrase`).

Para cerrar una sesión es necesario conocer la llave, para hacer uso de la función `auth.close_session(int sessionId, text key)`. Se pide la llave para que solo el usuario que inició la sesión pueda terminar la misma.

Para la autenticación de los usuarios se guarda su contraseña en la tabla `auth.credential_passphrase` la cual contiene la siguiente información:

- **id**
  Un identificador para la contraseña.
- **user_id**
  El identificador para el usuario al que pertenece la contraseña.
- **data**
  La contraseña *per se* (cifrada).

Para autenticar a un usuario se utiliza la función `auth.check_passphrase (int uid, text password)`, la cual se encarga de cifrar el texto que le pasan como parámetro y compararlo con los datos cifrados almacenados en la tabla.

Usuarios

Los usuarios se almacenan en la tabla `auth.user`, en la que se tienen los siguientes campos:

- **id**
  Un identificador único para cada usuario.
- **creator_id**
  El identificador del usuario que creó a este usuario.
- **name**
  El nombre del usuario.
- **add_group_grantor_id**
  El identificador del usuario que le otorgó a ese usuario el permiso para crear grupos.

- **add_user_grantor_id**
  El identificador del usuario que le otorgó a dicho usuario el permiso para crear usuarios.

- **add_module_grantor_id**
  El identificador del usuario que le otorgó a ese usuario el permiso para crear módulos.

Se pide que el nombre del usuario sea único. Además se tiene una restricción para que el único usuario al que se le permite tener un identificador de creador igual a su identificador de usuario sea el usuario creado por el bootstrap; este usuario debe tener como su identificador el número 1.

El nombre de la función para agregar usuarios es `auth.create_user (text name, boolean can_add_user, boolean can_add_group, boolean can_add_module)`.

Para eliminar a un usuario se hace uso de la función `auth.delete_user (text name)` la cual elimina al usuario si el usuario que está llamando a esta función es el creador original del usuario.

Para editar la información perteneciente a un usuario se tiene la función `auth.update_user (text name, text new_name, boolean can_add_user, boolean can_add_group, boolean can_add_module)` con la cual se puede cambiar el nombre de un usuario, así como los permisos que se le conceden. Para poder revocarle el permiso a un usuario es necesario ser el usuario que se lo concedió en un principio. Las concesiones de permisos se hacen mediante las siguientes funciones:

- auth.grant_add_user (text name)
- auth.grant_add_group (text name)
- auth.grant_add_module (text name)

Y para revocar permisos se hace uso de:

- auth.revoke_add_user (text name)
- auth.revoke_add_group (text name)
- auth.revoke_add_module (text name)

Grupos

Los grupos están almacenados en la tabla `auth.group`. Los siguientes datos son los utilizados en esta tabla:

- **id**
  Un identificador para el grupo.
- **owner_id**
  El creador de este grupo.
- **name**
  Un nombre para el grupo.

Se pide que el nombre del grupo sea único y que el creador del grupo sea no nulo.

Para crear un nuevo grupo se hace mediante la función `auth.create_group (text name)`.

Para eliminar un grupo existe la función `auth.delete_group (text)`.

Si es necesario modificar un grupo esto se logra mediante la función `auth.update_group (text name, text new_name)`.

La información de pertenencia a un grupo es almacenada en la tabla `auth.group_membership`, en la cual cada renglón contiene los siguientes campos:

- **user_id**
  El identificador de un usuario perteneciente al grupo.
- **group_id**
  El identificador del grupo al que pertenece el usuario registrado en este renglón.
- **add_member_grantor_id**
  El identificador del usuario que le otorgó a este usuario el permiso para agregar nuevos miembros a este grupo.
- **grantor_id**
  El identificador del usuario que agregó a este usuario al grupo.

Se tiene una restricción que pide que la pareja `(user_id, group_id)` sea única.

Las funciones utilizadas para manipular la información de estas tablas son las siguientes:

- **auth.create_group(text name)**
  Es utilizada para que un usuario agregue un nuevo grupo (si tiene los permisos suficientes). Si el usuario consigue crear el grupo, él mismo se concede permisos para agregar nuevos miembros.

- **auth.add_to_group**
  Permite a un usuario agregar a un nuevo miembro a algún grupo existente (si tiene permisos suficientes).

Sesiones

Para iniciar una sesión en el sistema es necesario proveer el nombre del usuario, su contraseña y el servidor en el que esta trabajando. Una vez que se autentica al usuario se le regresa la llave con la cual puede abrir su sesión cada vez que lo necesite. Esta llave debe ser almacenada por su cliente pues sin ella no podrá continuar con su sesión.

A.1.1. Manejo de módulos

Generalmente las personas agrupan dentro de un directorio los archivos que pertenecen a un mismo proyecto, dentro de xCVS esta misma tarea de agrupamiento es realizada por un módulo. Para la definición de éstos es necesaria la tabla `xcvs.module`, la cual contiene los siguientes datos:

- **id**
  Un identificador para el módulo.

- **owner_id**
  El identificador del dueño del módulo.

- **name**
  El nombre del módulo.

Cada módulo puede tener varias bifurcaciones que representan diferentes estados de un mismo proyecto. Cada una de estas bifurcaciones contiene los archivos y las diferencias (delta) que la constituyen como una unidad de trabajo. Es por eso que cada módulo debe tener al menos una bifurcación, que será el tronco a partir del cual se crearán todas las demás bifurcaciones.

La tabla encargada de las bifurcaciones es `xcvs.branch` en la que se tienen los siguientes datos:

- **id**
  El identificador de la bifurcación.

- **module_id**
  El módulo al que pertenece esta bifurcación.

- **name**
  El nombre de esta bifurcación.

Se tiene una restricción para que las tuplas (module_id, name) sean únicas.

Las bifurcaciones es en donde primero vamos a lidiar con el problema de seguridad en el sistema. Debemos asegurar que solo un grupo selecto de usuarios pueda hacer modificaciones a cierta rama de desarrollo, ya sea una rama secundaria o el tronco del árbol. En la tabla xcvs.branch_grant se tiene la siguiente información:

- **user_id**
  El usuario al que se le otorgan los permisos.
- **group_id**
  El grupo al que se le otorgan los permisos.
- **branch_id**
  La bifurcación sobre la cual se otorgan los permisos.
- **owner_grantor_id**
  El usuario que otorga permisos de posesión (para borrar la rama).
- **read_grantor_id**
  El usuario que otorga permisos de lectura.
- **add_grantor_id**
  El usuario que otorga permiso para agregar nuevos archivos a la rama.
- **branch_grantor_id**
  El usuario que otorga permisos para crear nuevas bifurcaciones.

Se tiene una restricción en la tabla para asegurar que a lo más uno de los campos user_id o group_id sean nulos. Además las tuplas (user_id, group_id, branch_id) deben ser únicas. Los permisos deben ser revocados de forma tal que si a un usuario se le retiran sus privilegios todos los usuarios de los que el sea el proveedor también perderán sus permisos.

A.1.2. Manejo de archivos

Los archivos corresponden a los datos en el repositorio, las clases dentro del paquete mx.unam.fciencias.conexa.xcvs.files son las encargadas del manejo de archivos dentro de la base de datos. Para ello se tiene la tabla xcvs.file en la que cada archivo se representa de la siguiente forma:

- **id**
  El identificador para cada archivo.
- **revision**
  El número de revisión actual de este archivo (manejado por el sistema, independiente del usuario).
- **owner_id**
  El identificador del dueño del archivo
- **branch_id**
  La bifurcación a la que pertenece.
- **parent_id**
  El padre de este archivo (estableciendo un sistema jerárquico similar al de un Sistema Operativo).
- **diff_handler_id**
  El manejador de diferencias que se usa para comparar este archivo.
- **input_handler_id**
  El filtro de entrada asociado a este archivo.
- **output_handler_id**
  El filtro de salida, utilizado antes de escribir una copia de este archivo.
- **mtype_id**
  El tipo MIME del archivo.
- **name**
  El nombre del archivo.
- **data**
  Los datos; si son nulos entonces quiere decir que el archivo es en realidad un directorio.

Se debe restringir el que las tuplas `(parent_id, name, branch_id)` sean únicas.

Además de lo mencionado anteriormente se debe tener una tabla de permisos similar a la de las bifurcaciones en la que se tiene lo siguiente:

- **user_id**
  El usuario al que se le otorgan permisos sobre un archivo.

- **group_id**
  El grupo al que se le otorgan permisos sobre un archivo.

- **file_id**
  El archivo sobre el que se otorgan permisos.

- **read_grantor_id**
  El usuario que otorgó permiso de lectura.

- **write_grantor_id**
  El usuario que otorgó permiso de escritura.

Al igual que con los permisos de bifurcaciones sólo se permite que uno de los dos identificadores de grupo o de usuario sea nulo a la vez. Es decir, no se permite tener un renglón que otorgue permisos a un grupo y a un usuario. Las tuplas `(user_id, group_id, file_id)` deben ser únicas. Al igual que en los permisos de bifurcación las revocaciones serán en cascada.

Uno de las principales características de los sistemas de manejo de versiones es el poder conservar la historia sobre cómo se ha modificado un archivo a lo largo de los cambios. Esto permite obtener comentarios sobre el porqué se realizaron cambios de una versión a otra, además de permitirnos conocer qué usuario fue el encargado de realizar dichos cambios. Para este propósito es que se tiene la tabla `xcvs.history` en la que se almacena la siguiente información:

- **id**
  Un identificador para cada entrada de historia.

- **commiter_id**
  El identificador del usuario que realizó el cambio.

- **branch_id**
  El identificador de la rama en la que se realizó el cambio.

- **file_id**
  El identificador del archivo.

- **old_revision**
  El número de la revisión anterior.

- **new_revision**
  El número de la nueva revisión.

- **diff_handler_id**
  El identificador del manejador encargado de hacer las actualizaciones.
- **delta**
  La delta (diferencia) para convertir la revisión anterior en la nueva.
- **comment**
  Comentarios agregados al momento de registrar los cambios en el repositorio.
- **cdate**
  La fecha en la que se registró el cambio.

En xCVS los cambios no sólo ocurren sobre los datos del archivo, sino que todo cambio en la estructura del repositorio también es registrado como un cambio. Por ejemplo, la copia de un archivo de un directorio a otro también es registrada como un cambio y se genera una delta para él. Es por eso que se tiene en esta tabla un campo `branch_id` y una restricción en la que sólo se permite que `branch_id` o (exclusivo) `file_id` sean nulos a la vez.

A.1.3. Manejo de diferencias

Los manejadores de diferencias de xCVS son, junto con los filtros, los encargados de la flexibilidad del sistema; es gracias a ellos que archivos con diferente contenido puedan ser tratados de una forma adecuada a la información que se tiene de ellos, aprovechando toda la meta-información de la que se dispone. Ambos tipos de manejadores tienen sus clases dentro del paquete `mx.unam.fciencias.conexa.xcvs.files.handlers`.

Tipos MIME

Los tipos MIME (Multipurpose Internet Mail Extensions) son un estándar de internet para el correo electrónico; casi todo el correo en la internet se transmite mediante SMTP [1] haciendo uso del formato MIME. Dado que SMTP sólo soporta caracteres ASCII de siete bits, es necesario hacer uso de algún método para enviar otro tipo de información en un correo, como lo son imágenes, películas, sonidos o programas. Dado que hay otros protocolos de comunicación que requieren que los datos sean transmitidos en un contexto similar al del correo (como lo es HTTP[2]), MIME se ha convertido en un componente fundamental de la internet. Los tipos MIME colocan a

[1]Simple Mail Transport Protocol.
[2]Hyper Text Transfer Protocol.

cada tipo de archivo en una categoría de la forma *tipo/subtipo* en la que se definen reglas para cifrar caracteres no pertenecientes a ASCII y así poder enviarlos por correo.Si el lector está interesado en obtener información más detallada sobre los tipos MIME puede consultar alguna de las siguientes direcciones:

- `http://www.iana.org/assignments/media-types/`
- `http://www.isi.edu/in-notes/rfc2045.txt`
- `http://www.isi.edu/in-notes/rfc2046.txt`

Son precisamente esas categorías en las que xCVS se basa para tomar decisiones sobre cómo debe tratar un archivo. Estas decisiones se encuentran en la forma de manejadores, los cuales se dividen en dos tipos:

- Manejadores de entrada/salida.
- Manejadores de diferencias.

Los manejadores de entrada/salida funcionan como filtros; si se tiene asignado un manejador de entrada para cierto tipo MIME cada vez que se agrega un archivo de ese tipo al repositorio esté es procesado por el filtro. Esto puede ser útil en el caso en el que se necesita tener consistencia dentro de todos los archivos de un módulo, por ejemplo si es necesario cumplir con ciertas convenciones de código para archivos de código fuente. Los filtros de salida permiten que el módulo defina cómo es que se verán los archivos cada vez que se obtiene una copia local.

Los manejadores de diferencias son los encargados de comparar dos versiones de un mismo archivo e identificar los cambios ocurridos entre uno y otro. Con esta información deberán ser capaces de generar una delta que convierta la versión anterior a la nueva. Para poder hacer esto de la mejor manera es necesario conocer a fondo sobre qué información estamos trabajando y eso lo logramos conociendo el tipo MIME.

Una de las clases que intervienen para poder identificar el tipo MIME de un archivo es la interfaz `mx.unam.fciencias.conexa.xcvs.files.MimeMatcher` que presenta los siguientes métodos:

- **String getMimeType(File input)**
  Obtiene la cadena de tipo/subtipo específica a este archivo.
- **String getMimeType(byte[] data)**
  Obtiene la cadena de tipo/subtipo específica a este archivo, representado con un arreglo de bytes.

- **String getMimeType(String name)**
  Obtiene la cadena de tipo/subtipo específica a este archivo, dependiendo de su nombre.

La clase concreta `mx.unam.fciencias.conexa.xcvs.files.XcvsMimeMatcher` implementa la interfaz mencionada anteriormente, y resuelve el tipo MIME en base a la extensión del archivo que recibe. Por esta razón el método que recibe un arreglo de bytes siempre regresa como resultado `null` pues es imposible obtener la extensión a partir de esa información.

La información correspondiente a los tipos MIME se almacena en dos tablas dentro del servidor, la primera de ellas es `xcvs.mime_type` que contiene la siguiente información:

- **id**
  El identificador de este tipo MIME.
- **name**
  La cadena tipo/subtipo que identifica a este tipo MIME.
- **owner_id**
  El identificador del usuario que agregó este tipo MIME.
- **in_handler_id**
  El manejador de entrada asociado a este tipo MIME.
- **out_handler_id**
  El manejador de salida asociado a este tipo MIME.
- **diff_handler_id**
  El manejador de diferencias asociado a este tipo MIME.

En esta tabla se relaciona un tipo MIME con sus manejadores por omisión; todo tipo MIME debe tener asociados éstos. Al inicio todos los tipos MIME tienen asociado como manejador de entrada/salida una instancia de `PassTroughHandler`, el cual simplemente deja pasar el archivo. El manejador de diferencias por omisión es el manejador de texto. Se pueden sobreescribir los manejadores asociados a cada archivo en su entrada en la tabla `xcvs.file`

La segunda tabla es `xcvs.mime_type_extension` la cual como su nombre indica es la encargada de asociar una extensión particular a un tipo MIME. Esta tabla contiene los siguientes campos:

- **id**
  El identificador de esta extensión.

- **mime_type_id**
  El tipo MIME asociado a esta extensión.
- **extension**
  La extensión *per se.*
- **owner_id**
  El identificador del usuario que agregó esta extensión.

La única restricción que se tiene en esta tabla es que las parejas `(mime_type_id, extension)` sean únicas.

Cada uno de los manejadores tiene almacenada dentro de la base de datos el bytecode de su clase en java. Esto hace necesario que cuando se resuelve el tipo MIME sea necesario cargar la clase y obtener una instancia de ella. Esto se logra haciendo uso de la clase `mx.unam.fciencias.conexa.xcvs.dbAccess.DatabaseClassLoader`, que es la encargada de establecer una conexión a la base de datos, obtener el bytecode y cargarlo a la máquina virtual de java. El proceso para llevar esto a cabo es el siguiente:

1. Se busca la clase en un cache para ver si se ha cargado previamente; si es así se regresa esa clase; en caso contrario continuamos con el siguiente paso.
2. Se revisa si la clase solicitada inicia con alguna de las cadenas "java." o "javax.". En ese caso asumimos que es una clase perteneciente al lenguaje, y por razones de seguridad debe ser resuelta por el cargador de clases primordial (el de la máquina virtual). En caso contrario continuamos con el siguiente paso.
3. Intentamos obtener la clase através de la base de datos y resolverla; en caso de no encontrarla o no poder resolverla continuamos con el siguiente paso.
4. Se intenta resolver la clase con el cargador de clases primordial. Si se falla en esta ocasión se lanza una excepción `ClassNotFoundException`.

La tabla encargada de los manejadores de entrada/salida es `xcvs.io_handler` que contiene los siguientes campos:

- **id**
  El identificador de este manejador.

- **name**
  El nombre de este manejador.

- **owner_id**
  El identificador del usuario que agregó este manejador.

- **bytecode_id**
  El identificador del bytecode de la clase de este manejador.

En la tabla `xcvs.diff_handler` se mantiene el registro de los manejadores como se ve a continuación:

- **id**
  El identificador del manejador.

- **owner_id**
  El usuario dueño de este manejador.

- **name**
  El nombre del manejador.

- **handler**
  El manejador *per se*.

A.2. Manejadores de entrada/salida

Además de los manejadores de diferencias, podemos tener filtros. Mediante la tabla `xcvs.io_handler` se asocia un filtro de entrada y de salida con un tipo MIME específico.

- **id**
  El identificador de la asociación del manejador como filtro.

- **name**
  El nombre del manejador de entrada/salida.

- **owner_id**
  El nombre del dueño de este manejador.

- **in_mtype_id**
  El tipo MIME que se asocia a este manejador como un filtro de entrada.

- **out_mtype_id**
  El tipo MIME que se asocia a este manejador como un filtro de salida.

- **handler**
  El manejador *per se*.

- **is_default**
  Indica si este manejador es el manejador por omisión para su tipo MIME.

Se crea una restricción para asociar este manejador sólo a un tipo de entrada o de salida, esto es no se permite que `in_mtype_id` y `out_mtype_id` sean nulos a la vez, o que ambos sean no nulos a la vez.

Además los manejadores se pueden aplicar como filtros, (de ahí el hecho que sea importante que un manejador tenga asociados tipos de entrada y de salida). Esto facilita el crear manejadores que hagan labores más "mundanasçomo por ejemplo, un usuario puede tener asociado un filtro al final de todas sus operaciones de checkout para que los archivos tengan cierto formato que a él le gusta más o el que sea el estándar de la compañía para la que trabaja.

A.3. Esquema xcvs

En este esquema están las tablas que relacionan todos los elementos de xCVS entre sí. La tabla `xcvs.module` contiene la información que integra un módulo, que es la siguiente:

- **id**
  Un identificador para el módulo.

- **owner_id**
  El identificador del dueño del módulo.

- **name**
  El nombre del módulo.

Cada módulo está constituido por al menos una ramificación; estas se encuentran definidas en la tabla `xcvs.branch`, con los siguientes campos:

- **id**
  Un identificador para la ramificación.

- **module_id**
  El módulo al que pertenece esta ramificación.
- **name**
  El nombre de la ramificación.

Además se tiene una restricción para que las parejas (`module_id`, `name`) sean únicas.

Cada una de estas ramificaciones tiene ciertos permisos asociados, los cuales se manejan en la tabla `xcvs.branch`, la que contiene la siguiente información:

- **user_id**
  El usuario al que se le otorgan permisos.
- **group_id**
  El grupo al que se le otorgan permisos.
- **branch_id**
  La ramificación sobre la que se están otorgando permisos.
- **owner_grantor_id**
  El identificador del usuario que otorgó permisos de pertenencia.
- **read_grantor_id**
  El identificador del usuario que otorgó permisos de lectura.
- **add_grantor_id**
  El identificador del usuario que otorgó permisos para agregar archivos.
- **branch_grantor_id**
  El identificador del usuario que otorgó permisos para ramificar.

Se tienen restricciones para que sólo se cree un renglón por usuario o por grupo, para cada uno de las ramificaciones.

Para almacenar las clases que utiliza xCVS se tiene la tabla `xcvs.bytecode` en la que es posible guardar bytecode para la máquina virtual de Java. Esta tabla contiene la siguiente información:

- **id**
  Un identificador para el bytecode.
- **owner_id**
  El identificador del dueño del bytecode.

- **name**
  El nombre del bytecode.
- **data**
  Los bytes que componen el bytecode.
- **jar_file**
  El archivo jar que contiene los bytes.

Para cada archivo o tipo MIME se puede definir un manejador con el que el archivo será procesado para su entrada o salida del repositorio. Estos manejadores están registrados en la tabla `xcvs.io_handler` la cual contiene los siguientes campos:

- **id**
  El identificador de este manejador.
- **name**
  El nombre de este manejador.
- **owner_id**
  El dueño de este manejador.
- **bytecode_id**
  El identificador de la clase de este manejador.

Además de los manejadores de entrada/salida es necesario almacenar los manejadores encargados de procesar las diferencias entre archivos; para eso se tiene la tabla `xcvs.diff_handler` la cual contiene los siguientes campos:

- **id**
  El identificador de este manejador.
- **name**
  El nombre de este manejador.
- **owner_id**
  El dueño de este manejador.
- **bytecode_id**
  El identificador de la clase de este manejador.

Cada archivo dentro de xCVS tiene asociado un tipo MIME que sirve para identificar cuáles son los manejadores que se deben utilizar para procesarlo. La tabla `xcvs.mime_type` es la encargada de relacionar esa información y contiene los siguientes campos:

- **id**
  El identificador del tipo MIME.
- **name**
  El nombre del identificador.
- **owner_id**
  El identificador del dueño de este renglón.
- **in_handler_id**
  El manejador de entrada.
- **out_handler_id**
  El manejador de salida.
- **diff_handler_id**
  El manejador de diferencias.

Para cada uno de los tipos MIME definidos en la tabla anterior es necesario asociarle una extensión, para lo que se creó la tabla `xcvs.mime_type_extension` con los siguientes campos:

- **id**
  El identificador de esta extensión.
- **mime_type_id**
  El tipo MIME al que se asocia esta extensión.
- **extension**
  La extensión *per se*
- **owner_id**
  El identificador del dueño de este renglón.

Se tiene una restricción para asociar sólo una extensión a cada tipo MIME.

La parte central de el manejo de versiones son los archivos; en xCVS éstos se representan mediante la tabla `xcvs.file`, que tiene los siguientes campos:

- **id**
  El identificador de este archivo.
- **revision**
  La versión de este archivo.

- **owner_id**
  El identificador del dueño de este archivo.
- **branch_id**
  A qué rama pertenece este archivo.
- **parent_id**
  Que archivo es el padre de éste.
- **diff_handler_id**
  El manejador de diferencias para este archivo.
- **input_handler_id**
  El manejador de entrada para este archivo.
- **output_handler_id**
  El manejador de salida de este archivo.
- **mtype_id**
  El tipo MIME de este archivo.
- **name**
  El nombre de este archivo.
- **data**
  La información de este archivo; puede ser `null` en cuyo caso este archivo es un directorio.

La única restricción que se tiene sobre esta tabla es para garantizar que las tuplas (`parent_id, name, branch_id`) no se repitan.

Para evitar recalcular el resumen de cada archivo cada vez que es necesario, éstos se almacenan en la tabla `xcvs.file_digest`, la cual contiene los siguientes campos:

- **id**
  El identificador de este resumen.
- **file_id**
  El identificador del archivo del cual éste es su resumen.
- **digest**
  El resumen del archivo.

Para manejar los permisos de cada archivo se tiene la tabla `xcvs.file_grant` la cual almacena la siguiente información:

- **user_id**
  El usuario al que se le conceden permisos.
- **group_id**
  El grupo al que se le conceden permisos.
- **file_id**
  El identificador del archivo sobre el cual se otorgan permisos.
- **read_grantor_id**
  El usuario que otorga permiso de lectura.
- **write_grantor_id**
  El usuario que otorga permiso de escritura.

Se tiene una restricción para que cada renglón se refiera (de forma exclusiva) a un usuario o a un grupo. Además se garantiza que las tuplas (`user_id, grour_id, file_id`) sean únicas.

Para registrar los cambios que ha sufrido cada uno de los archivos sujetos al control de versiones se tiene la tabla `xcvs.history` con los siguientes campos:

- **id**
  El identificador para este registro.
- **commiter_id**
  El usuario que realizó este cambio.
- **branch_id**
  La rama a la que pertenece este cambio.
- **file_id**
  El archivo que fue modificado.
- **old_revision**
  La versión que fue modificada.
- **new_revision**
  La nueva versión.
- **diff_handler_id**
  El manejador de diferencias que se debe utilizar para procesar este cambio.

- **delta**
  El guión para actualizar la versión anterior a ésta.
- **comment**
  El comentario que se hizo al realizar este cambio.
- **timestamptz**
  La hora a la que se realizó el cambio.

Se tiene una restricción para que cada cambio sea sólo sobre un archivo o sobre una rama.

Bibliografía

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504. ACM Press, 1996.

[CW03] Jonathan Kaplan Crawford William. *J2EE Design Patterns*. O'Reilly, 2003.

[Eli02] Gurovich Elisa. *Teoría de la Computación, notas de clase*. Facultad de Ciencias, UNAM, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[gnu] Sitio oficial de gnu. `http://www.gnu.org`.

[Wik] Wikipedia, la enciclopedia gratis que todos pueden editar. `http://en.wikipedia.org/wiki/Main_Page`.

[xst] Xstream, serializador de objetos en java. `http://xstream.codehaus.org/index.html`.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.