Generación automática de código con WebRatio

Memoria

Javier Martín Rubio
Ingeniería Informática

Jordi Ceballos Villach
Consultor

18 de junio de 2007

A todas las personas que me quieren, por su comprensión
A Jordi Ceballos, por su apoyo moral.

Resumen

El desarrollo de aplicaciones Web a partir de modelos facilita la automatización y seguridad de la calidad de los productos resultantes. Existen, no obstante, algunas cuestiones relacionadas con los modelos navegacionales en las que es necesario avanzar un poco más, una de ellas sería la calidad de los modelos navegacionales con operaciones que modifican contenidos de una base de datos.

Una de las propuestas [2] es incluir en estos modelos dos nuevas propiedades de calidad: corrección y completitud, y mediante éstas verificar si los modelos navegacionales obtenidos son conformes a sus respectivos modelos de datos. La primera propiedad determinaría si el modelo navegacional contiene todas las operaciones que permitan modificar cada uno de los datos del modelo conceptual, la segunda, si los caminos navegacionales dejan los datos en un estado consistente.

En este proyecto han sido analizados los fundamentos teóricos y algoritmos que propugnan sus autores [2], aunque el trabajo se ha centrado en el diseño e implementación de un programa en Java que permita obtener las propiedades mencionadas y en la resolución de forma satisfactoria y con cierto grado de optimización de aquellas cuestiones que, desde la perspectiva de diseño e implementación, pueden ayudar a mejorar todas las etapas previas a la obtención del modelo navegacional: cálculo de dependencias, grafo navegacional y refactorización.

El programa resultado de este proyecto permite obtener un modelo navegacional a partir de un modelo conceptual; posteriormente, este modelo navegacional podrá ser utilizado para generar de forma automática el código Java de la aplicación Web, permitiendo ésta última a los investigadores interesados comprobar los fundamentos y algoritmos propuestos [2]. Para la obtención del modelo conceptual y el código de la aplicación es necesario utilizar la herramienta WebRatio, ya que el programa implementado está diseñado para operar exclusivamente con la metodología WebML.

Palabras clave:

Modelo de datos, modelo conceptual, modelo navegacional, grafo navegacional, refactorización, WebML, WebRatio.

Área del PFC:

Ingeniería de Programación

Índice de contenidos

1. Introducción 7
1.1. Justificación del PFC y contexto en el que se desarrolla 7
1.2. Punto de partida 7
1.3. Aportación del PFC 7
1.4. Objetivos del PFC 7
1.5. Enfoque y método seguido 8
1.6. Planificación del proyecto 8
1.7. Productos obtenidos 11
1.8. Descripción del resto de capítulos de la memoria 12
2. Arquitectura 13
2.1. Introducción 13
2.2. WebRatio y WebML 13
2.2.1. Descripción general 13
2.3. Requisitos 14
2.3.1. Requisitos funcionales del programa 14
2.3.2. Requisitos no funcionales 15
2.4. Componentes del proyecto 16
2.5. Diseño de base de datos 17
2.6. Diseño de los componentes 17
3. Diseño general de la aplicación 18
3.1. Introducción 18
3.2. Consideraciones previas 18
3.3. Estructura general 18
3.3.1. Clases principales 19
3.3.2. Procedimiento general para la obtención del modelo navegacional 19
3.3.3. Diagrama de clases 20
3.4. Árbol de operaciones 21
3.4.1. Introducción 21
3.4.2. Consideraciones previas 21
3.4.3. Proceso general de creación del árbol de operaciones 21
3.5. Grafo operacional 23
3.5.1. Introducción 23
3.5.2. Consideraciones previas 23
3.5.3. Procedimiento para la obtención del *modelo navegacional* a partir de un *grafo operacional* 23
3.5.4. Diagrama de clases 24
3.5.5. Refactorización 25
3.5.6. Procedimiento para la obtención del grafo operacional 28
3.5.7. Grafo navegacional 29
3.6. Modelo navegacional 34
3.6.1. Diagrama de clases 34
3.6.2. Modelado de los elementos navegacionales 35
3.6.3. Procedimiento para la obtención del modelo navegacional 36
4. Detalles de diseño 37
4.1. Introducción 37
4.2. Diccionario de dependencias 37
4.2.1. Consideraciones previas 37

4.2.2. Diagrama detallado de clases 38
4.2.3. Procedimiento para la obtención del diccionario de dependencias 39
4.3. Caminos operacionales 40
4.4. Árbol de operaciones 41
4.4.1. Diagrama de clases 41
4.5. Tratamiento de los modelos definidos en WebRatio 43
4.5.1. Consideraciones previas 43
4.5.2. Diagrama general detallado 44
4.5.3. Funcionalidades generales de la clase principal WebMLBuilder 45
5. Conclusiones 47
6. Líneas de desarrollo futuro 48
6.1. Ampliación de las funcionalidades del modelo navegacional 48
6.2. Ampliación de las funcionalidades del programa 48
6.3. Mejorar el algoritmo de cálculo de dependencias 48
7. Glosario 49
8. Bibliografía y referencias 50
9. Anexo 1: Herramientas utilizadas 51

Índice de figuras

Fig. 2.1. Fases para la generación automática del código Java de la aplicación Web.... 14
Fig. 2.2. Arquitectura general del proyecto 17
Fig. 3.1. Diseño de la estructura general del programa 20
Fig. 3.2. Árbol de operaciones (ejemplo) 21
Fig. 3.3. Árbol de operaciones (diagrama general de clases) 22
Fig. 3.4. Grafo operacional/navegacional (diagrama de clases) 24
Fig. 3.5. Algoritmo Head-Merge 26
Fig. 3.6. Algoritmo Tail-Merge 27
Fig. 3.7. Representación de un grafo operacional 29
Fig. 3.8. Representación de un camino navegacional en WebRatio (vista parcial) 30
Fig. 3.9. Modelo navegacional (diagrama de clases) 34
Fig. 4.1. Estructura interna del diccionario (ejemplo) 37
Fig. 4.2. Obtención del diccionario (diagrama detallado de clases) 38
Fig. 4.3. Algoritmo de procesamiento de las dependencias 40
Fig. 4.5. Árbol de operaciones (diagrama detallado de clases) 41
Fig. 4.6. Clases envolventes (diagrama) 43
Fig. 4.7. Tratamiento de los modelos definidos en WebRatio (diagrama detallado de clases) 44
Fig. 4.8. Código Java del método *outputNavigationModel* 46

1. Introducción

1.1. Justificación del PFC y contexto en el que se desarrolla

La complejidad que supone la construcción de aplicativos Web ha propiciado la aparición de diferentes técnicas de ingeniería de software que tienen como objetivo la automatización de algunas tareas de desarrollo y la sistematización de la calidad del software. En la mayoría de los casos estas mejoras consisten en la aplicación de una determinada metodología de desarrollo, algo que puede llegar a simplificar algunas etapas del ciclo de vida, como la de diseño o de codificación, pero que continúan exigiendo una importante intervención manual.

Existen varias propuestas para el desarrollo de aplicaciones Web a partir de modelos, permitiendo diseñar un aplicativo considerando las distintas vistas y representaciones, coincidiendo cada una de éstas con las capas en las que suele descomponerse la estructura de un aplicativo: datos, presentación, navegación, ...

Entre las metodologías más ampliamente aceptadas y utilizadas en la actualidad están las siguientes: OO-H (Object-Oriented Hypermedia), WebML (Web Modeling Language) y UWE (UML-based Web Engineering), derivando éstas en sendas herramientas: VisualWade, WebRatio y ArgoUWE. Aunque el grado de automatización puede considerarse elevado, todas ellas requieren sean modeladas manualmente las diferentes capas de las que se compone una aplicación.

1.2. Punto de partida

En la automatización a la que se hacía referencia aún quedarían muchas cuestiones por mejorar, entre otras, la generación automática de restricciones de integridad [1] o la calidad de los modelos de navegación en entornos en los que se producen modificaciones de los contenidos de las bases de datos [2].

Respecto a la calidad de estos modelos de navegación, según sus autores [2] la introducción de algunas propiedades adicionales podría llegar a asegurar que estos modelos contienen todas las operaciones necesarias, evitando así el riesgo de dejar la base de datos en un estado inconsistente.

1.3. Aportación del PFC

La aportación de este PFC consistirá en el desarrollo de un programa informático que verifique los fundamentos teóricos y los algoritmos propuestos por sus autores [2] y proporcionar una herramienta para la experimentación.

1.4. Objetivos del PFC

El objetivo principal de este proyecto será desarrollar un programa que sea capaz de generar completamente el grafo y modelo navegacional de una aplicación a partir exclusivamente del modelo de datos.

Este objetivo exige de la adopción de alguna metodología y herramienta, ya que es necesario comprobar que el grafo navegacional resultante es completo[1] y correcto[2].

[1] El modelo navegacional debe tener como mínimo un camino que permita modificar cada uno de los datos del modelo conceptual.
[2] Todo camino debe dejar los datos en un estado consistente.

También se considera importante proporcionar una visión general de la estructura de los ficheros XML creados por WebRatio, ya que ello facilitará la incorporación de nuevas funcionalidades al programa resultado de este PFC y avanzar en la línea de investigación abierta.

Se establece la necesidad de disponer de algún mecanismo que permita, sin necesidad de otras aplicaciones, obtener y representar el grafo refactorizado así como el resto de información de interés manejada por el programa. La implementación de este mecanismo forma parte del objetivo principal al que se hacía referencia.

Queda fuera del alcance de este proyecto considerar todas las posibilidades de WebML y WebRatio relacionadas con el modelo de presentación y navegación, así como tampoco su óptimo tratamiento.

1.5. Enfoque y método seguido

Las restricciones temporales y las características de este proyecto han requerido de una metodología que centra gran parte del esfuerzo en la codificación.

Al tratarse de un proyecto de investigación existe una alta probabilidad de que puedan aparecer cuestiones que obliguen a la introducción de modificaciones sustanciales en el diseño, siendo éste el motivo por el que se ha dado prioridad al diseño e implementación de una estructura flexible que facilite la incorporación de variaciones o modificaciones del diseño original, especialmente con aquellas partes responsables de obtener el grafo y modelo navegacional.

Las cuestiones organizativas han dirigido en cierta manera el método a seguir, ya que el establecimiento inicial de plazos y tareas a realizar, orientadas éstas últimas a la entrega de piezas de código totalmente funcionales, obliga a iniciar el desarrollo sin tener conocimientos suficientes de las especificaciones, funcionalidades y posibilidades de alguna de las herramientas utilizadas.

Las problemáticas aparecidas durante la primera fase, todas ellas relacionadas con WebRatio, introdujeron riesgos importantes en el proyecto, llegando en algún momento a replantear los objetivos iniciales, así como también la duración y orden de ejecución de las tareas planificadas.

El método seguido se asemeja a la metodología XP (eXtreme Programming), orientada a evaluar los progresos y a ajustar la dirección del esfuerzo en cada una de las etapas programadas.

1.6. Planificación del proyecto

La mayoría de tareas especificadas en la planificación están muy directamente relacionadas con el diseño y codificación del programa, siendo las tareas principales todas aquellas que han supuesto un mayor esfuerzo y dedicación:

1. Estudio preliminar de WebRatio y ficheros XML generados (Id. Tarea 1)

   Con este estudio se pretende familiarizarse con esta herramienta y determinar la estructura de los ficheros XML generados con modelos conceptuales complejos.

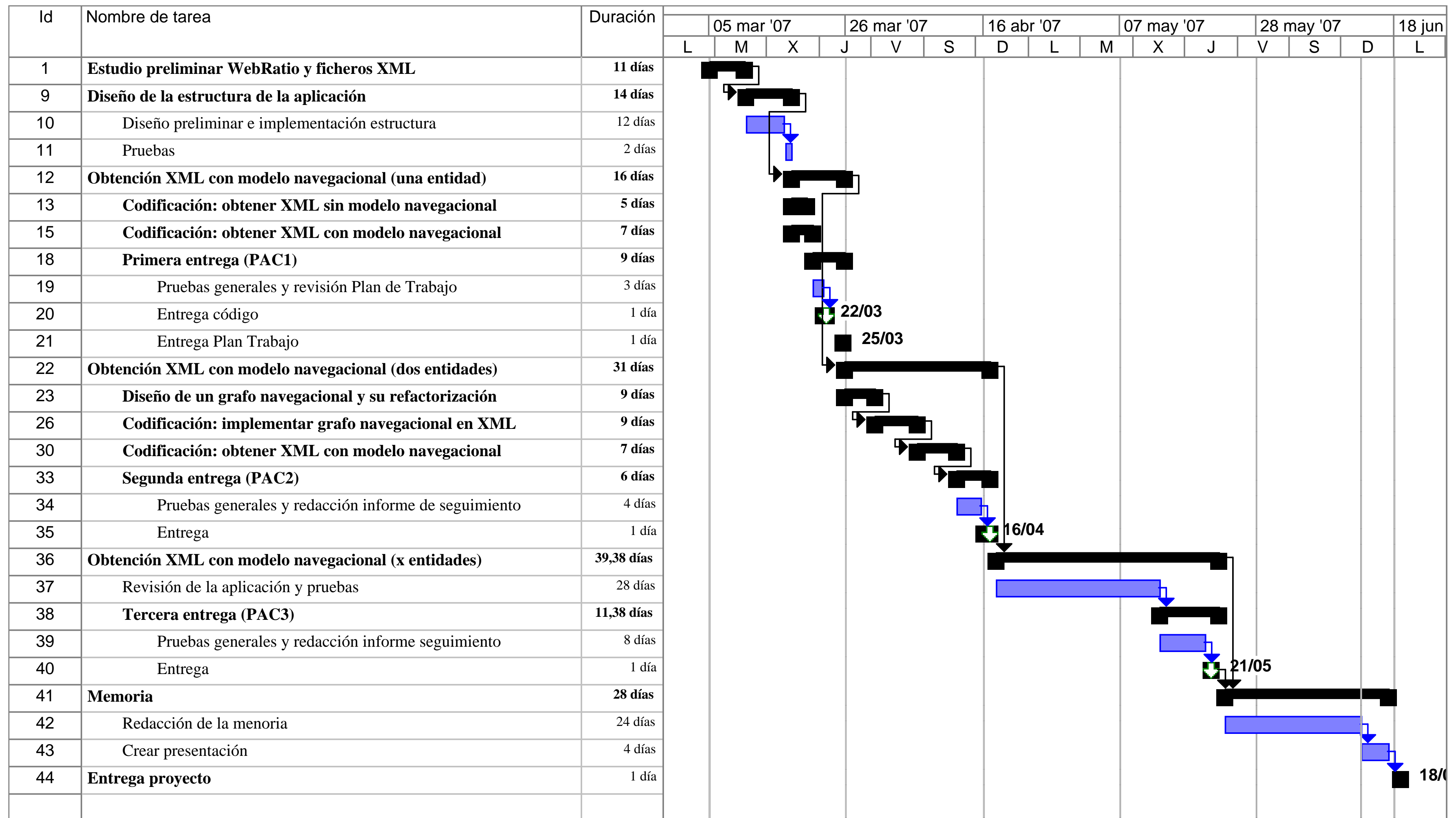
2. Diseño preliminar de la estructura del aplicativo a implementar (Id. Tarea 9)

   Con la información obtenida en el paso anterior se diseña una estructura de aplicativo que facilita la incorporación progresiva de un mayor número de funcionalidades, haciendo uso de patrones de diseño, básicamente Builder y Composite.

3. Plantear y diseñar un grafo navegacional y la refactorización del mismo (Id. Tarea 23)

   Considerando las diferentes dependencias entre las entidades del modelo conceptual, plantear un grafo que permita obtener caminos de navegación correctos.

4. Implementación completa del algoritmo que genera el grafo navegacional en formato XML para un número indeterminado de entidades. (Id. Tarea 36)



Id	Nombre de tarea	Duración
1	**Estudio preliminar WebRatio y ficheros XML**	**11 días**
9	**Diseño de la estructura de la aplicación**	**14 días**
10	Diseño preliminar e implementación estructura	12 días
11	Pruebas	2 días
12	**Obtención XML con modelo navegacional (una entidad)**	**16 días**
13	**Codificación: obtener XML sin modelo navegacional**	**5 días**
15	**Codificación: obtener XML con modelo navegacional**	**7 días**
18	**Primera entrega (PAC1)**	**9 días**
19	Pruebas generales y revisión Plan de Trabajo	3 días
20	Entrega código	1 día
21	Entrega Plan Trabajo	1 día
22	**Obtención XML con modelo navegacional (dos entidades)**	**31 días**
23	**Diseño de un grafo navegacional y su refactorización**	**9 días**
26	**Codificación: implementar grafo navegacional en XML**	**9 días**
30	**Codificación: obtener XML con modelo navegacional**	**7 días**
33	**Segunda entrega (PAC2)**	**6 días**
34	Pruebas generales y redacción informe de seguimiento	4 días
35	Entrega	1 día
36	**Obtención XML con modelo navegacional (x entidades)**	**39,38 días**
37	Revisión de la aplicación y pruebas	28 días
38	**Tercera entrega (PAC3)**	**11,38 días**
39	Pruebas generales y redacción informe seguimiento	8 días
40	Entrega	1 día
41	**Memoria**	**28 días**
42	Redacción de la menoria	24 días
43	Crear presentación	4 días
44	**Entrega proyecto**	1 día

1.7. Productos obtenidos

Los productos obtenidos resultado de este proyecto son los siguientes:

1. Paquete con el programa completo, documentación y archivos complementarios.

   El diseño modular de este programa facilita la reutilización de código y utilizar algunas de sus partes en otros proyectos con características similares. Los paquetes que forman parte de este programa también tendrán la consideración de productos.

   Contenido principal del paquete

Producto	Descripción
Paquete uoc	Programa completo que genera el modelo navegacional de una aplicación
Paquete uoc.tree	Clases para la creación del árbol de operaciones, tratamiento de caminos operacionales y sus dependencias.
Paquete uoc.graph	Clases para la creación y tratamiento de grafos navegacionales
Paquete uoc.model	Clases para obtención y procesamiento de los modelos definidos en WebRatio
Javadocs	Documentación de las clases implementadas
Archivos DTD	Definen la estructura y sintaxis esperada por el programa.

   Este paquete también incluye scripts para la creación y carga inicial de la base de datos con datos para hacer pruebas y ficheros con distintos modelos conceptuales creados con WebRatio.

2. Documento con la sintaxis de los ficheros XML generados por WebRatio.

   Este documento analiza en profundidad la sintaxis utilizada por WebRatio y los elementos contemplados en los modelos de datos y navegacionales.

3. Manual de instalación y experimentación.

   Describe los procedimientos a seguir para el correcto procesamiento de un modelo conceptual, obtención del modelo navegacional, generación automática de código y ejecución de la aplicación resultante.

1.8. Descripción del resto de capítulos de la memoria

Capítulo 2: Arquitectura. Se presentan de forma resumida los requisitos funcionales y no funcionales del proyecto, incluyendo una descripción general del funcionamiento de sus componentes. Se justifica la elección de WebRatio como una de las herramientas más apropiadas para este proyecto.

Capítulo 3: Diseño general del programa. Se presenta la estructura general del programa, los paquetes que la forman, decisiones de diseño, funcionalidades específicas de las clases principales y descripción de los procedimientos que se consideran más importantes.. Se incluyen apartados específicos para aquellos elementos directamente vinculados con la obtención de un modelo navegacional: árbol de operaciones, grafo operacional y grafo navegacional.

Capítulo 4: Detalles de diseño. Se presenta de forma mucho más detallada algunos elementos fundamentales en la obtención del modelo navegacional: diccionario de dependencias, caminos operacionales y árbol de operaciones. De todos ellos se muestran los diagramas de clases correspondientes, funcionalidades específicas de las clases principales y descripción de los procedimientos que se consideran más importantes. Para finalizar, se incluye un apartado que presenta la estructura general del paquete que genera el modelo navegacional, decisiones de diseño y descripción de los procedimientos más relevantes.

En los últimos capítulos se presentan las conclusiones y líneas de desarrollo futuras.

Anexo 1: Herramientas utilizadas. Se relacionan todas las herramientas y aplicativos utilizados en este proyecto y una breve descripción de éstos.

2. Arquitectura

2.1. Introducción

Con este proyecto se persigue obtener el modelo navegacional de una aplicación a partir de su modelo de datos y posteriormente generar automáticamente el código de la aplicación Web. Esta última permitirá comprobar la correcta ejecución de las secuencias de operaciones a la hora de insertar o borrar filas en las tablas de la base de datos y el estado en la que se encuentra ésta una vez completada la secuencia.

La obtención de un producto con todas las características mencionadas sería complejo de implementar y, en este caso, totalmente innecesario, ya que es posible apoyarse en herramientas que disponen de la mayor parte de las funcionalidades requeridas, concretamente la de diseñar el modelo conceptual y la de generar automáticamente el código de la aplicación a partir de una especificación.

La elección de una herramienta como WebRatio se considera fundamental para establecer con precisión el alcance, requerimientos y la arquitectura de este proyecto

2.2. WebRatio y WebML

En este caso se ha optado por la herramienta WebRatio y la metodología WebML. Los motivos que han primado en la elección se resumen en lo siguiente:

- Dispone de las funcionalidades necesarias para diseñar modelos de datos y navegacionales utilizando una metodología específica: WebML.
- WebRatio crea un único fichero XML con todos los modelos definidos. La sintaxis de estos ficheros no puede considerarse excesivamente complicada.
- Permite utilizar prácticamente cualquier gestor de base de datos.
- A petición del usuario, crea en la base de datos y de forma automática las tablas, atributos y relaciones indicados en el modelo conceptual.
- Genera automáticamente el código Java de una aplicación Web a partir del modelo navegacional.
- La metodología WebML la forman un conjunto de elementos no demasiado extenso, lo que facilita su estudio y aprendizaje.

Aparte de todo esto, también se ha valorado muy positivamente la puntuación obtenida en el estudio realizado por ingenieros de la Universidad de Extremadura [3].

2.2.1. Descripción general

El proceso completo para obtener de forma automática el código de la aplicación Web consta de tres fases claramente diferenciadas:

Fase 1. Se crea un modelo de datos con WebRatio. Esto crea un fichero XML con el modelo de datos y otra información adicional.

Fase 2. El programa implementado carga este fichero XML en una estructura acorde con las necesidades y, partiendo del modelo datos obtenido en la Fase 1, crea un grafo navegacional refactorizado y su correspondiente modelo navegacional. Para finalizar, el programa crea un fichero XML con todos los datos necesarios para que pueda ser procesado por WebRatio.

Fase 3. Se abre el fichero XML resultado de la Fase 2 con WebRatio y se genera el código de la aplicación Web.

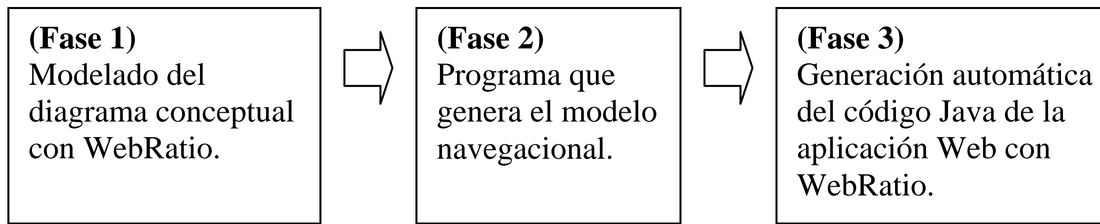


Fig. 2.1. Fases para la generación automática del código Java de la aplicación Web

2.3. Requisitos

Resueltas por WebRatio las cuestiones relacionadas con el diseño y obtención del modelo conceptual y la generación automática del código de la aplicación Web, las funcionalidades específicas que deberá ofrecer el programa a diseñar serán las que se describen a continuación.

2.3.1. Requisitos funcionales del programa

1. Con capacidad de obtener y procesar directamente el modelo de datos y resto de información contenida en el fichero XML generado por WebRatio. A partir de esta información, y sin intervención manual, debe crear el modelo navegacional.

2. Debe generar el modelo navegacional en un fichero XML respetando la sintaxis y estructura definidas en los DTD correspondientes. Este fichero debe poder ser procesado posteriormente por WebRatio y, a partir del modelo navegacional que contiene, generar automáticamente el código Java de la aplicación Web.

3. El modelo navegacional debe ser obtenido a partir de un grafo navegacional refactorizado, siendo necesario contemplar los procesos de refactorización indicados (Head_Merge y Tail_Merge).

4. Se debe añadir al modelo navegacional toda aquella información considerada necesaria y que permita lo siguiente:

   - Visualizar en WebRatio el modelo navegacional resultante y los elementos de los que se compone, distinguiendo claramente cada uno de ellos y sin superposición.

   - Identificar apropiadamente todas las páginas, operaciones, enlaces y resto de elementos contenidos en el modelo navegacional. Estos identificadores deberán aparecer en el aplicativo Web que se genere.

5. Poder extraer directamente con el programa una representación de lo siguiente:

   - Representación del diccionario: operaciones y dependencias; el formato para las dependencias será similar al indicado [2].

   - Caminos operacionales y grafo navegacional de todas las operaciones que permita el modelo de datos.

6. En la medida de lo posible, contemplar en el diseño el resto de elementos y consideraciones descritos [2].

2.3.2. Requisitos no funcionales

Los indicados expresamente para este proyecto son los siguientes:

- Implementación de un programa que pueda ser desplegado en equipos con sistemas operativos Windows o Linux indistintamente.
- Programa modular minimizando las dependencias entre los módulos de los que se componga.

Los condicionados por la elección de Webratio son los siguientes:

- Servidor Tomcat para la ejecución y pruebas del aplicativo Web generado por WebRatio.
- Base de datos que disponga de driver JDBC (Java Database Connectivity).

2.4. Componentes del proyecto

En el diseño de la estructura se ha prestado especial atención a todas aquellas cuestiones que tienen como objetivo minimizar las dependencias entre los componentes o módulos de los que se compone el programa.

En la Fig. 2.2 se muestra la estructura a la que se hace referencia y en la que podemos apreciar que la aplicación se encuentra encapsulada dentro de un gran componente llamado *uoc*, conteniendo éste una serie de paquetes adicionales con las clases necesarias para resolver satisfactoriamente las problemáticas específicas de este proyecto:

- Paquete *tree*

  Responsable de obtener los caminos y operaciones de las entidades y relaciones definidas en el *modelo de datos*.

- Paquete *graph*

  Responsable de crear un grafo a partir de las operaciones establecidas, así como también de su *refactorización*.

- Paquete *model*

  Responsable de leer el fichero XML de entrada (*SDSProject.xml*) con el modelo de datos y de crear un fichero XML de salida (*SDSProject-out.xml*) con el *modelo navegacional*; ambos cumpliendo la sintaxis indicada [14].

  Las clases que contiene hacen uso del paquete *org.jdom*.

Con el componente WebRatio se define el *modelo de datos* utilizando la metodología WebML [4] y se genera el código de la aplicación Web a partir del *modelo navegacional*.

El paquete de utilidad *org.jdom* es imprescindible para la correcta ejecución del aplicativo, ya que incorpora todo el código necesario para cargar y procesar estructuras arbóreas, así como también para su posterior conversión a formato XML.

Este proyecto se centra en el diseño e implementación del componente *uoc*.

En los siguientes capítulos se proporcionará una visión mucho más detallada de estos componentes y sus funcionalidades específicas.

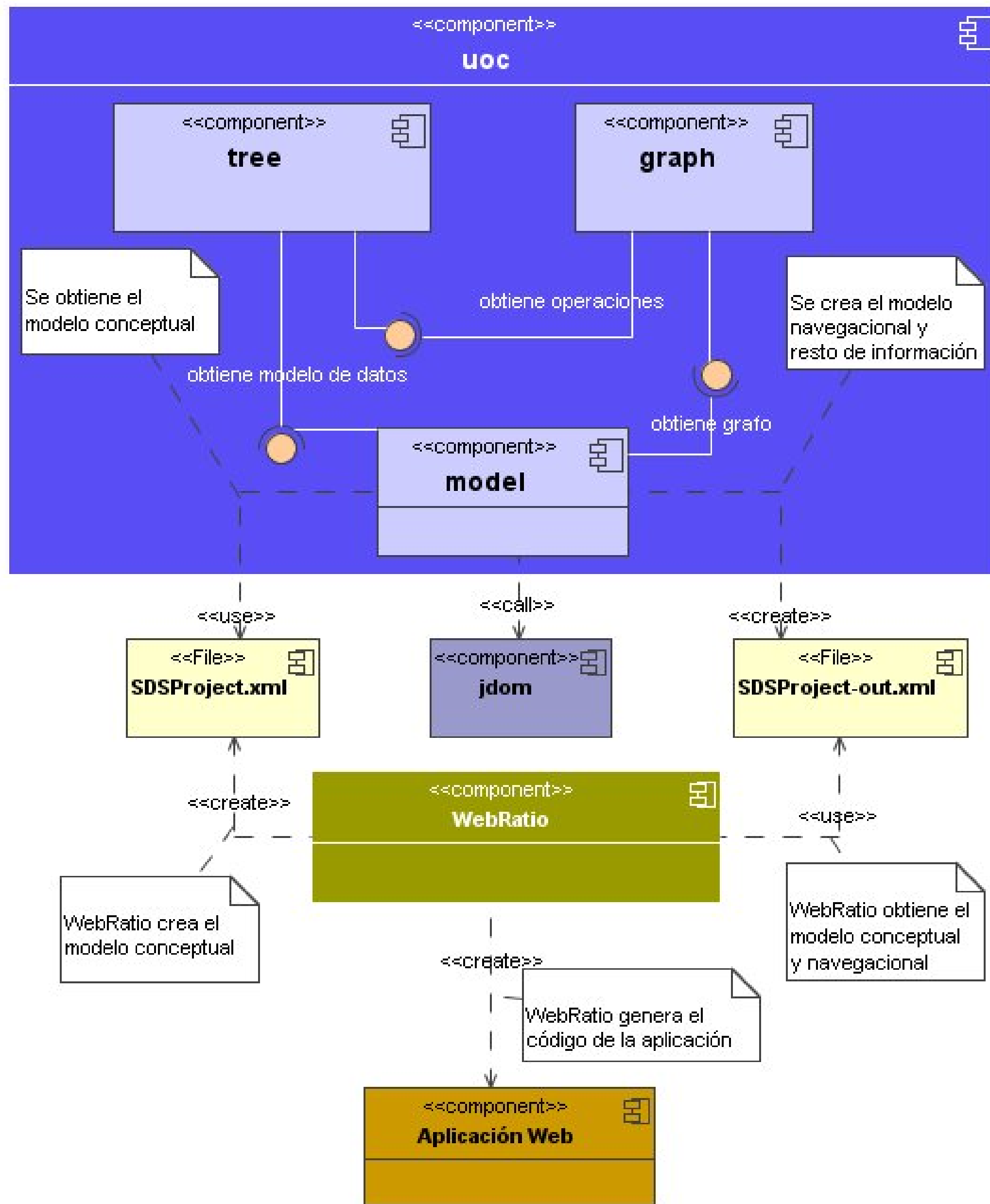


Fig. 2.2. Arquitectura general del proyecto

2.5. Diseño de base de datos

En este proyecto no se requiere diseñar la base de datos, ya que entre las funcionalidades de WebRatio se encuentra la de crear todos los elementos necesarios partiendo del *modelo de datos.*

2.6. Diseño de los componentes

Se describen con detalle en el apartado 3 y 4.

3. Diseño general de la aplicación

3.1. Introducción

En los siguientes apartados se exponen con cierto nivel de detalle los mecanismos utilizados para dotar al programa de las funcionales requeridas. A diferencia de otros proyectos, en éste las funcionalidades de las que se dotará al mismo serán, por llamarlo de alguna manera, de bajo nivel o estructurales, no siendo perceptibles para el usuario final. Esto tiene sus implicaciones en el contenido y estructuración de esta memoria, ya que la mayor parte de los apartados se centrarán en estas funcionalidades, resultando imprescindible entrar en un cierto nivel de detalle con objeto de exponer claramente las soluciones ideadas para conferir al programa las funcionalidades requeridas.

Muchas de estas funcionalidades vienen determinadas por los algoritmos y fundamentos teóricos expuestos [2], por lo que es conveniente disponer de la documentación correspondiente.

3.2. Consideraciones previas

Con objeto de evitar confusiones se cree oportuno introducir varios conceptos adicionales: diccionario de dependencias, caminos operacionales y árbol de operaciones; siendo explicados posteriormente con mayor detalle en los apartados correspondientes.

- **Diccionario de dependencias**: contiene todas aquellas operaciones definidas sobre alguna entidad (*InsertET, DeleteET*) [3] o relación (*InsertRT y DeleteRT*) del *modelo de datos*, así como las *dependencias* de cada una de éstas.

- **Camino operacional**: camino formado exclusivamente por operaciones, sin ningún otro elemento adicional.

- **Árbol de operaciones**: estructura que contiene todo los posibles caminos operacionales.

- **Grafo operacional**: grafo creado a partir de un árbol de operaciones y sin información navegacional.

3.3. Estructura general

La estructura general mostrada en la Fig. 3.1 coincide en su mayoría con el diagrama de componentes de la Fig. 2.2, sustituyendo los componentes por paquetes, ya que en el lenguaje Java está mucho más extendido el concepto de paquete. Esto no representa una variación de la especificación original, ya que se sigue manteniendo la forma de acceder a los paquetes a través de *Interfaces,* minimizando así las dependencias entre ellos.

En el apartado anterior se describieron, a grandes rasgos, las funciones generales de estos paquetes, siendo conveniente en este momento concretar un poco más.

- Paquete *tree*

  Responsable de obtener el *árbol de operaciones* a partir del *diccionario de dependencias* y los *caminos de operaciones*.

- Paquete *graph*

  Responsable de crear un *grafo operacional* refactorizado a partir del *árbol de operaciones*.

[3] La operación Update no se ha tenido en cuenta por no presentar interés respecto a los objetivos de este proyecto

- Paquete *model*

  Crea una estructura arbórea en la que se carga el *modelo de datos* y resto de información del fichero procedente del XML de entrada. Transforma el *grafo operacional* en un *grafo navegacional* y, a partir de éste, obtiene el *modelo navegacional*. Con el modelo de datos, modelo navegacional y resto de información obtiene el fichero XML de salida y preparado para ser procesado por WebRatio y generar la aplicación Web.

- Paquete *common*

  Contiene aquellas clases –en su mayoría *Interfaces*- comunes a todos los paquetes mencionados.

Para que los paquetes aquí descritos puedan hacer uso de los objetos o clases de los restantes deben utilizar las clases del paquete *common*. Esto facilitará la incorporación de nuevas funcionalidades o la revisión de las existentes, ya que se minimizan las dependencias entre paquetes.

Cada uno de estos paquetes se explican con mayor detalle en los apartados correspondientes.

3.3.1. Clases principales

Una característica a resaltar es que cada una estas clases principales dispone de un único punto de entrada, siendo estos puntos las clases que se muestran: *OperationsTree, Graph, WebMLBuilder*. De esta última cabe destacar que, dado el número y complejidad de los objetos a crear, se ha creído oportuno apoyarse en el patrón Builder, aunque con una variación: se delega la creación de objetos a clases específicas en función del objeto a crear y su localización en la estructura del fichero XML de salida.

La clase *Main* es la responsable de ejecutar las operaciones establecidas en el orden descrito (apartado 3.3.2), así como también de iniciar la aplicación y de crear instancias de las clases mencionadas.

3.3.2. Procedimiento general para la obtención del modelo navegacional

Una cuestión importante que no se muestra en el diagrama es el orden en la creación de las instancias: para poder obtener el *árbol de operaciones* es necesario, en primer lugar, disponer de las entidades y relaciones participantes, pero éstas no estarán disponibles hasta que no se haya cargado el fichero XML de entrada. La solución a esta problemática se resuelve respetando una determinada secuencia de procesamiento:

Fase 1. Leer el fichero XML con el modelo de datos definido en WebRatio.

Este proceso crea las estructuras necesarias, y prepara las entidades y relaciones para que sean manipuladas con comodidad (clases envolventes en Fig. 4.6) así como también crea los elementos principales del *modelo navegacional*.

Fase 2. Obtención del *árbol de operaciones* (apartado 3.4) a partir del *modelo de datos* recuperado de la fase anterior.

Fase 3. Obtención de un *grafo operacional refactorizado* a partir del *árbol de operaciones* de la fase anterior.

Fase 4. Añadir información específica a los nodos del grafo: operaciones en lenguaje WebML y enlaces (links) de los nodos finales.

El grafo resultante tendrá la consideración de *grafo navegacional*, ya que contiene toda la información necesaria para construir el *modelo navegacional* una vez hayan sido establecidos todos los enlaces entre las páginas.

Fase 5. Creación del *modelo navegacional*.

Se completa el contenido del elemento estructural *Navigation* [14] a partir del *grafo navegacional*.

Fase 6. Obtención del fichero XML de salida.

El contenido de este fichero coincide con el XML de entrada, pero al cual se le ha añadido el *modelo navegacional*.

3.3.3. Diagrama de clases

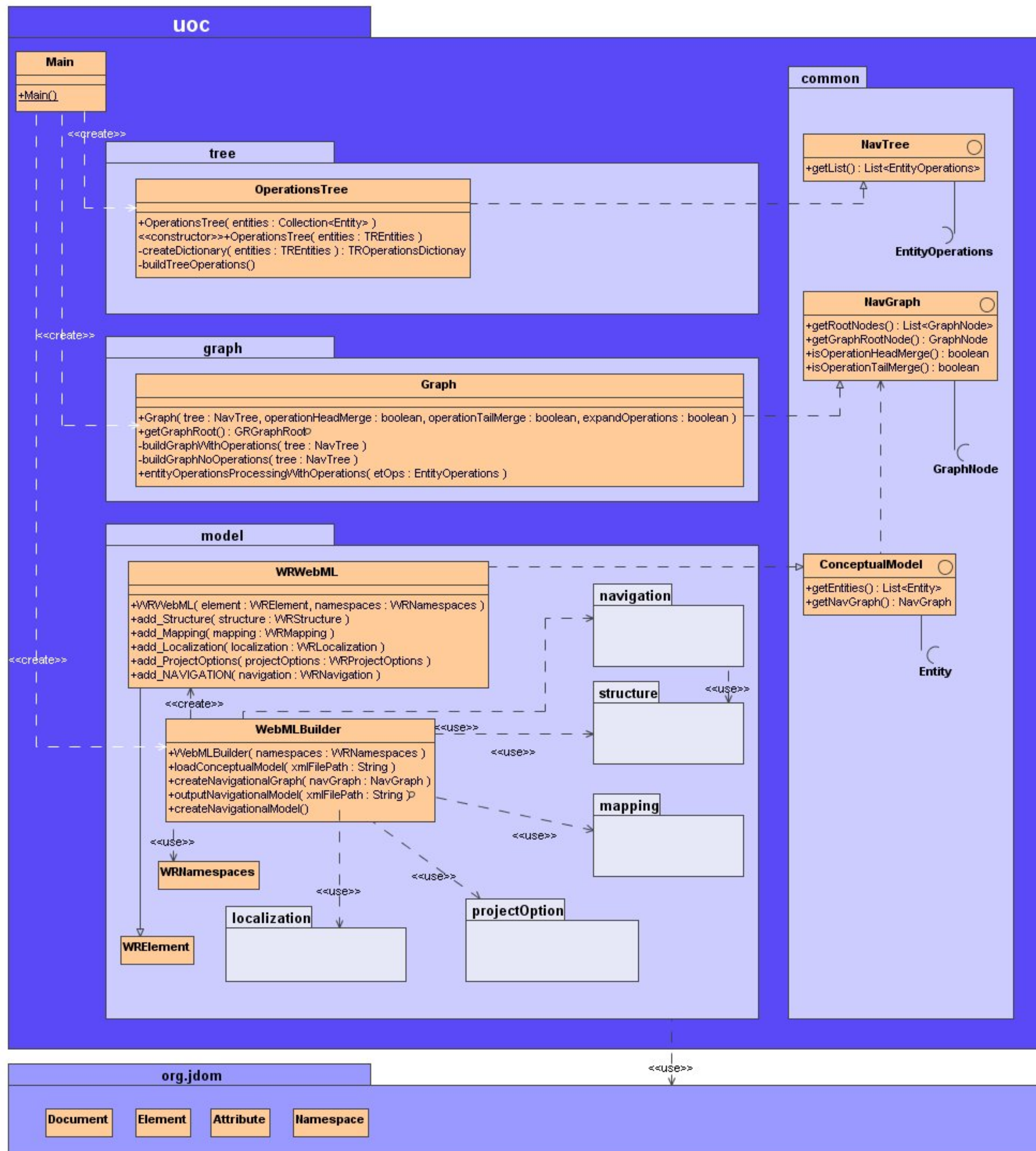


Fig. 3.1. Diseño de la estructura general del programa

3.4. Árbol de operaciones

3.4.1. Introducción

Para cada modelo de datos existirá un único *árbol de operaciones*, cuya estructura será similar a la de la Fig. 3.2. Este árbol estará formado por todos los *caminos operacionales* posibles, existiendo como mínimo un camino operacional para cada una de las entidades del modelo de datos y correspondientes operaciones (*InsertET, DeleteET*). Esto último implica que un modelo de datos, por ejemplo, con dos entidades – con o sin relación que las una - tendrá un mínimo de cuatro caminos (2 entidades x 2 operaciones).

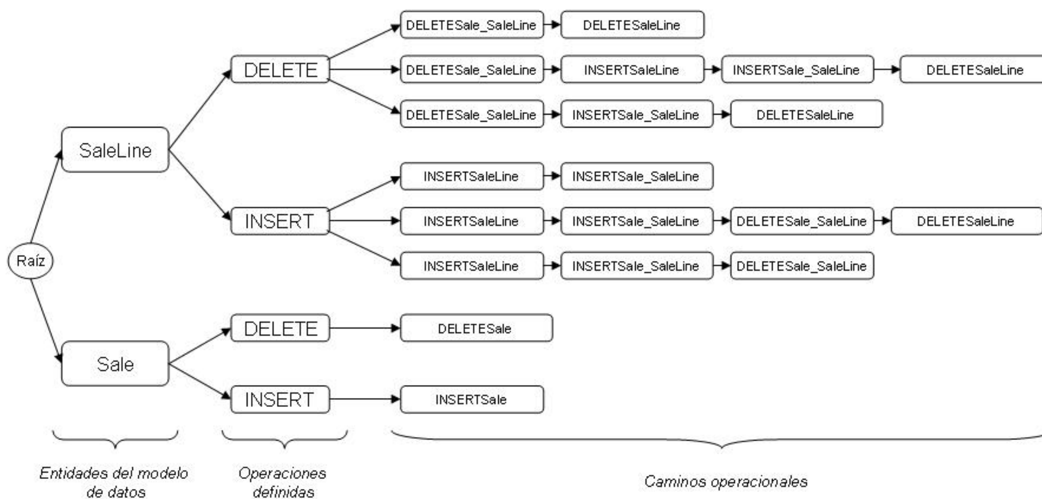


Fig. 3.2. Árbol de operaciones (ejemplo)

3.4.2. Consideraciones previas

El árbol de operaciones estará formado por el conjunto de todos los *caminos operacionales*. Estos caminos también podrán ser considerados correctos, ya que se satisfacen todas las dependencias de las operaciones que contiene, aunque no en los términos indicados en [2, section 3.3.2], entre otros motivos porque esta propiedad se entiende que sólo es de aplicación a *modelos navegacionales*.

La obtención de cada uno de los *caminos operacionales* y su adición al *árbol de operaciones* se realiza en un mismo proceso, siendo esta la razón que motiva la estructura de clases que se muestra en la Fig. 3.3, ya que cada una de ellas es responsable de tareas específicas.

Respecto a lo indicado en [2, definition 3.3.3] referente a que una operación *op* puede requerir $n > 1$ operaciones de tipo *op*, indicar que cada uno de los caminos operacionales del árbol sólo puede contener una operación *op* de un mismo tipo. La expansión de esta operación *op* en tantas como sean necesarias se realiza en el momento de crear el *grafo operacional*.

3.4.3. Proceso general de creación del árbol de operaciones

Tal y como se muestra en la figura Fig. 3.3, la clase *OperationsTree* se responsabiliza de coordinar ambos procesos (crear el *diccionario de dependencias* y obtener los *caminos operacionales*), delegando la tarea específica de crear el diccionario a la clase *TROperationsDictionary*.

Siguiendo los principios de encapsulación y minimización de dependencias entre paquetes, la estructura interna del árbol de operaciones fuera del paquete *tree* permanecerá oculta, pudiendo ser tratada por otras partes del programa mediante las correspondientes *Interfaces*.

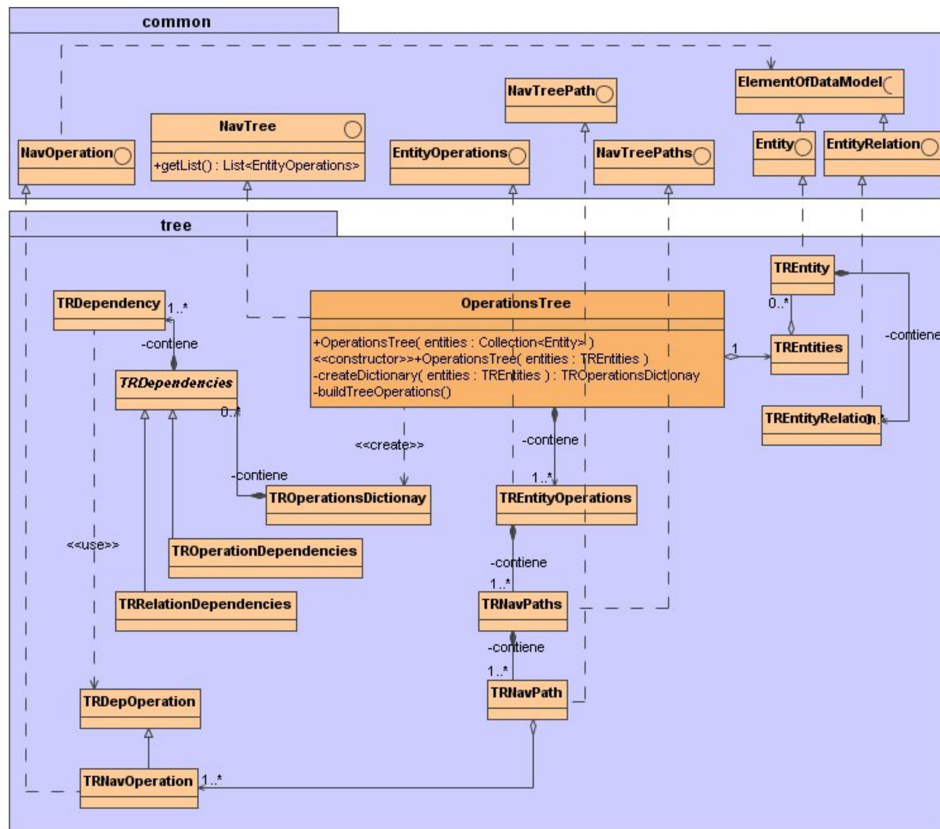


Fig. 3.3. Árbol de operaciones (diagrama general de clases)

Las funcionalidades más relevantes de la clase principal *OperationsTree* son las siguientes:

1. Creación de un objeto *OperationsTree* mediante cualquiera de los métodos constructores, siendo necesario sólo informar de las entidades que forman parte del modelo de datos.

Método	Paquete : **tree**	Clase : **OperationsTree**
+OperationsTree(Collection<Entity> entities)		
1) Se crea el objeto OperationsTree a partir de una colección de objetos Entity obtenidos del fichero XML que contiene el modelo de datos. Internamente son transformados en objetos TREntity, siendo gestionada la colección por un objeto TREntities. 2) Crea el diccionario de dependencias 3) Construye el árbol de operaciones		

2. Creación del diccionario de dependencias.

Método	Paquete : **tree**	Clase : **OperationsTree**
-createDictionary(TREntities entities):TROperationsDictionary		
Se crea el diccionario de dependencias a partir del objeto TREntities que contiene las entidades del modelo de datos.		

3. Construcción del árbol de operaciones.

Método	Paquete : **tree**	Clase : **OperationsTree**
-buildTreeOperations()		
Construye el árbol de operaciones a partir del diccionario de dependencias		

3.5. Grafo operacional

3.5.1. Introducción

En este apartado se evitará expresamente utilizar la expresión *grafo navegacional*, ya que, en realidad, la estructura de datos resultante de la Fase 2 (apartado 3.5.3) se asemeja mucho más a un árbol: no existen enlaces entre nodos o páginas de diferentes caminos y los nodos finales no disponen de información sobre el nodo que representa la página principal o la página de inicio de camino. Será al finalizar la Fase 3, resultado de la refactorización, cuando se hayan establecido parte de los enlaces entre nodos, dejando para la Fase 4 la creación del verdadero *grafo navegacional*.

3.5.2. Consideraciones previas

En el diseño del grafo operacional se ha tenido en consideración la representación de grafo mostrada en [2, section 3.1], aunque en este proyecto no se han considerado algunas de sus características:

1. Sólo podrá existir un arco entre dos nodos distintos.

2. Los arcos o sus etiquetas no guardan información de las operaciones asociadas al enlace, ya que no se requiere. Si no se tratase de esta manera, parte de los arcos guardarían información redundante.

   En este proyecto, todas las páginas que identifiquen a una misma operación, con independencia de la página de destino, siempre realizan las mismas acciones.

3. Los enlaces no se establecen entre páginas, sino entre los nodos del grafo. Estos nodos contienen información de la página, otros elementos relevantes y datos específicos de los sistemas que tratan el modelo de datos resultante (en este caso, WebRatio).

Respecto a los algoritmos de refactorización [2, section 4.3], el programa implementado dispone de modalidades[4] de funcionamiento que simplifican notablemente la construcción del grafo resultante y su estudio posterior. Estas cuestiones se discuten en el apartado 3.5.5.

3.5.3. Procedimiento para la obtención del *modelo navegacional* a partir de un *grafo operacional*

Han sido variados los pasos indicados [2, section 2.2] para la generación de *un modelo navegacional completo* y *correcto*, ya que complicarían en exceso el diseño e implementación del programa. Los pasos establecidos en este proyecto son las siguientes:

Paso 1. Obtener un *grafo operacional* a partir de los *caminos operacionales* que contiene el *árbol de operaciones*, refactorizando cada uno de estos caminos mediante el algoritmo *Head-Merge*.

Paso 2. Refactorización del grafo mediante el algoritmo *Tail-Merge*.

En la refactorización estándar[5] el algoritmo de *Head-Merge* no se aplica a todo grafo, sino a los *caminos operacionales* que tienen la misma *operación identificativa*.

[4] En función de la modalidad se obtendrá un grafo con determinadas características.

Paso 3. Añadir a los nodos información específica para la correcta visualización y representación del grafo en WebRatio.

Se considera esta última fase la apropiada para completar el grafo con toda aquella información adicional que pueda requerir el *modelo navegacional*, ya que ésta podría ser variable en función de las posibilidades de los sistemas que exploten el modelo resultante.

3.5.4. Diagrama de clases

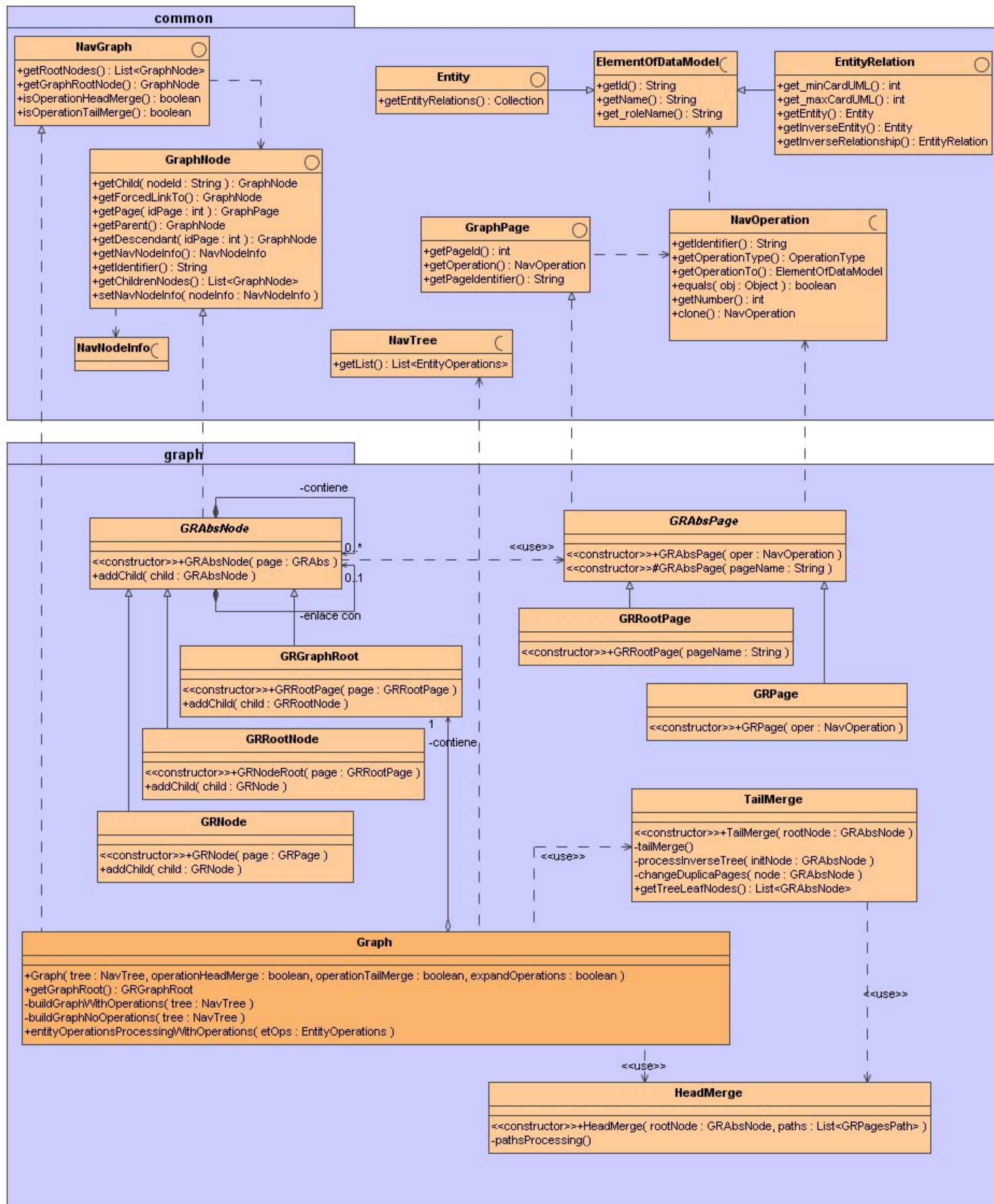


Fig. 3.4. Grafo operacional/navegacional (diagrama de clases)

[5] cuando el algoritmo Tail-Merge opera en *modalidad de grafo completo* y el algoritmo Head-Merge opera en *modalidad por operación* (apartado 3.5.5).

3.5.5. Refactorización

Los procesos implementados de refactorización tienen como misión unir aquellos caminos del grafo que tengan las mismas operaciones iniciales o finales. Si dos o más caminos tienen las mismas operaciones a lo largo de todo el recorrido, entonces todos ellos se unirán en uno.

En este proyecto se utilizan dos sistemas de refactorización, a los que llamaremos *Head-Merge* y *Tail-Merge*, pudiendo cada uno de ellos operar en dos modalidades: *por operación* o *grafo completo*.

- *Modalidad por operación*

  La refactorización se aplicará de forma individual a aquellos caminos del grafo que tengan la misma *operación identificativa*.

- *Modalidad de grafo completo*

  La refactorización se aplicará a todo el grafo, sin tener en consideración la *operación identificativa* de los distintos caminos.

3.5.5.1. Head-Merge

Este proceso de refactorización une aquella parte de camino que contiene las mismas operaciones iniciales. Si las operaciones son iguales en todo el recorrido, la unión será total.

La modalidad por defecto se establece *por operación*, ya que para este proyecto es importante conocer los caminos que genera cada una de sus *operaciones identificativas*. En la modalidad *de grafo completo*, al no tener en consideración estas últimas, puede suceder que se unan caminos con distintas *operaciones identificativas*, algo que dificultaría enormemente las tareas de verificación del *modelo navegacional* resultante. Se menciona esto ya que la modalidad *de grafo completo* es la que se propone en [2, section 4.3].

Uno de los aspectos a destacar del proceso *Head-Merge* implementado es la forma en el que se aplica. Normalmente, estos algoritmos suelen idearse para aplicarse sobre una estructura de datos ya formada, algo que, en este caso, habría exigido diseñar un algoritmo mucho más complejo, porque la manipulación de gran parte de los nodos que conforman el grafo no habría sido tarea trivial ni tampoco exenta de complicaciones. Se ha evitado la complejidad a la que se hacía referencia con una fórmula considerada interesante:

La refactorización *Head-Merge* tiene lugar en el mismo instante de añadir los nodos al grafo, ya que es muy sencillo determinar si una determinada operación ya se encuentra en el grafo y su posición; si no existe, se añade en la posición que le corresponde, en caso contrario, se prescinde de ella.

Los pasos para la refactorización *Head-Merge* serían los siguientes:

Paso 1. Asignación de un identificador a las operaciones que contienen los caminos.

En la Fig. 3.5 se observa que todas las operaciones son identificadas en un solo paso, aunque esto se muestra de esta manera para facilitar su comprensión. En el programa implementado la asignación de este identificador se realiza mucho antes.

Paso 2. Creación de un nuevo grafo con un único nodo, el nodo raíz.

Paso 3. Se añaden todas las operaciones en el grafo

En la Fig. 3.5 observamos que finalizado el paso 3.1 obtenemos un árbol con una operación existente en el mismo nivel –la número 3-. Este algoritmo no necesita para su correcto funcionamiento conocer las operaciones que no han sido añadidas al grafo, aunque como se verá, sí el algoritmo Tail-Merge.

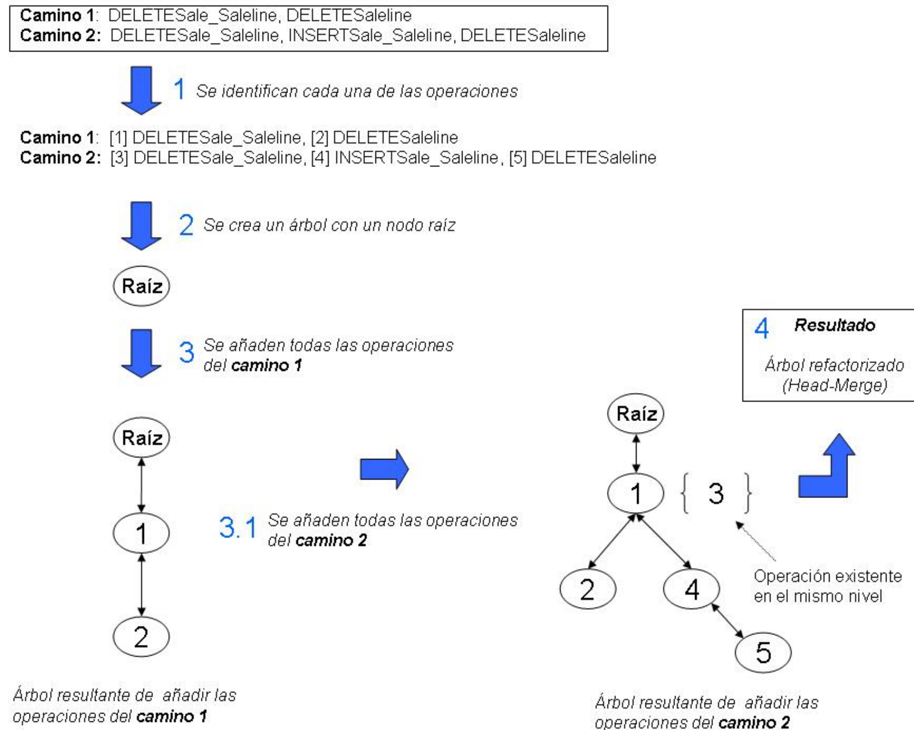


Fig. 3.5. Algoritmo Head-Merge

3.5.5.2. Tail-Merge

Este proceso de refactorización une aquella parte de camino que contiene las mismas operaciones finales. Si las operaciones son iguales en todo el recorrido, la unión será total, aunque con este programa no puede producirse esta situación, ya que los procesos de *Head-Merge* son lanzados en una fase previa y la detección de dos caminos iguales obliga a su unión.

La modalidad por defecto se establece *de grafo completo*, ya que reduce el espacio ocupado por el *modelo navegacional*.en fichero XML. Si la modalidad de *Head-Merge* es *de grafo completo*, los cambios en la modalidad de *Tail-Merge* no surtirán efecto, ya que siempre se asumirá la misma modalidad que la indicada para el *Head-Merge*.

A diferencia del *Head-Merge*, este proceso sí que se aplica una sola vez y a todo el grafo cuando la modalidad se establece *de grafo completo*. Esto puede parecer sumamente complejo, pero no lo es si se analiza con detenimiento el objetivo de este proceso de refactorización y se compara con el establecido para el *Head-Merge*. El resultado de este análisis se considera interesante, siendo éste el motivo por el que se introduce: el algoritmo *Tail-Merge* es similar al algoritmo *Head-Merge*, pero añadiendo varias funcionalidades adicionales; la principal, poder obtener las operaciones en orden inverso de cómo han sido añadidas al grafo.

Los pasos para la refactorización *Tail-Merge* del grafo existente (E) serían los siguientes:

Paso 1. Obtención de los caminos de (E) en orden inverso.

Se obtienen todos los nodos terminales de (E) y, a partir de estos, todos los nodos que representan los caminos del grafo, pero en orden inverso.

Paso 2. Creación de un nuevo grafo (N) con un único nodo, el nodo raíz.

Paso 3. Se añaden todas las operaciones en el grafo (N) en el mismo orden de recuperación.

Paso 4. Para cada uno de los nodos de (N) se determina si existen páginas duplicadas y, si procede, se establece en el grafo (E) un enlace con algún otro nodo.

En la Fig. 3.6 observamos que finalizado el paso 3.1 obtenemos un árbol con una operación existente en el mismo nivel –la número 5-. Esto le indica al programa que se puede establecer un enlace entre el nodo 2 y el 5, ya que las operaciones de ambos nodos son iguales.

En realidad este algoritmo resulta un poco más complicado, ya que la situación descrita anteriormente podría producirse en cualquier nivel del árbol, siendo necesario comprobar que realmente se puede efectuar un determinado enlace. La comprobación es sencilla: si en el árbol/grafo (E) todos los descendientes de los nodos involucrados no contienen exactamente las mismas operaciones, entonces no se podrán enlazar, ya que los caminos son distintos.

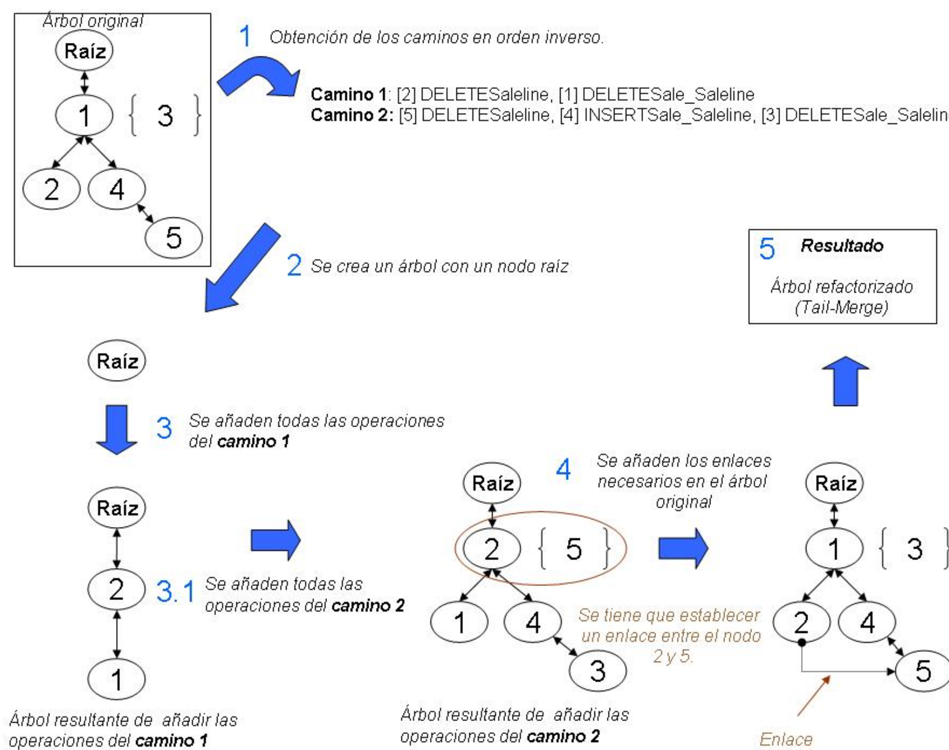


Fig. 3.6. Algoritmo Tail-Merge

3.5.6. Procedimiento para la obtención del grafo operacional

1. Creación de una nueva instancia de Graph.

<table>
<tr><td>Método</td><td>Paquete : graph</td><td>Clase : Graph</td></tr>
<tr><td colspan="3">+Graph(NavTree tree, boolean operationHeadMerge, boolean operationTailMerge, boolean expandedOperations)</td></tr>
<tr><td colspan="3">Parámetros</td></tr>
<tr><td colspan="3">• operationHeadMerge: si es false, el algoritmo de Head-Merge [2, pp.13] efectuará la refactorización de los caminos sin tener en cuenta las operaciones iniciales que los inician.
• operationTailMerge: si es false, el algoritmo de Tail.Merge [2, pp.13] efectuará la refactorización de los caminos sin tener en cuenta las operaciones iniciales que los inician
• expandedOperations: si es false, no serán expandidas las operaciones en las que así se indica.</td></tr>
<tr><td colspan="3">Notas</td></tr>
<tr><td colspan="3">Valores por defecto (modalidad estándar): operationHeadMerge = verdadero, operationTailMerge = falso, expandedOperations = verdadero.</td></tr>
<tr><td colspan="3">Funciones</td></tr>
<tr><td colspan="3">1) Creación de una nueva instancia de Graph, pasando como parámetros las modalidades de refactorización y de expansión de las operaciones.
2) Creación del nodo raíz GRGraphRoot del grafo navegacional, asignándole una página GRRootPage con el identificador ‘GRAPH’.
3) Ejecutar el método de creación adecuado en función del parámetro operationHeadMerge.
Se invocará el método buildGraphWithOperations si operationHedMerge es verdadero. Si es falso, se recurrirá al método buildGraphNoOperations.</td></tr>
</table>

2. Construcción del grafo a partir del árbol de operaciones

<table>
<tr><td>Método</td><td>Paquete : graph</td><td>Clase : Graph</td></tr>
<tr><td colspan="3">-buildGraphWithOperations(NavTree tree)</td></tr>
<tr><td colspan="3">Funciones</td></tr>
<tr><td colspan="3">1) Obtiene todos los objetos TREntityOperations y para cada uno de ellos invoca al método entityOperationsProcessingWithOperations.
2) Si operationTailMerge = false, efectúa una refactorización Tail-Merge del grafo.</td></tr>
</table>

3. Creación de los nodos del grafo y refactorización Head-Merge

<table>
<tr><td>Método</td><td>Paquete : graph</td><td>Clase : Graph</td></tr>
<tr><td colspan="3">-entityOperationsProcessingWithOPerations(EntityOperations etOps);</td></tr>
<tr><td colspan="3">Funciones</td></tr>
<tr><td colspan="3">1) Obtiene los objetos NavTreePaths de TREntityOperations, creando un nodo GRRootNode para cada uno de ellos. El identificador del nodo coincidirá con el identificador de la operación inicial. Estos nodos serán hijos del nodo raíz.
2) Obtención de los distintos caminos operacionales que contiene cada NavTreePaths y conversión de las operaciones contenidas a objetos GRPage. Estos objetos serán las páginas del grafo.
3) Adicción de las páginas al grafo efectuando en ese mismo momento una refactorización Head-Merge. Si operationTailMerge = true, efectúa una refactorización Tail-Merge de todos los caminos operacionales contenidos en NavTreePaths, es decir, todos aquellos que tienen la misma operación inicial.</td></tr>
</table>

3.5.7. Grafo navegacional

En este apartado se describen los pasos seguidos para que, a partir de un *grafo operacional*, pueda obtenerse un *grafo navegacional* y que resumiendo serían los siguientes:

Paso 1. Traducir las operaciones del grafo en operaciones WebML.

En la traducción también es necesario considerar aquellas situaciones en las que no se debe crear algún elemento, teniendo éste la consideración de especial. En este programa sólo se contempla un supuesto:

Si consideramos un *grafo operacional*, deberá tratarse de forma especial aquella página en la que -como resultado de la refactorización *Head_Merge-* se hayan establecido varios enlaces a otras páginas. Esto, desde la perspectiva de un árbol, sería equivalente a lo siguiente: se tratarán de forma especial aquellos nodos que como resultado de la refactorización Head_Merge tengan más de un nodo hijo.

En la Fig. 3.7 puede apreciarse que la página número 3 -cuya operación es *DELETESale_Saleline-* experimentará el tratamiento especial al que se hacía referencia. El elemento especial a crear –mostrado en la Fig. 3.8 será una página intermedia que permita seleccionar el camino a seguir, o dicho de otra manera, la siguiente operación a ejecutar entre varias alternativas.

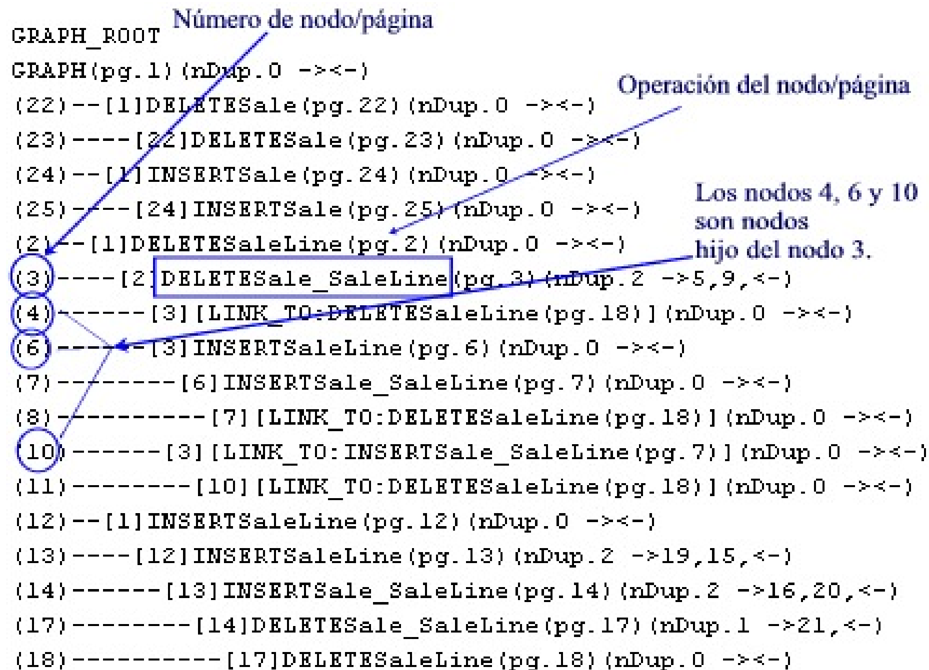


Fig. 3.7. Representación de un grafo operacional

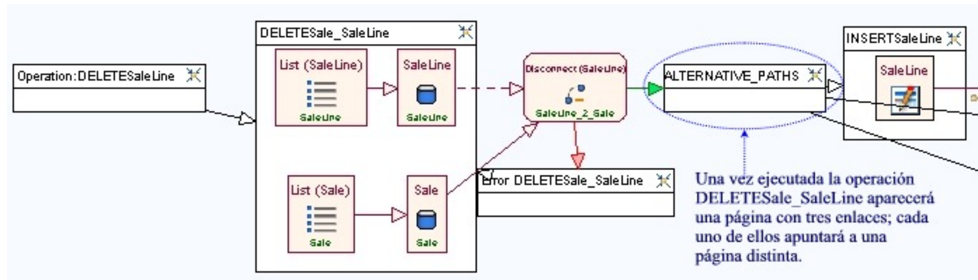


Fig. 3.8. Representación de un camino navegacional en WebRatio (vista parcial)

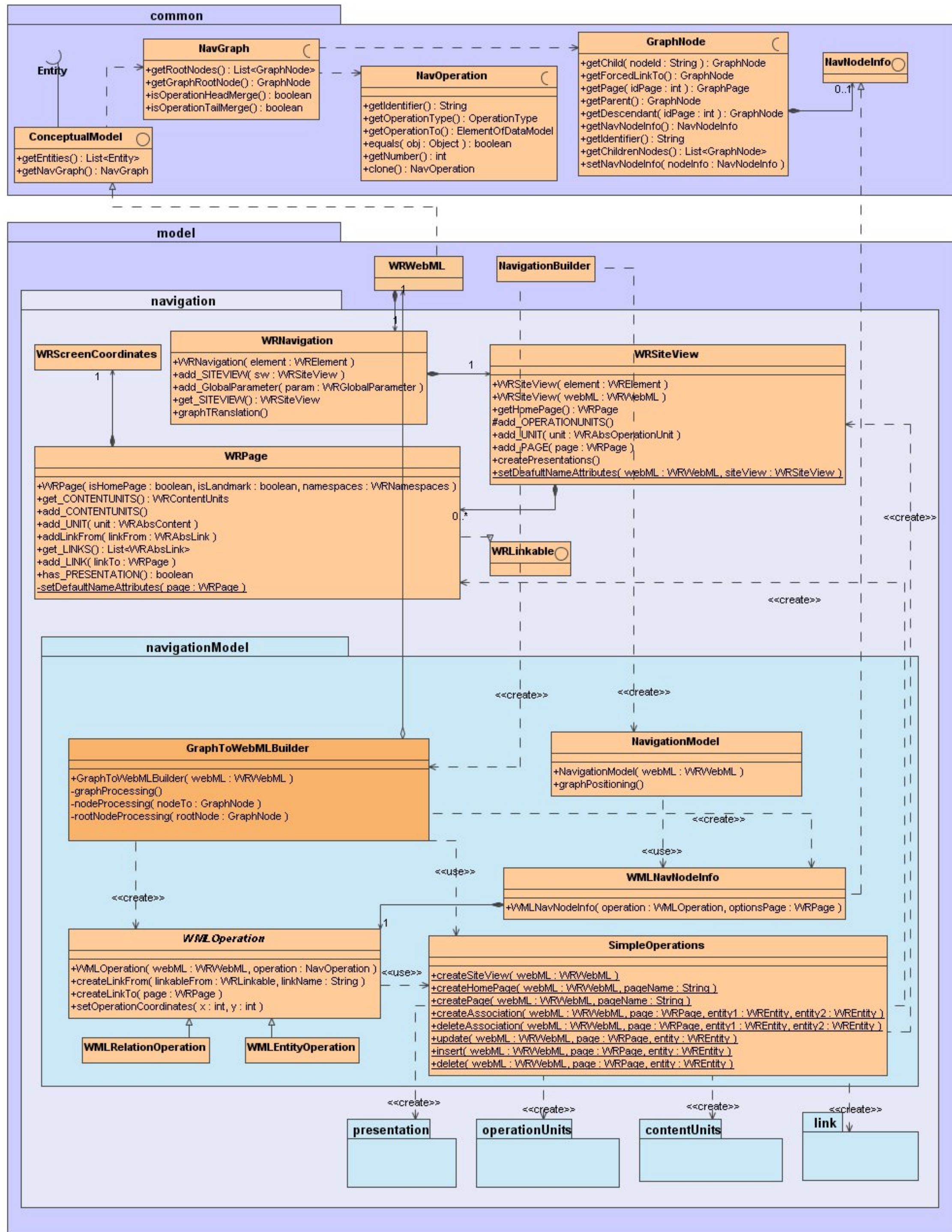
Paso 2. Traducir los enlaces del grafo en enlaces WebML.

Se debe tener en cuenta que los *modelos navegacionales* pueden requerir de otros tipos de enlaces no previstos en el grafo, como por ejemplo los enlaces a páginas de error. En este programa, aparte de lo comentado, también se establecen los enlaces entre los nodos finales y la página principal o la de inicio del camino, según sea el caso

Paso 3. Añadir información específica

En este caso, información de posicionamiento de los elementos WebML. Esto permite visualizar correctamente el *modelo navegacional* en WebRatio.

3.5.7.1. Diagrama de clases



3.5.7.2. Procedimiento para la obtención del grafo navegacional

1 Recurrir al método createNavigationModel de WebMLBuilder.

Método	Paquete : **model**	Clase : **WebMLBuilder**
+createNavigationModel(NavGraph navGraph)		
Parámetros		
• navGraph: en este caso un objeto uoc.graph.Graph, ya que esta clase implementa uoc.common.NavGraph		
Funciones		
1. Asignar al objeto WRWebML la referencia del objeto navGraph. 2. Invocar al método graphTranslation de NavigationBuilder.		

2 Recurrir al método graphTranslation de WRNavigationBuilder

Método	Paquete : **model**	Clase : **NavigationBuilder**
+graphTranslation()		
Parámetros		
Funciones		
1. A partir del objeto WRWebML recuperar la referencia al objeto navGraph. 2. Crear una instancia de GraphToWebMLBuilder.		

3 Creación de una nueva instancia de GraphToWebMLBuilder

Método constructor	Paquete : **model**	Clase : **GraphToWebMLBuilder**
+GraphToWebMLBuilder(WRWebML webML)		
Parámetros		
• webML: nodo raíz del modelo navegacional.		
Funciones		
1) Procesar el grafo para añadir la información necesaria. Se invoca al método graphProcessing.		

4 Procesar el grafo para añadir la información necesaria.

Método	Paquete : **model**	Clase : **GraphToWebMLBuilder**
+graphProcessing()		
Parámetros		
Funciones		
1) Crear el SiteView (elemento de WebML) y añadirlo al modelo navegacional 2) Crear la página principal, añadirla al modelo navegacional y asignar una referencia de la página al nodo raíz del grafo. 3) Procesar, uno a uno, los nodos hijos del nodo raíz invocando al método rootNodeProcessing 4) Crear los elementos WebML de presentación; se invoca al método createPresentations del objeto WRSiteView.		

5 Procesar, uno a uno, los nodos hijos del nodo raíz del grafo

Método	Paquete : **model**	Clase : **GraphToWebMLBuilder**
+rootNodeProcessing(GraphNode node)		
Parámetros		
Funciones		
1) Crear la página de inicio de operación, añadirla al modelo navegacional y asignar una referencia de la página al nodo 2) Procesar, uno a uno, los nodos hijos del nodo actual invocando al método nodeProcessing		

6 Procesar, uno a uno, los nodos hijos del nodo actual (proceso recursivo)

Método	Paquete : **model**	Clase : **GraphToWebMLBuilder**
+nodeProcessing(GraphNode nodeTo)		
Parámetros		
• nodeTo: nodo actual y del que se tiene que crear la operación WebML		
Funciones		
1) Crear la operación WebML, añadirla al modelo navegacional y asignar una referencia de la operación al nodo actual		

3.6. Modelo navegacional

3.6.1. Diagrama de clases

En este diagrama se muestran la mayoría de las clases y/o interfaces que intervienen en la obtención del *modelo navegacional*, indicando sólo aquellos métodos considerados importantes para la comprensión de su estructura interna.

Se puede apreciar que el paquete *navigation* contiene a otros no mencionados hasta este momento: *contentUnits, operationUnits, presentation, link* y *navigationModel*. Todos ellos son necesarios para la obtención del *modelo navegacional* y, a excepción de *navigationModel*, las instancias de sus clases formarán parte de la voluminosa y compleja estructura arbórea que representa el *modelo navegacional*.

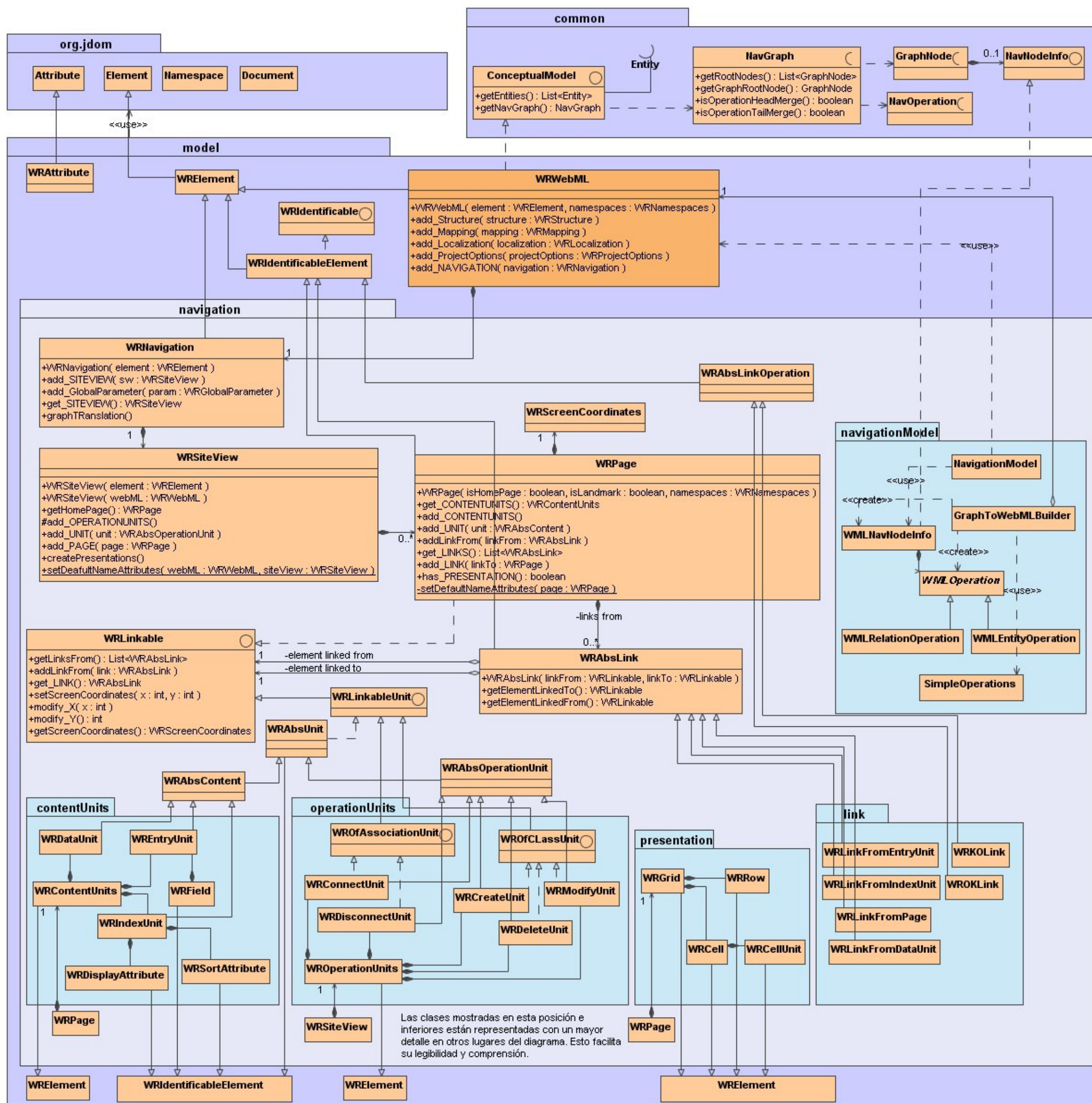


Fig. 3.9. Modelo navegacional (diagrama de clases)

3.6.2. Modelado de los elementos navegacionales

WebRatio dispone de una serie de elementos que permiten la representación de un *modelo navegacional* y que, resumiendo, serían los siguientes: páginas, enlaces (=links), contenido de las páginas (*ContentUnits* [10]) y operaciones *(OperationUnits* [10]). En este proyecto se ha pretendido modelar adecuadamente todos estos elementos, siendo éste el motivo del diseño de los paquetes mostrados (apartado 3.6.1). El contenido y función de cada uno de ellos es el siguiente:

- Paquete *contentUnits*

  Las clases que se muestran modelan los distintos elementos de los que pueden estar formados las páginas [10]: *DataUnits, EntryUnits, IndexUnits*, requiriendo alguno de ellos elementos adicionales (ej: un objeto WREntryUnit está formado por objetos WRField, manejando estos últimos la información específica de los campos mostrados en la ventana)

- Paquete *operationUnits*

  Contiene aquellas clases que modelan las operaciones que se pueden realizar sobre las entidades y relaciones definidas en el modelo de datos [10]: *ConnectUnit, DisconnectUnit, CreateUnit, DeleteUnit, ModifyUnit*. Las dos primeras operan con relaciones, las restantes con entidades.

- Paquete *link*

  Cada una de sus clases modela un enlace entre los distintos 'Units' (*OperationUnits* o *ContentUnits*) o entre páginas del *modelo navegacional*.

- Paquete *presentation*

  Las páginas en WebML se componen de filas (row) y cada una de éstas por celdas (cell). El conjunto de filas y celdas forma la cuadrícula (grid). Este sistema permite asignar la posición que ocuparán los diversos *ContentUnit* y otros elementos de visualización en la página Web resultante.

- Paquete *navigationModel*

  Contiene las clases necesarias para transformar un *grafo operacional* en *navegacional* y crear el *modelo navegacional* correspondiente.

3.6.3. Procedimiento para la obtención del modelo navegacional

1 Recurrir al método createNavigationModel de WebMLBuilder.

Método	Paquete : **model**	Clase : **WebMLBuilder**
+createNavigationModel()		
Funciones		
1. Comprobar que el objeto WRWebML contiene una referencia válida al objeto navGraph. Esta referencia fue registrada en el momento de crear el grafo navegacional 2. Recurrir al método createNavigationModel de NavigationBuilder.		

2 Recurrir al método createNavigationModel de WRNavigationBuilder

Método	Paquete : **model**	Clase : **NavigationBuilder**
+createNavigationModel()		
Funciones		
1. A partir del objeto WRWebML recuperar la referencia al objeto navGraph. 2. Crear una instancia de NavigationModel.		

3 Creación de una nueva instancia de NavigationModel

Método constructor	Paquete : **Model**	Clase : **NavigationModel**
+NavigationModel(WRWebML webML)		
Parámetros		
• webML: nodo raíz del modelo navegacional.		
Funciones		
1) Procesar el grafo para añadir la información necesaria. Se recurre al método graphPositioning.		

4 Procesar el grafo para añadir la información necesaria.

Método	Paquete : **model**	Clase : **NavigationModel**
+graphPositioning()		
Funciones		
1) Crear los enlaces entre operaciones WebML y asignar coordenadas de posición; se recurre al método graphPositioning		

4. Detalles de diseño

4.1. Introducción

En este capítulo se exponen aquellos detalles de diseño considerados relevantes y que permiten al lector profundizar un poco más en el diseño del programa, presentando estos de forma ascendente, es decir, de de menor a mayor granulidad, empezando por el diccionario de dependencias y acabando con el tratamiento del modelo navegacional.

4.2. Diccionario de dependencias

El *diccionario de dependencias* está formado por todas aquellas operaciones básicas necesarias para modificar el estado de la información gestionada por la aplicación y sus dependencias., tal y como se muestra la Fig. 4.1.

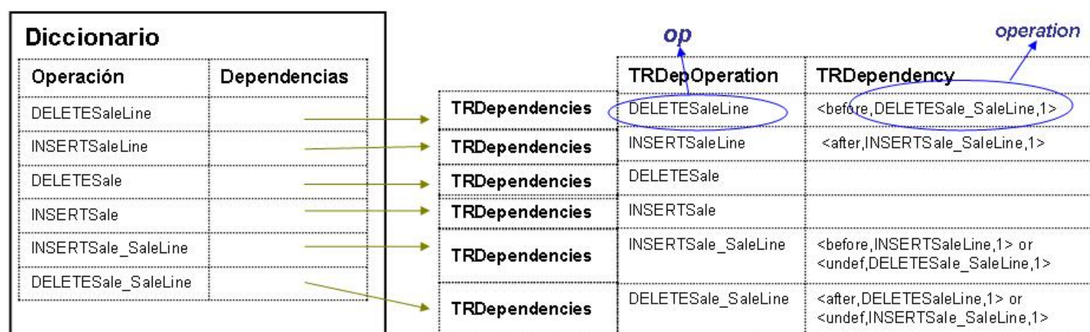


Fig. 4.1. Estructura interna del diccionario (ejemplo)

El conjunto de las entradas del diccionario estará formado por todas aquellas operaciones *op* definidas sobre alguna entidad (*InsertET, DeleteET*) [6] o relación (*InsertRT y DeleteRT*) del modelo de datos.

Las dependencias de cada una de las operaciones son calculadas por la propia aplicación mediante el algoritmo propuesto por sus autores [2, definition 3.3.3, 3.3.4 y 3.3.5].

4.2.1. Consideraciones previas

En la obtención del diccionario se han seguido las definiciones dadas [2], siendo la estructura la que se muestra en la Fig. 4.1. Como se puede comprobar, cada una de las entradas del diccionario tienen asociadas un objeto *TRDependencies*, pudiendo éstos ser del tipo *TROperationDependencies* (Dep_{OP}, según [2, definition 3.3.5]) o *TRRelationDependencies* (Dep_{RT}, según [2, definition 3.3.4]). Estas dependencias contienen uno o varios objetos *TRDependency* con una estructura similar a la indicada en [2, definition 3.3.3].

Respecto a esta última definición mencionar que se ha modificado la especificación original. El cambio ha consistido en sustituir el símbolo ← por → en el supuesto de que *op = DeleteRT*. Con este cambio se consigue que no se ejecute una operación *DeleteET* antes de eliminar la relación correspondiente.

[6] La operación Update no se ha considerado por no tener interés para los objetivos de este proyecto

4.2.2. Diagrama detallado de clases

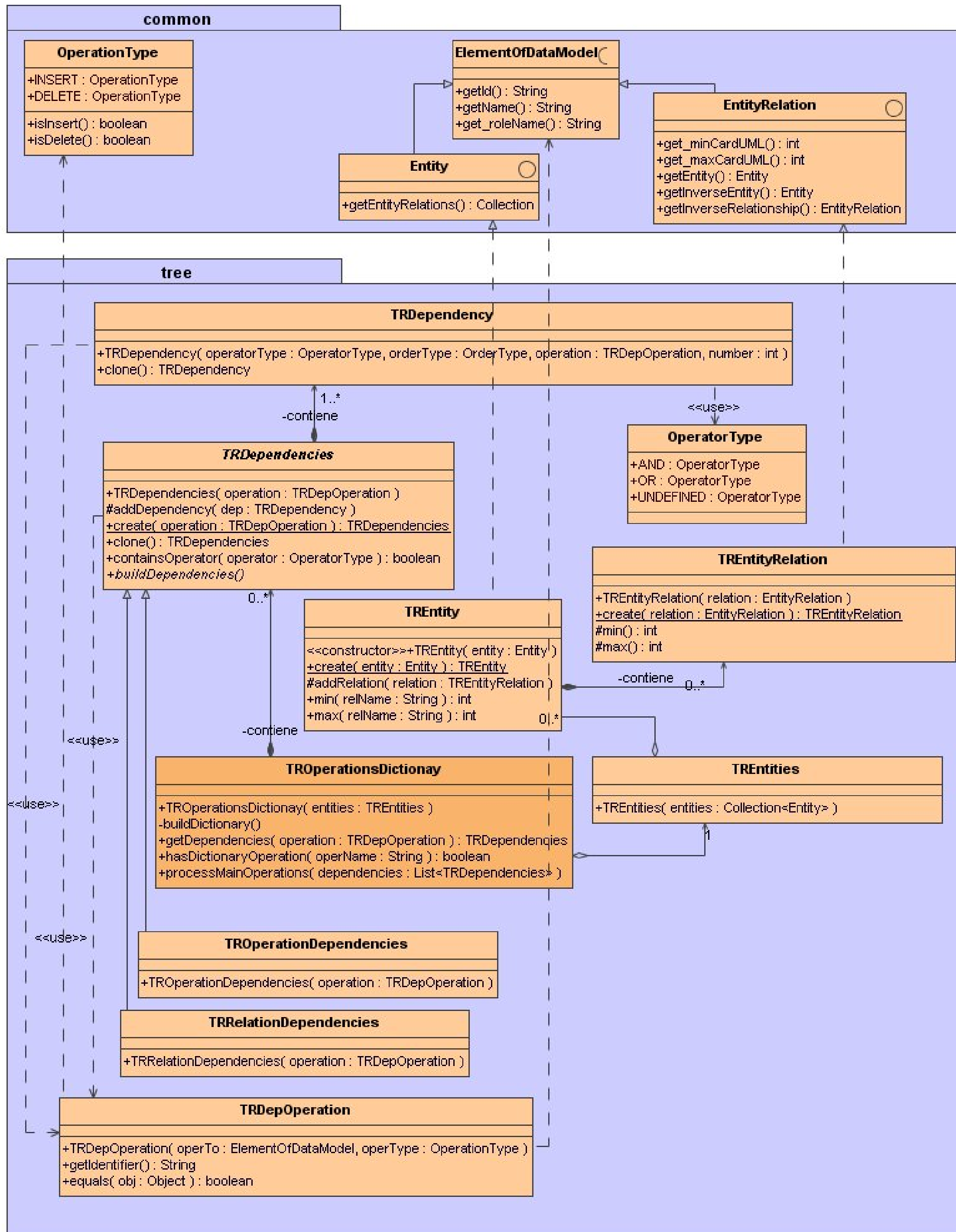


Fig. 4.2. Obtención del diccionario (diagrama detallado de clases)

4.2.3. Procedimiento para la obtención del diccionario de dependencias

1. Creación de una nueva instancia de TROperationsDictionary.

Método constructor	Paquete : **tree**	Clase : **TROperationsDictionary**
+TROperationsDictionary(TREntities entities)		
Funciones		
1) Creación de una nueva instancia de TROperationsDictionary pasando como único parámetro un objeto TREntities. 2) Construcción del diccionario de dependencias mediante el método buildDictionary		

2. Construcción completa del diccionario.

Método	Paquete : **tree**	Clase : **TROperationsDictionary**
+buildDictionary()		
Funciones		
1) Crea tantos objetos TRDepOperation como operaciones 'op' establecidas (Insert, Delete) para cada una de las entidades. A partir de estas operaciones son obtenidas sus dependencias TRDependencies. Estas dependencias – que incluyen la operación que las ha generado- se guardan en una lista para ser posteriormente procesadas por el método processMainOperations.		

3. Procesamiento de las dependencias

Método	Paquete : **tree**	Clase : **TROperationsDictionary**
+processMainOperations(List<TRDependencies> dependencies)		
Funciones		
1) De cada una de las dependencias de la lista *dependencies* se obtiene la operación *TRDepOperation* que las ha generado, comprobando si esta operación '*op'* existe en el diccionario. Si existe, se obtienen las siguientes dependencias hasta que no quede ninguna por procesar. 2) Si no existe y tiene dependendencias, se crea la entrada en el diccionario. Posteriormente se obtiene la operación '*operation'* de cada uno de los objetos *TRDependency* y se verifica su existencia en el diccionario. Si no existe, se obtienen sus dependencias y se añade en la lista *dependencies* para procesarse posteriormente.		

En la Fig. 4.3 se muestra un diagrama con el algoritmo implementado.

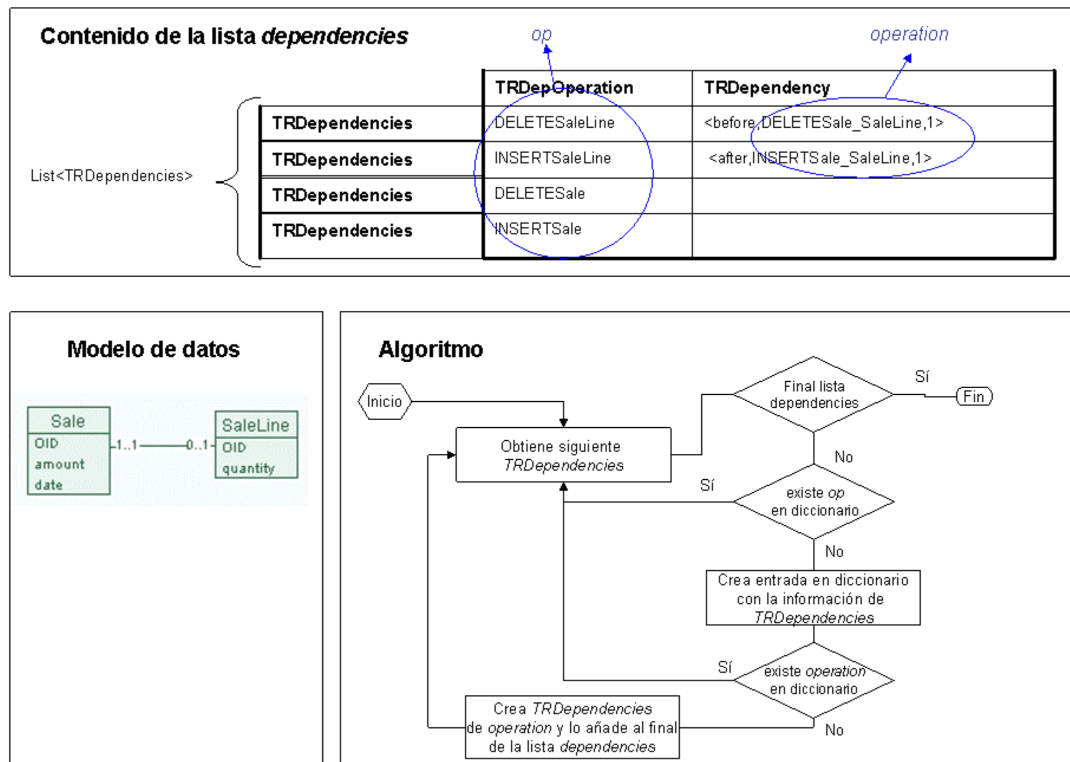


Fig. 4.3. Algoritmo de procesamiento de las dependencias

4.3. Caminos operacionales

Los *caminos operacionales* están formados exclusivamente por secuencias de operaciones tal y como se muestra en la Fig. 4.4. Estas secuencias se obtienen una vez procesadas las operaciones y las *dependencias* que contiene el *diccionario*.

Una secuencia de operaciones no podrá ser considerada un *camino navegacional* si no procede de un *grafo navegacional*. Esto se plantea así ya que un *camino navegacional* se entiende está formado por páginas (páginas de operaciones) y enlaces entre éstas (links). Un *camino operacional*, sin embargo, sólo contiene una secuencia de operaciones, sin ningún otro elemento adicional.

op

Operación	Camino operacional
DELETESaleLine	DELETESale_SaleLine, INSERTSaleLine, INSERTSale_SaleLine, DELETESaleLine
INSERTSaleLine	INSERTSaleLine, INSERTSale_SaleLine, DELETESale_SaleLine, DELETESaleLine
DELETESale	DELETESale
INSERTSale	INSERTSale

Fig. 4.4. Camino operacional (ejemplo)

Todo camino operacional tendrá como identificador de camino una operación de alto nivel *op* que opera (=realiza una acción) exclusivamente sobre una entidad específica del modelo de datos. Las operaciones sobre relaciones pueden formar parte del camino, pero <u>nunca</u> identificarán al mismo, ya que no se contempla esta posibilidad.

4.4. Árbol de operaciones

El árbol de operaciones, como ya se ha indicado en otros apartados, se obtiene una vez se dispone de los *caminos operacionales*

4.4.1. Diagrama de clases

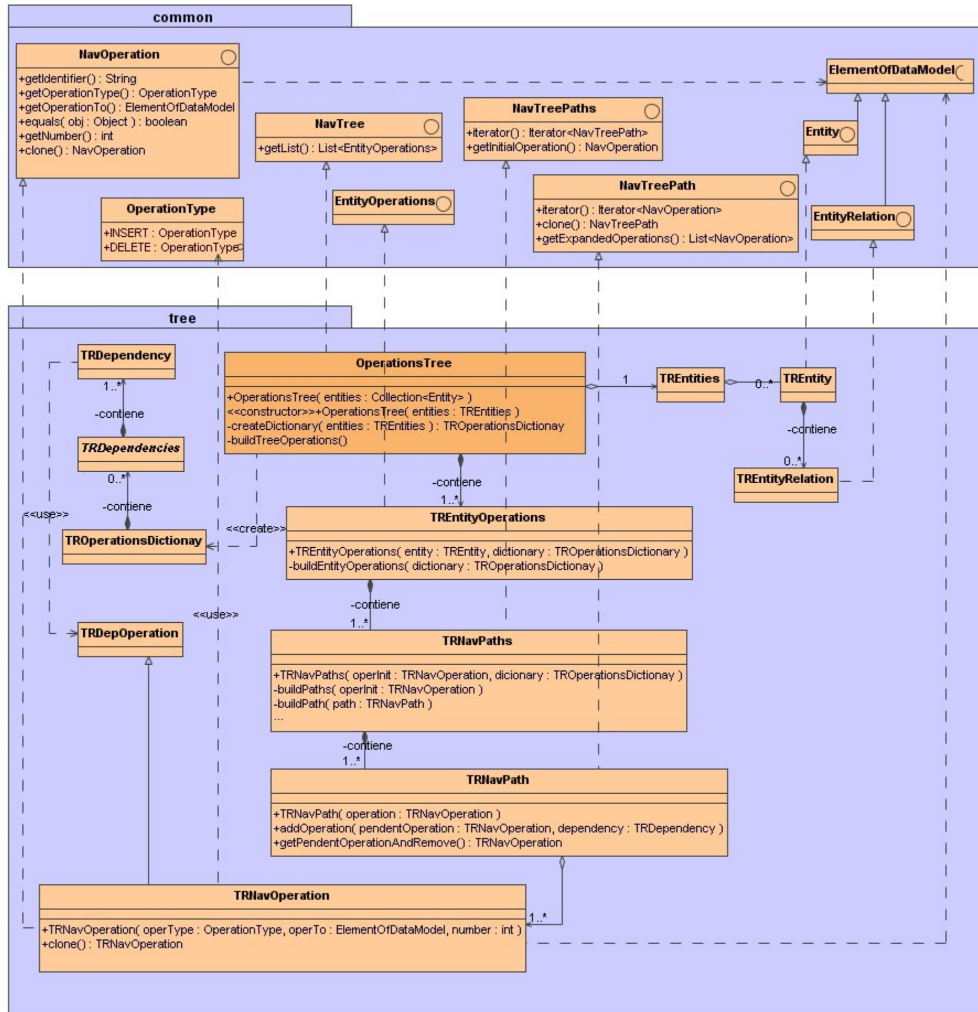


Fig. 4.5. Árbol de operaciones (diagrama detallado de clases)

4.4.1.1. Procedimiento para la obtención del árbol de operaciones

La clase *OperationsTree*, que representa el nodo raíz de este árbol, crea una instancia de *TREntityOperations* para cada una de las entidades del modelo de datos, ocupando estos objetos el siguiente nivel en el árbol de operaciones.

1. Creación de una nueva instancia de TREntityOperations.

Método constructor	Paquete : **tree**	Clase : **TREntityOperations**
+TREntityOperations(TREntity entity, TROperationsDictionary, dictionary)		
Funciones		
1) Creación de una nueva instancia de TREntityOperations pasando como parámetros el diccionario y la entidad de la que se desean obtener los caminos operacionales. 2) Construcción de las operaciones iniciales de la entidad mediante el método buildEntityOperations.		

2. Construcción de las operaciones iniciales de la entidad.

Método	Paquete : **tree**	Clase : **TREntityOperations**
+buildEntityOperations(TROperationsDictionary dictionary)		
Funciones		
1) Crea las operaciones iniciales de la entidad (Insert y Delete) y que serán del tipo TRNavOperation. 2) Para cada una de estas operaciones iniciales crea una instancia de TRNavPaths, conteniendo ésta los diversos caminos de la misma operación inicial. Estos objetos TRNavPaths ocuparán el siguiente nivel en el árbol, siendo hijos de TREntityOperations.		

3. Construcción de los caminos de una misma operación inicial

Método	Paquete : **tree**	Clase : **TRNavPaths**
+buildPaths(TRNavOperationList operInit)		
Funciones		
1) Crea un camino operacional que consta de una sola operación, la operación inicial. 2) Se completa cada uno de los caminos existentes –incluidos los añadidos posteriormente- invocando al método buildPath.		

4. Construcción del camino operacional

Método	Paquete : **tree**	Clase : **TRNavPaths**
+buildPath(TRNavPath path)		
Funciones		
1) Se comprueba que todas las dependencias de las operaciones que contiene el camino operacional sean satisfechas. Si no fuese así, se añadirían las operaciones correspondientes (mediante invocación del método addOperation de TRNavPath) y/o se crearían los caminos adicionales necesarios.		

5. Adición de operaciones al camino operacional

Método	Paquete : **tree**	Clase : **TRNavPath**
+addOperation(TRNavOperation pendentOperation, TRDependency dependency)		
Funciones		
1) Se procesa la dependencia *dependency* y se modifica el camino en función de lo que ésta indique.		

4.5. Tratamiento de los modelos definidos en WebRatio

4.5.1. Consideraciones previas

Se puede apreciar una gran coincidencia entre los nombres de las clases y las relaciones establecidas con los que aparecen en el fichero XML, ya que uno de los objetivos de diseño ha consistido en emular al máximo la sintaxis y los elementos manejados por WebRatio, algo que se considera fundamental para conseguir la flexibilidad deseada, aunque éste no ha sido el único motivo.

En la lectura del fichero XML de entrada se ha utilizado la librería o paquete JDOM (Java Document Object Model), permitiendo cargar en objetos específicos los elementos que forman la estructura del fichero. Esto se considera como una importante ventaja, ya que sus clases tienen implementadas gran parte de las funcionalidades necesarias para manejar estructuras arbóreas y su posterior representación en diferentes formatos, entre ellos XML. Solucionada la carga inicial, el tratamiento de los elementos y la creación del fichero de salida en formato XML, sólo quedaban por resolver tres cuestiones importantes: poder diferenciar claramente cada uno de los elementos, la necesidad de funcionalidades específicas y el establecimiento de vínculos entre los distintos elementos del árbol. La solución a todo ello pasaba por crear clases envolventes[7] de algunas clases existentes en el paquete *org.jdom*: *Element y Attribute*.

El diseño de estas clases se muestra en la Fig. 4.6, pudiendose apreciar en este diagrama que las clases que representan elementos de WebRatio (*WRWebML, WRStructure, WREntity, WREntityRelation, WREntityAttribute*) heredan las propiedades de *WRIdentificableElement* o *WRElement* y éstas, a su vez, heredan las propiedades de la clase *org.jdom.Element*. Esto sería extensible a todas las clases que representan elementos de WebRatio, tal y como se muestra en la figura Fig. 3.9.

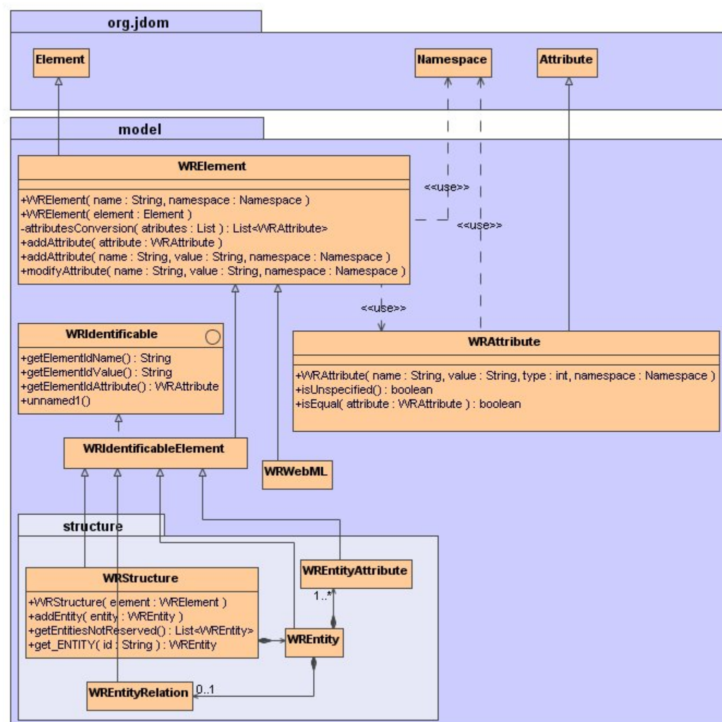


Fig. 4.6. Clases envolventes (diagrama)

[7] Clase que actúa como interface de la clase a la que envuelve, facilitando la incorporación de nuevas funcionalidades o la redefinición de las ya existentes.

4.5.2. Diagrama general detallado

En la Fig. 4.7 se muestran las relaciones entre paquetes y clases principales en el tratamiento de los distintos modelos requeridos en este proyecto. Las clases que permiten obtener el *modelo navegacional* no están representadas, ya que su estudio no es objetivo de este apartado.

Dentro del paquete *model* se aprecian cinco paquetes principales: *structure, navigation, mapping, projectOption y localization*, coincidiendo cada uno de ellos con los elementos estructurales de los ficheros tratados por WebRatio aunque sólo los dos primeros contienen información específica del modelado de las capas que tradicionalmente conforman un aplicativo Web.

El programa en su conjunto hace un uso intensivo de la información gestionada por *structure*, ya que este paquete contiene las clases que representan los elementos imprescindibles de todo *modelo de datos*; una vez leído el fichero XML de entrada y creadas las estructuras necesarias ya no se modifica su contenido posteriormente. Los restantes paquetes, a excepción de *navigation*, no aportan ninguna funcionalidad especial al programa, siendo sólo necesarios en el momento de procesar ficheros XML, tanto de entrada como de salida.

El paquete *navigation* contiene las clases necesarias para crear el *modelo navegacional*, haciéndose uso de ellas una vez obtenido el grafo.

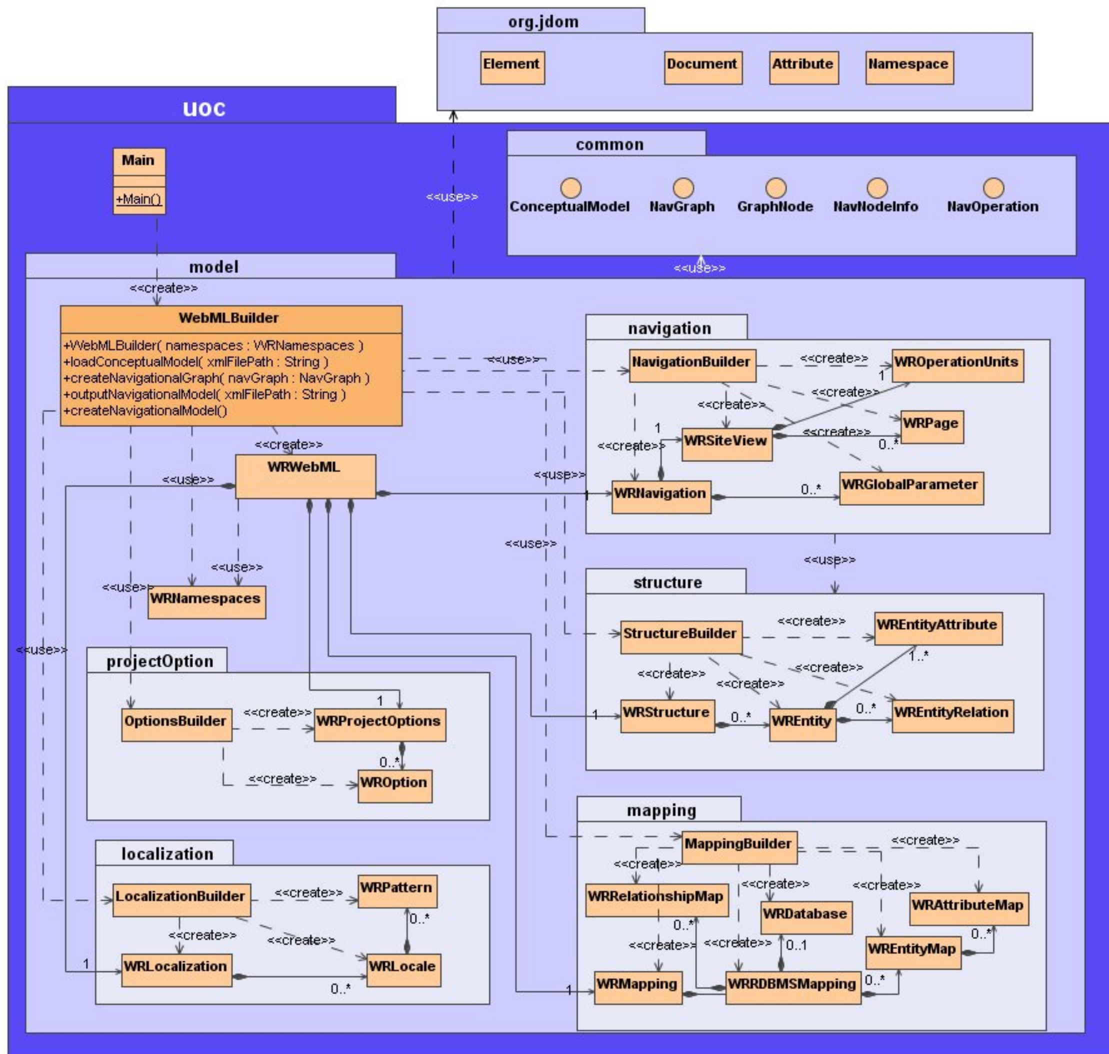


Fig. 4.7. Tratamiento de los modelos definidos en WebRatio (diagrama detallado de clases)

4.5.3. Funcionalidades generales de la clase principal *WebMLBuilder*

Las funcionalidades previstas son las siguientes:

- Carga del fichero XML de entrada (*SDSProject.xml*).

  Proporciona al programa toda la información necesaria sobre los modelos definidos en WebRatio y otros elementos importantes.

- Creación del grafo navegacional.
- Creación del modelo navegacional.
- Obtención del fichero XML de salida con los modelos previstos y otros elementos importantes para la generación automática del código de la aplicación Web.

4.5.3.1. Proceso para la carga inicial del fichero XML de entrada

El proceso se inicia ejecutando la clase *Main*, sin parámetros, lo que instruye al aplicativo a cargar el fichero XML de entrada por defecto. A partir de este momento el programa realiza lo siguiente:

1. Creación de una nueva instancia de WebMLBuilder.

Método	Paquete : **model**	Clase : **WebMLBuilder**
+WebMLBuilder(WRNamespaces namespaces)		
Parámetros		
• namespaces: los nombres de espacio usados en el fichero		
Funciones		
1. Cargar el fichero XML de entrada mediante el método loadConceptualModel. 2. Creación del grafo de navegación mediante el método createNavigationalGraph. 3. Creación del modelo navegacional mediante el método createNavigationalModel. 4. Crear el fichero de salida en formato XML mediante el método outputNavigationalModel.		

2. Carga el fichero XML de entrada.

Método	Paquete : **model**	Clase : **WebMLBuilder**
+loadConceptualModel(String xmlFilePath)		
Parámetros		
• xmlFilePath: ruta completa al fichero XML		
Funciones		
1. Crea el nodo raíz del documento XML que se conservará en memoria. Este será un objeto del tipo WRWebML 2. Crea una instancia de las clases responsables de construir y/o procesar los elementos estructurales obtenidos del fichero XML de entrada: NavigationBuilder, StructureBuilder, MappingBuilder, LocalizationBuilder, OptionsBuilder. 3. Lee todos los elementos que contiene el fichero de entrada, delegando su creación y su posicionamiento en el árbol XML a los objetos constructores del punto anterior.		

3. El resto de tareas -creación del grafo y modelo navegacional- se detallan en los apartados 3.5.7.2 y 3.6.3 respectivamente.

4.5.3.2. Proceso para la obtención del fichero XML a procesar por WebRatio

Para poder hacer uso de esta funcionalidad es imprescindible que el programa haya realizado la carga del fichero XML y la creación del modelo navegacional. Esto implicará la existencia de una instancia de *WebMLBuilder*, así como que se dispone de un objeto *NavGraph* que contiene el grafo navegacional.

1. Crear el fichero XML de salida.

Método	Clase **WebMLBuilder**
+outputNavigationModel(String xmlFilePath)	
Parámetros	
• xmlFilePath: ruta completa del fichero XML	
Funciones	
1. Utilizando las funcionalidades del paquete *org.jdom* se crea un documento XML con las siguientes opciones de formato: • Sangrado equivalente a dos espacios • Expansión de los elementos vacíos: <tagName/> to <tagName></tagName>. 2. Creación del fichero de salida haciendo uso de la clase *org.jdom.XMLOutputter*	

El código fuente de esta funcionalidad (Fig. 4.8) muestra la simplicidad y el poco esfuerzo que supone obtener el fichero XML de salida. Ello es consecuencia de las acertadas decisiones de diseño ya comentadas anteriormente.

```java
public void outputNavigationalModel(String xmlFilePath){
    String url =  xmlFilePath;
    Document root = this.getWebMLDocument();
    Format formato = Format.getPrettyFormat();
    //Coloca etiqueta final si no tiene contenido
    formato = formato.setExpandEmptyElements(true);
    XMLOutputter fmt = new XMLOutputter(formato);
    FileOutputStream fs = null;
    try {
        fs = new FileOutputStream(url);
        fmt.output(root,fs);
    } catch (IOException e) {
        System.out.println("Error creando fichero XML");
    }
}
```

Fig. 4.8. Código Java del método *outputNavigationModel*

5. Conclusiones

La dedicación a este proyecto ha sido máxima y aunque siempre quedan cuestiones por resolver o mejorar, creo que los objetivos se han cumplido ampliamente, siendo mi deseo que este trabajo facilite a mis colegas J. Ceballos y J.Cabot sus tareas de investigación o, al menos, permita a otros investigadores avanzar en la línea abierta.

Este proyecto no se ha considerado en ningún momento como un PFC tradicional, siendo éste en sí mismo uno de los motivos del esfuerzo y del tiempo empleados, no sólo en el diseño e implementación, sino también en la redacción de la memoria y de los productos complementarios. Considero que el conocimiento adquirido durante todo este tiempo es necesario se transmita de alguna forma. Creo que esta memoria es el medio adecuado, aunque para ello haya resultado necesario estructurar los contenidos de una forma un tanto especial.

El producto en el que se centra este proyecto no es el aplicativo Web generado por WebRatio, sino el programa que construye el modelo navegacional para que ello sea posible, y aunque considere que la obtención de este modelo navegacional en un fichero XML tampoco sea lo esencial. En mi opinión, lo más importante –y complejo- es todo aquello que se encuentra en las diversas capas inferiores del *modelo navegacional* y que permite que se obtenga dicho modelo: *diccionario de dependencias*, *árbol de operaciones*, *grafo operacional*, *refactorización y grafo navegacional*; el resto, se generan en mayor o menor medida de forma automática.

Todo este proyecto se fundamenta en la teoría expuesta en [2, definition 3.3.4]: cálculo de las dependencias. En la mayoría de los casos se obtienen *caminos navegacionales* completos y correctos, pero no en todos los casos.

Existe también alguna cuestión que podría resultar interesante estudiar posteriomente, como por ejemplo:

En [2] se establecen dos operaciones sobre relaciones: *InsertRT* y *DeleteRT*, entendiendo que ello obedece a cuestiones de uniformidad y simplicidad. En mi opinión, si se considera importante crear *modelos navegacionales* lo más parecidos posible con la realidad de los diseñadores de bases de datos y programadores, podría ser conveniente introducir una nueva operación: *UpdateRT*. Esto abriría nuevas posibilidades a la investigación y permitiría crear modelos más naturales y, posiblemente, mejor comprendidos. Al igual que se definen tres operaciones posibles para una entidad: *InsertET*, *DeleteET* y *UpdateET*, deberían contemplarse estas mismas operaciones para las relaciones, ya que la operación *UpdateRT* tiene un objetivo claro: la actualización de relaciones. Por norma general, se hace uso de una operación *DeleteRT* cuando se borra una o ambas entidades participantes, en el resto de casos, posiblemente sería más apropiado una operación *UpdateRT*. Esto tendría una consecuencia: las atribuciones que se dan a la operación *UpdateET* [2] quizás deberían ampliarse, ya que también podrían modificar el contenido de un campo participante en una relación.

6. Líneas de desarrollo futuro

Este proyecto me ha permitido conocer un poco más los fundamentos que rigen la generación automática de código a partir de *modelos navegacionales,* así como aquellas posibilidades que estos nos pueden ofrecer a los que, como yo, nos dedicamos –quizás menos de lo que nos gustaría - a la ingeniería de programación.

En este caso, las líneas a seguir, todas ellas relacionadas entre sí, serían las siguientes:

6.1. Ampliación de las funcionalidades del modelo navegacional

- Operaciones *DeleteRT* e *InsertRT* condicionadas a la existencia de un cierto valor en los campos participantes en la relación.

  Cuando se ejecuta la aplicación Web generada por WebRatio y se desea establecer o suprimir una relación nos encontramos con un inconveniente: aparecen todas las filas de las tablas participantes. Esto no supone mayor prpblema si se dispone de un conjunto de datos de prueba reducido, pero podría llegar a serlo si el conjunto fuese mayor.

  Con las operaciones *DeleteRT* la mejora consistiría en mostrar sólo aquellas filas cuyo campo de relación se encuentra informado. Con las operaciones *InsertRT* la mejora sería la opuesta, es decir, sólo se mostrarían aquellas filas cuyo campo de relación no esté informado.

- Añadir una operación UpdateRT al *modelo navegacional*.

  Esto obligaría a modificar el algoritmo [2, definition 3.3.4 y 3.3.5], pero permitiría nuevas posibilidades en el momento de calcular las dependencias de las operaciones. Todo ello, como es lógico, supeditado a que sea factible su modelado con WebRatio, ya que la metodología WebML sólo incorpora dos operaciones para tratar relaciones.

6.2. Ampliación de las funcionalidades del programa

- Comprobar directamente si los caminos obtenidos son correctos y completos.

  Si para una operación se generan varios caminos y alguno de ellos es incorrecto o incompleto, quizás sería más sencillo no replantearse la validez del algoritmo que calcula las dependencias y buscar un mecanismo que anule todos aquellos caminos que no cumplan las propiedades mencionadas.

6.3. Mejorar el algoritmo de cálculo de dependencias

Con alguna de las líneas de desarrollo mencionadas sería suficiente para obtener un *modelo navegacional* completo y correcto, pero la pretensión fuera otra: generar modelos navegaciones que permitan crear aplicaciones de gestión totalmente funcionales. Sería éste un objetivo muy ambicioso, pero resultaría factible siempre que se solucionaran algunos aspectos:

- No se tiene en cuenta la cardinalidad máxima en la inserción y/o borrado de entidades y relaciones, por lo que no se puede garantizar la consistencia de la base de datos.

- Una cardinalidad mínima de 0 en un extremo de la relación se trata como una relación no obligatoria, aunque ello no implica que no exista.

7. Glosario

Árbol de operaciones: estructura que contiene todos los posibles caminos operacionales.

Camino operacional: camino formado exclusivamente por operaciones, sin ningún otro elemento adicional.

Camino de operaciones: camino operacional.

Dependencia: necesidad de que se ejecuten otras operaciones antes o después de una operación concreta.

Diccionario de dependencias: contiene todas aquellas operaciones definidas sobre alguna entidad (*InsertET, DeleteET*) o relación (*InsertRT y DeleteRT*) del *modelo de datos*, así como las *dependencias* de cada una de éstas.

DTD (Document Type Definition): la definición de tipo de documento es una descripción de estructura y sintaxis de un documento XML. Su función básica es la descripción del formato de datos, para usar un formato común y mantener la consistencia entre todos los documentos que utilicen la misma DTD.

Enlace (Link): es la conexión, en un sentido, que se establece entre dos nodos de un árbol/grafo.

Esquema XML: lenguaje de esquema utilizado para describir la estructura y las restricciones de los contenidos de un documento XML.

Grafo navegacional: grafo operacional al que se le ha añadido información navegacional (páginas, enlaces y otros elementos).

Grafo operacional: grafo creado a partir de un árbol de operaciones y sin información navegacional.

Head-Merge: refactorización Head-Merge.

JDBC: paquete para el lenguaje de programación Java que permite la ejecución de operaciones sobre bases de datos con independencia del sistema operativo donde se ejecute o la base de datos a la que se accede.

Modalidad : Aplicable a los algoritmos de refactorización Head_Merge y Tail-Merge. Se definen dos modalidades: *de grafo completo y por operación*.

Modalidad de grafo completo : el proceso de refactorización se aplicará a todo el grafo, sin tener en consideración la *operación identificativa* de los distintos *caminos operacionales*.

Modalidad estándar : cuando el algoritmo Tail-Merge opera en *modalidad de grafo completo* y el algoritmo Head-Merge opera en *modalidad por operación*.

Modalidad por operación : el proceso de refactorización se aplicará de forma individual a aquellos caminos del grafo que tengan la misma *operación identificativa de camino*.

Operación identificativa de camino: operación *op* que identifica un *camino operacional* o secuencia de operaciones.

Refactorización Head-Merge: proceso que une aquella parte de camino que contiene las mismas operaciones iniciales.

Refactorización Tail-Merge: proceso de refactorización que une aquella parte de camino que contiene las mismas operaciones finales.

Tail-Merge: refactorización Tail-Merge.

8. Bibliografía y referencias

[1] J. Cabot, E. Teniente. *Generación Automática de Restricciones de Integridad: Estado del Arte*. http://www.dsic.upv.es/workshops/dsdm05/files/06-Cabot.pdf

[2] J. Cabot, J. Ceballos, C. Gómez. *On the Quality of Navigation Models with Content-Modificacion Operations.*

[3] J. Preciado, M. Linaje, D. Sánchez: *Un Entorno Cooperativo para la Generación Automática de Aplicaciones Web*. JISBD 2006, pp. 2-4 , 2006.

[4] M. Brambilla, J. Cabot. *Constraint tuning and management for web applications*. ICWE'06,

[5] A. Bozzon, S. Comai, P. Fraternali. G.Toffetti, *Capturing RIA Concepts in a Web Modeling*. http://www2006.org/programme/files/xhtml/p162/pp162-Bozzon/pp162-Bozzon-xhtml.html

[6] W. Clay Richardson, D. Avondolio, J. Vitale. (2005). *Professional Java 2 v5.0*. Anaya Multimedia. ISBN: 84-415-1855-6.

[7] R. C. Martin (2004). *UML para Programadores Java*. Pearson Prentice Hall. ISBN: 84-205-4109-5.

[8] S. Stelting, O. Maassen. (2003) *Patrones de diseño aplicados a Java*. Pearson Prentice Hall. ISBN: 84-205-3839-6.

[9] MSDN España. *Arquitectura de Software*. http://www.microsoft.com/spanish/msdn/arquitectura/roadmap_arq/arquitectura_soft.asp

[10] Información general de WebML (Web Modelling Language). http://www.webml.org

[11] WebML DTD http://www.webml.org/webml/page18.do?dau70.oid=27&UserCtxParam=0&GroupCtxParam=0&ctx1=EN

[12] Información general de WebRatio http://www.webratio.com

[13] JDOM JavaDocs http://www.jdom.org/docs/apidocs/index.html

[14] J.Martin (2007). *PFC: Generación automática de código con WebRatio. Análisis de los ficheros XML*. UOC

9. Anexo 1: Herramientas utilizadas

Las herramientas, software complementario y paquetes de utilidades utilizados en la implementación son los siguientes:

- NetBeans v 5.5

  Entorno de desarrollo para programadores que simplifica, entre otras, las tareas de codificación, depuración y despliegue de aplicaciones basadas en Java.

- JDK 1.6.0

  Java Development Kit que incorpora un conjunto de herramientas, utilidades y documentación asociada para desarrollar aplicaciones en Java.

- JDOM v 1.0

  Proporciona un solución simple e intuitiva para acceder, manipular y formatear datos XML desde aplicativos codificados con Java.

- Tomcat v 5.5

  Servidor de aplicaciones que implementa las especificaciones de los Servlet (2.4) y JSP (2.0) de Sun Microsystems y que funciona en cualquier sistema operativo que disponga de la máquina virtual de Java.

  Éste forma parte de la distribución de WebRatio.

- MySql v 5.0.27

  Sistema gestor de base de datos. Las entidades del modelo conceptual serán tablas de la base de datos, encargándose WebRatio de su creación.

- MySql Connector/J v 5.0.4

  Driver JDBC oficial para MySQL.

- WebRatio v 4.3 Academic Trial Edition (Build 070228)