UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

DISEÑO Y DESARROLLO DE MECANISMOS PARA LA CONFIGURACIÓN DE TRANSFORMACIONES DE MODELOS EN MOSKITT

TESIS
Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Miguel Llàcer San Fernando

PRODEVELOP

Septiembre 2008

Dirigida Por:
Vicente Pelechano Ferragud
Javier Muñoz Ferrara

Índice general

Anexos

Índice de Figuras

Índice de Tablas

Lista de Abreviaturas

CIT	Consellería de Infraestructura y Transporte
CASE	Computer Aided Software Engineering
MOSKitt	MOdeling Software Kitt
BPMN	Business Process Modeling Notation
MDA	Model Driven Architecture
OMG	Object Management Group
MOF	Meta Object Facility
UML	Unified Modeling Language
EMF	Eclipse Modeling Framework
ATL	Atlas Transformation Language

Capítulo 1 – Introducción

En este capítulo se presenta el entorno en el que se ha desarrollado el trabajo del Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información.

Este trabajo se ha desarrollado con una orientación profesional en el seno de la empresa Prodevelop y para un proyecto concreto de la Consellería de Infraestructura y Transporte (CIT). Dicho proyecto tiene como principal objetivo el desarrollo de una herramienta CASE que de soporte a la aplicación de la metodología gvMétrica durante el Desarrollo de Sistemas de Información en la CIT.

Dentro de este proyecto, aparece la necesidad de implementar una infraestructura para dar soporte a la transformación entre modelos y a la configuración de transformaciones.

Este capítulo se estructura de la siguiente manera: En primer lugar se presenta una introducción al trabajo realizado. En el siguiente apartado presentamos el entorno en el que se ha desarrollado y una breve introducción al proyecto donde se enmarca. Finalmente se describe el problema, los objetivos y el diseño propuesto.

1.1. Introducción

Este trabajo, desarrollado para el máster en Ingeniería del Software, Métodos Formales y Sistemas de Información, tiene una orientación profesional. En él se expone un trabajo de tipo práctico relacionado con una de las actividades profesionales de un titulado en informática. El trabajo ha sido desarrollado en la empresa Prodevelop, junto con otras empresas y organizaciones, en el contexto de un proyecto real para la Consellería de Infraestructura y Transporte.

La CIT, utiliza una metodología concreta (gvMétrica) para el desarrollo de Sistemas de Información. Dentro de este proceso de desarrollo de software, en el que los modelos son las principales fuentes de información, surge la necesidad de mantener dichos modelos. Cada uno de estos modelos almacena distinta información y en algunas ocasiones unos modelos dependen de otros. Es en este punto cuando surge la necesidad de obtener modelos a partir de otros y de mantenerlos sincronizados. Así pues, este trabajo tiene como principal objetivo explicar de manera detallada la transformación y sincronización de modelos así como la configuración de las transformaciones para el proyecto MOSKitt [1].

1.2. Entorno de trabajo

Como ya hemos comentado, este trabajo se está desarrollando dentro de la empresa Prodevelop, como parte del proyecto MOSKitt. Además de Prodevelop, en el proyecto también participan: la CIT, la empresa *Integranova* y el grupo *OO-Method* del DSIC (Departamento de Sistemas Informáticos y Computación) de la UPV (Universidad Politécnica de Valencia).

El proyecto MOSKitt, uno de los proyectos integrados dentro de gvPontis (proyecto global de la CIT para la migración de todo su entorno tecnológico a software libre), surge por la necesidad de una herramienta CASE de software libre que de soporte a la metodología gvMétrica (adaptación de Métrica III) utilizada en la Consellería de Infraestructuras y Transporte.

En esta adaptación, se han definido e identificado los Métodos, Prácticas y Técnicas a aplicar en el proceso de Desarrollo de los Sistemas de Información, usando el lenguaje de modelado estándar UML propuesto por la OMG y la Guía de Estilo definida para las aplicaciones de la CIT.

Así pues, los objetivos perseguidos tras varios estudios realizados por la CIT y el DSIC son:

- Desarrollo de una Herramienta CASE que de soporte a la aplicación de la metodología gvMétrica durante el Desarrollo de Sistemas de Información en la CIT. En general para cumplir este requisito la herramienta a desarrollar debe dar principalmente soporte a tres tipos de tareas:
    - Editar modelos (mayoritariamente se trata de modelos UML).
    - Enlazar modelos dependientes (Generar + Producir traza y/o dependencias + Mantener consistencia).
    - Producir activos a partir de los modelos (Documentación, DDL, etc.)
- La herramienta proporcionará soporte al aspecto procedimental del método definido por gvMétrica, guiando a los usuarios en los distintos pasos que deben realizar para llevar a cabo sus tareas.
- Deberá tratarse de una solución dentro del paradigma del Software Libre, basada en estándares, multiplataforma e interoperable.
- La herramienta deberá interoperar en la medida de lo posible con el resto de herramientas utilizadas por el personal de la CIT durante el Desarrollo de sus Sistemas de Información.
- La herramienta deberá dar soporte a la importación de los modelos actualmente especificados utilizando PowerDesigner.
- La herramienta proporcionará soporte al trabajo colaborativo entre grupos de usuarios, permitiendo la gestión de versiones de los modelos implicados en un proyecto de desarrollo, su comparación y sincronización.
- Deberá diseñarse de forma que se trate de una herramienta que pueda integrarse bien con otras aplicaciones para ser así extendida en un futuro. Se pretende tener una arquitectura modular y extensible, de forma que la herramienta se desarrolle como un conjunto de herramientas o plug-ins de Eclipse que colaboren entre sí.

1.3. Problemas

En la actualidad, la CIT utiliza PowerDesigner, de la empresa SYBASE, como herramienta de modelado de datos y modelado de procesos de negocio, pero esta herramienta presenta ciertos inconvenientes de acuerdo a los objetivos presentados anteriormente:

- No es una herramienta libre.
- No mantiene los modelos sincronizados.
- No da soporte al proceso de Desarrollo de Sistemas de Información, definido por gvMétrica.
- No proporciona soporte para el trabajo en equipo y se tienen que utilizar herramientas adicionales.

A raíz de estos inconvenientes y de acuerdo a los objetivos definidos por y para el proyecto, surge la herramienta MOSKitt (Modeling Software KITT).

Siguiendo una arquitectura modular, MOSKitt está compuesta por los siguientes módulos los cuales están siendo desarrollados en los siguientes subproyectos semi-independientes:

- **MOSKitt-UML2**: Modelador de UML 2.0 con soporte a la edición de Diagramas de Clases, de Casos de Uso, de Secuencia, de Actividad, de Estados y de Perfiles.
- **MOSKitt-DB**: Modelador de Esquemas de Bases de Datos Relacionales: edición gráfica de los esquemas de bases de datos relacionales, con soporte a los niveles lógico y físico, generación de código DDL e ingeniería inversa a partir de esquemas Postgresql 8.X, MySql 5, Oracle 8i y Oracle 10.
- **MOSKitt-TrManager**: Gestor de Transformaciones Modelo-A-Modelo y Modelo-A-Texto.
- **MOSkitt-ModelSync**: Gestor de Sincronización entre modelos y soporte a la Configuración de transformaciones. Además, este módulo define los editores basados en formularios para las trazas entre modelos y la configuración de modelos.
- **MOSKitt-gvMetrica**: Soporte al proceso de desarrollo definido por gvMétrica. MOSKitt asiste a los distintos participantes en el proceso de desarrollo durante la generación de cada uno de los productos definidos por la metodología. Este módulo definirá la semántica de los enlaces entre los modelos definidos a través de los distintos modeladores proporcionados por la herramienta.
- **MOSKitt-Interop**: Módulo de Interoperabilidad que permitirá a MOSKitt interoperar con otras herramientas comenzando por aquellas actualmente utilizadas por la CIT, en este sentido ya ha sido implementada la importación de modelos UML definidos con PowerDesigner.

- **MOSKitt-RMP**: Repositorio con soporte para el trabajo colaborativo: concurrencia, seguridad basada en roles, almacenamiento, mantenimiento y versionado tanto de modelos como de otro tipo de productos propuestos por la metodología.

- **MOSKitt-REQ**: Modelador de Requisitos para la generación y mantenimiento de un catálogo de requisitos, enlace de requisitos con los modelos, jerarquía entre requisitos, trazabilidad entre requisitos y casos de uso, generación de matrices de trazabilidad.

- **MOSKitt-EIU**: Modelador de Interfaces de Usuario.

- **MOSKitt-MPR**: Módulo para Modelado de Procesos, incluye editor BPMN.

- **MOSKitt-WBS**: Editor de EDT, Estructura de Descomposición de Trabajo o Estructura de Desglose del Trabajo (EDT) (en inglés Work Breakdown Structure, WBS). Es una estructura exhaustiva, jerárquica y descendente formada por los entregables y las tareas necesarias para completar un proyecto.

1.4. Objetivos

El objetivo general de este trabajo de tesis de máster es presentar una infraestructura y los mecanismos necesarios para la configuración de transformaciones de modelos. Así pues, este trabajo solamente abarca los módulos de transformación (**MOSKitt-TrManager**) y sincronización/configuración (**MOSkitt-ModelSync**).

Para desarrollar esta infraestructura, habrá que analizar:

- La tecnología a utilizar para la transformación y sincronización de modelos.

- La infraestructura genérica a diseñar para almacenar la configuración de transformaciones.

- Los requisitos o criterios a tener en cuenta para realizar las transformaciones entre modelos.

- Los modelos a configurar/transformar.

Finalmente, se presenta una propuesta de editor basado en formularios para la configuración de transformaciones. Este editor tiene como objetivo agilizar y hacer más intuitiva la edición de los modelos de configuración por parte del usuario.

Capítulo 2 – Contexto tecnológico

En este capítulo se presentan las tecnologías y estándares existentes y que han servido de base al trabajo realizado. Primero se presenta MDA (Model Driven Architecture), como una iniciativa de la OMG que define una aproximación para el desarrollo de software basada en modelado y transformaciones entre modelos. A continuación se presenta Eclipse, que es un entorno de desarrollo de código abierto multiplataforma, sobre el cuál se está desarrollando el proyecto. En el tercer apartado se presenta EMF (Eclipse Modeling Framework), que como su nombre indica, es un framework de modelado para eclipse con facilidad de generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado. Para finalizar se comenta ATL (Atlas Transformation Language), que es el lenguaje que se va a utilizar para definir las transformaciones entre modelos.

2.1. Model Driven Architecture

En los últimos años, muchas organizaciones han comenzado a prestar atención a MDA, ya que promueve el uso eficiente de modelos de sistemas en el proceso de desarrollo de software.

MDA [2] representa, para los desarrolladores, una nueva manera de organizar y administrar arquitecturas empresariales, basada en la utilización de herramientas de automatización de etapas en el ciclo de desarrollo y servicios. De esta forma, permite definir los modelos y facilitar trasformaciones paulatinas entre diferentes modelos. Algunos ejemplos de modelos son: el modelo de análisis, el de diseño y el de comportamiento, entre otros. Es decir que, a partir de uno de ellos, podemos generar otro de menor abstracción.

Un ejemplo común de generación de modelos, es la generación de código a partir del modelo de diseño, mediante el uso de una herramienta de modelado UML. En este caso usamos una herramienta para transformar el modelo de diseño en el "modelo" de código.

2.1.1. ¿Qué es MDA?

En el año 2001, el OMG (Object Management Group) estableció el framework MDA, acrónimo de **Model Driven Architecture** (Arquitectura Dirigida por Modelos), como arquitectura para el desarrollo de aplicaciones. Mientras que el anterior framework propuesto por OMG, formado por las especificaciones OMA y CORBA, estaba destinado al desarrollo de aplicaciones distribuidas, MDA representa un nuevo paradigma de desarrollo de software en el que los modelos guían todo el proceso de desarrollo. Este nuevo paradigma se ha denominado **Ingeniería de modelos** o **Desarrollo basado en modelos**.

2.1.2. Taxonomía de modelos en MDA

En la actualidad, la construcción de software se enfrenta a continuos cambios en las tecnologías de implantación, lo que implica realizar esfuerzos importantes tanto en el diseño de la aplicación, para integrar las diferentes tecnologías que incorpora, como en el mantenimiento para adaptar la aplicación a cambios en los requisitos y en las tecnologías de implementación. Por otra parte, las aplicaciones distribuidas Business to Business (B2B) y Client to Business (C2B) son cada vez más comunes, siendo difícil satisfacer los requisitos de escalabilidad, seguridad y eficiencia. La idea clave que subyace a MDA es que si el desarrollo está guiado por los modelos del software, se obtendrán beneficios importantes en aspectos fundamentales tales como son la **productividad**, la **portabilidad**, la **interoperabilidad** y el **mantenimiento**.

Para conseguir estos beneficios, el proceso de desarrollo que plantea MDA se puede dividir en tres fases:

- Construcción de un **Modelo Independiente de la Plataforma** (Platform Independent Model o PIM), un modelo de alto nivel del sistema independiente de cualquier tecnología o plataforma.
- Trasformación del modelo anterior a uno o varios **Modelos Específicos de la Plataforma** (Platform Specific Model o PSM). Un PSM es un modelo de más bajo nivel que el PIM que describe el sistema de acuerdo con una tecnología de implementación determinada.
- Generación de código a partir de cada PSM. Debido a que cada PSM está muy ligado a una tecnología concreta, la transformación de cada PSM a código puede automatizarse.

El paso de PIM a PSM y de PSM a código no se realiza "a mano", sino que se usan herramientas de transformación para automatizar estas tareas. Estas herramientas de transformación son, de hecho, uno de los elementos básicos de MDA.

2.2. Eclipse

La plataforma de desarrollo que se ha usado para implementar los proyectos, es Eclipse [6]. Este framework nos proporciona un completo entorno de trabajo con el que podemos desarrollar fácilmente nuestros proyectos, así como disponer de una ayuda en línea muy completa.

Eclipse es una herramienta de código libre, que se puede descargar gratuitamente desde su página Web. Se basa en el concepto de plug-ins, que son los encargados de darle funcionalidad a Eclipse. Cada uno de ellos le aporta algo nuevo, de modo que el conjunto de todos ellos forma una potente herramienta, que nos sirve para desarrollar nuestro propósito.

El lenguaje de programación principal que maneja Eclipse, es Java. Posee una función de ayuda al programador, llamada Intelligence, que va autocompletando el código como se va escribiendo, mostrando las posibles alternativas. El hecho de que el lenguaje

principal sea Java, ofrece a nuestros proyectos una completa portabilidad, ya que este lenguaje es ejecutable en muchos sistemas operativos, como Windows y Linux.

El trabajo de Eclipse se basa en un proyecto central, el cual incluye un framework genérico para la integración de las herramientas, y un entorno de desarrollo Java construido utilizando el primero. Otros proyectos extienden el framework central para soportar clases específicas de herramientas y entornos de desarrollo.

El software producido por Eclipse está disponible bajo la Common Public License (CPL), la cual básicamente dice que puedes usar, modificar, y distribuir el software de manera gratuita, o incluirlo como parte de un producto propietario. CPL es una Open Source Initiative (OSI), aprobada y reconocida por la Free Software Foundation (FSF) como una licencia de software libre. Cualquier software que contribuya a Eclipse debe hacerlo bajo la licencia Common Public License (CPL).

Eclipse.org es un consorcio de compañías las cuales tienen un acuerdo para proporcionar el soporte esencial al proyecto Eclipse en términos de tiempo, experiencia, tecnología y conocimiento.

2.2.1. Los proyectos

El trabajo de desarrollo en Eclipse se divide en tres proyectos principales:

- el proyecto Eclipse,
- el proyecto de Herramientas y
- el proyecto de Tecnología.

El proyecto Eclipse es el que contiene los componentes centrales necesarios para desarrollar mediante Eclipse. Sus componentes conforman una única unidad conocida como Eclipse SDK (Software Development Kit). Los componentes de los otros dos proyectos son usados para propósitos específicos y son generalmente independientes.

2.2.1.1. El proyecto Eclipse

El proyecto Eclipse soporta el desarrollo de una plataforma, o framework, para la implementación de entornos de desarrollo integrados (Integrated Development Environments, IDEs). El framework Eclipse esta implementado utilizando Java pero es usado para implementar herramientas de desarrollo para otros lenguajes (por ejemplo, C++ y XML, entre otros).

El proyecto Eclipse se divide en tres subproyectos:

- La Plataforma (Platform) es el componente central de Eclipse, y es a menudo considerado como el propio Eclipse. Se utiliza para definir frameworks y los servicios

requeridos para soportar la conexión e integración de las herramientas.

- En segundo lugar, las Herramientas de Desarrollo Java (Java Development Tools, JDT) proporcionan un completo entorno de desarrollo Java. Pueden ser utilizadas para desarrollar programas Java para Eclipse o para otras plataformas.

- Por último, el Entorno de Desarrollo de Plug-ins (Plug-in Development Environment, PDE) proporciona vistas y editores para facilitar la creación de plug-ins para Eclipse. Además proporciona soporte para actividades propias del desarrollo de un plug-in, como el registro de extensiones.

En conjunto, estos tres subproyectos proporcionan todo lo necesario para extender el framework y desarrollar herramientas basadas en Eclipse.

2.2.1.2. El proyecto de Herramientas

Este proyecto define y coordina la integración de diferentes conjuntos o categorías de herramientas basadas en la plataforma Eclipse. El subproyecto de Herramientas de Desarrollo C/C++ (C/C++ Development Tools, CDT), por ejemplo, engloba un conjunto de herramientas que definen un IDE C++.

Los editores gráficos que utilizan Graphical Editing Framework (GEF), y los editores basados en modelos usando Eclipse Modeling Framework (EMF) representan categorías de herramientas de Eclipse para los cuales se da un soporte común desde los subproyectos de Herramientas.

2.2.1.3. El proyecto de Tecnología

El proyecto de Tecnología proporciona una oportunidad para los investigadores y educadores para formar parte de la continua evolución de Eclipse.

2.2.2. Descripción de la plataforma Eclipse

La Plataforma Eclipse es un framework para construir entornos de desarrollo integrados (IDEs). Ha sido descrita como "una IDE para cualquier cosa, y para nada en particular". Su cometido es definir una estructura básica para un IDE. Las herramientas específicas extienden el framework, conectándose a él para definir un IDE particular de manera colectiva.

Veamos una descripción de los componentes principales de la Plataforma Eclipse:

2.2.2.1. La arquitectura de plug-ins

Una unidad funcional básica, o componente, en Eclipse se conoce como plug-in. La misma Plataforma Eclipse, y las herramientas que la extienden, están compuestas por plug-ins.

Desde una perspectiva de paquetes, un plug-in incluye todo lo que se necesita para que funcione un componente como código Java, imágenes, texto traducido, etc.

También incluye un archivo de manifiesto, llamado plugin.xml, que declara las interconexiones con otros plug-ins. En este archivo se declaran, entre otras cosas, lo siguiente:

- Required plug-ins → sus dependencias con otros plug-ins.
- Exported plug-ins → la visibilidad de sus clases públicas con otros plug-ins.
- Extension points → declaración de la funcionalidad que se pone a la disposición de otros plug-ins.
- Extensions → implementación de puntos de extensión de otros plug-ins.

En el arranque, la Plataforma Eclipse (específicamente Platform Runtime) descubre todos los plug-ins disponibles y asocia las extensiones con sus correspondientes puntos de extensión. Un plug-in, sin embargo, solamente se activa cuando su código necesita ejecutarse, evitando una lenta secuencia de arranque.

Cuando se activa un plug-in, es asignado a su propia clase de carga, la cual provee la visibilidad declarada en su archivo de manifiesto.

2.2.2.2. Recursos del espacio de trabajo (Workspace)

Las herramientas integradas en Eclipse trabajan con archivos y carpetas ordinarios, pero utilizan una API de alto nivel basada en recursos (resources), proyectos (projects), y un espacio de trabajo (workspace).

Un recurso es la representación que utiliza Eclipse para los archivos y carpetas, proporcionando funcionalidades adicionales (registro de controladores de cambios en el recurso, marcadores, mantenimiento de historiales,…).

Un proyecto es un tipo especial de recurso que se asocia a una carpeta del usuario en el sistema de ficheros. Las subcarpetas del proyecto son las mismas que las de la carpeta física, pero los proyectos son carpetas de alto nivel en un contenedor virtual de proyectos, llamado espacio de trabajo.

2.2.2.3. El Framework UI

El framework UI (User Interface) de Eclipse consiste en dos conjuntos de herramientas de propósito general, SWT y JFace, y el Escritorio de Eclipse (workbench UI).

SWT (Standard Widget Toolkit) es un conjunto de librerías gráficas independientes del sistema operativo utilizado.

JFace es un conjunto de herramientas de alto nivel, implementado usando SWT. Proporciona clases para soportar tareas comunes relativas a interfaces de usuario, como manejo de registros de imágenes y fuentes, diálogos, asistentes, monitores de progreso, etc. Una parte importante de JFace son las clases utilizadas como visores para listas, árboles y tablas, que proporcionan un mayor nivel de conexión con los datos que SWT (por ejemplo, mecanismos de población a partir de un modelo de datos y sincronización con este). Otra parte importante de JFace es su framework para acciones, usado para añadir comandos a menús y barras de herramientas.

Por último, el Escritorio de Eclipse (workbench) es la ventana principal que el usuario ve cuando arranca Eclipse. Esta implementado mediante SWT y JFace. Como interfaz principal de usuario, se considera a menudo que es la propia Plataforma.

2.2.2.4. Soporte para el trabajo en grupo

La Plataforma Eclipse permite asignar una versión a un proyecto en el workspace y gestionar sus versiones mediante un repositorio de trabajo en grupo. Pueden coexistir multitud de repositorios de trabajo en grupo en la Plataforma, no obstante la Plataforma Eclipse incluye, por defecto, soporte para repositorios CVS accedidos mediante protocolos pserver o ssh.

2.2.2.5. Ayuda

Eclipse Platform Help incluye herramientas que permiten definir y contribuir a la documentación de uno o más libros en línea.

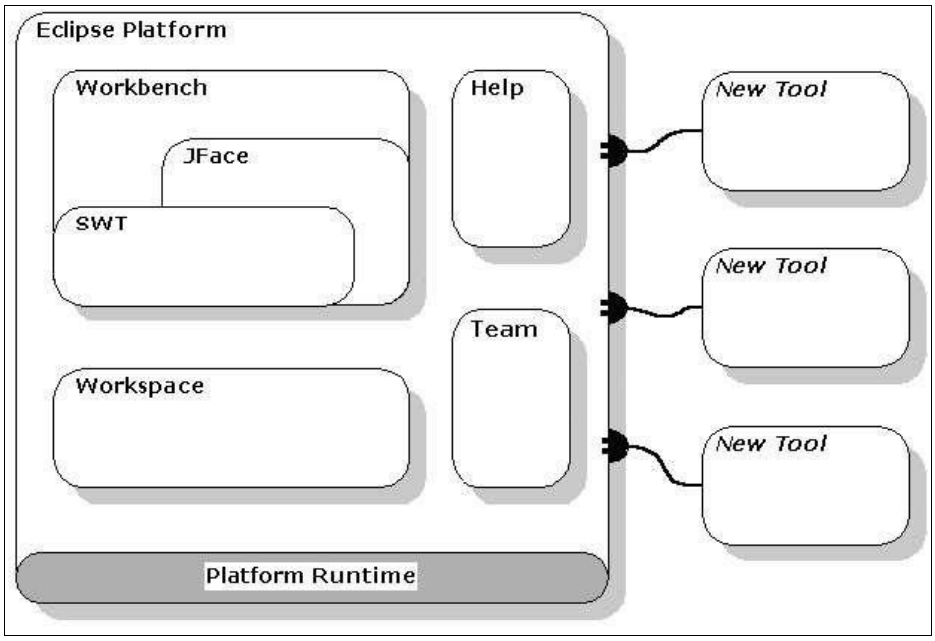


Figura 1. Arquitectura de la Plataforma Eclipse

2.3. Eclipse Modeling Framework

EMF [7] es básicamente "un entorno de modelado y una herramienta de generación de código para la plataforma Eclipse" [Budinsky2003]. Su objetivo es construir herramientas y otras aplicaciones basadas en modelos estructurados. EMF ayuda además a generar código Java a partir de estos modelos de una manera fácil, correcta, personalizable y eficiente. EMF utiliza XMI como una manera canónica de definir y persistir los modelos. Además proporciona un editor gráfico para definir modelos.

Cuando se ha especificado un MetaModelo EMF, el generador EMF puede crear una serie de clases Java que permiten crear instancias del MetaModelo y manipular sus elementos. Una vez generado el código se pueden editar las clases generadas para añadir nuevos métodos y variables. Además, se proporcionan mecanismos para la notificación de cambios en el modelo, serialización por defecto en XMI y XML Schema, un entorno de trabajo para la validación de modelos, y una API reflexiva eficiente para manipular objetos EMF de manera genérica. Y lo más importante, EMF proporciona las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.

EMF consiste en dos entornos de trabajo fundamentales: el core y EMF.Edit. El entorno core proporciona la generación de código básica y el soporte en tiempo de ejecución para la creación de las clases Java del modelo. El EMF.Edit extiende y se construye sobre el core, añadiendo soporte para la generación de las clases que permiten utilizar los comandos para editar los modelos, además de activar los visores y editores gráficos básicos para un modelo.

2.3.1. Ecore

EMF comenzó como una implementación de la especificación MOF de OMG. En realidad EMF se puede considerar una eficiente implementación en Java de un subconjunto del core de la API de MOF (Essential MOF). Sin embargo, para evitar cualquier confusión, el MetaMetaModelo en EMF se llama Ecore. Por tanto, Ecore permite definir los vocabularios locales de dominio o MetaModelos, que permiten la creación o modificación de modelos en distintos contextos. Ecore es un vocabulario diseñado para permitir la definición de cualquier tipo de MetaModelos. Para ello proporciona elementos útiles para describir conceptos y relaciones entre ellos. En definitiva, Ecore es un subconjunto de MOF, el cual esta basado en el diagrama de clases de UML. En la siguiente figura se muestran los elementos principales del MetaModelo de Ecore.

El elemento más importante es EClass, que modela el concepto de clase, con una semántica similar al elemento Clase de UML. EClass es el mecanismo principal para describir conceptos mediante Ecore. Una EClass está compuesta por un conjunto de atributos y referencias, así como por un número de super clases (el símil con UML sigue siendo aplicable). A continuación se comentan el resto de los elementos aparecidos en el diagrama:

- **EClassifier**. Tipo abstracto que agrupa a todos los elementos que describen conceptos.

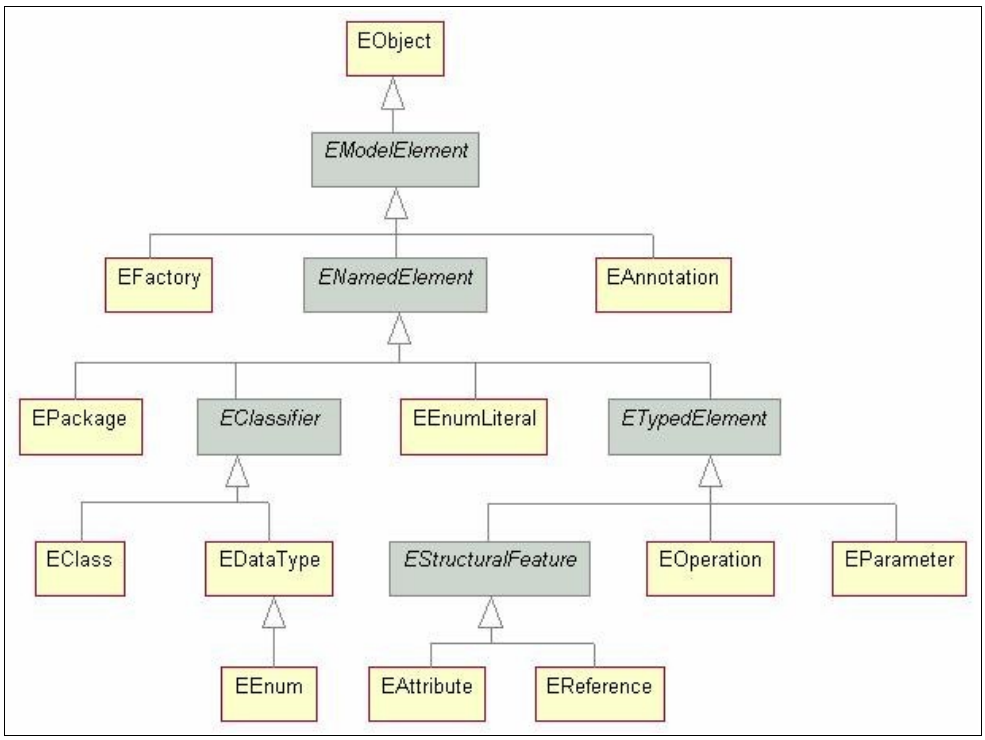


Figura 2. Jerarquía de Clases Ecore

- **EDataType** se utiliza para representar el tipo de un atributo. Un tipo de datos puede ser un tipo básico como int o float o un objeto, como por ejemplo java.util.Date.

- **EAttribute**. Tipo que permite definir los atributos de una clase. Estos tienen nombre y tipo. Como especialización de ETypedElement, EAttribute hereda un conjunto de propiedades como cardinalidad (lowerBound, upperBound), si es un atributo requerido o no, si es derivado, etc.

- **EReference**. Permite modelar las relaciones entre clases. En concreto EReference permite modelar las relaciones de asociación, agregación y composición que aparecen en UML [4]. Al igual que EAttribute, es una especialización de ETypedElement, y hereda las mismas propiedades. Además define la propiedad containment mediante la cual se modelan las agregaciones disjuntas (denominadas composiciones en UML).

- **EPackage** agrupa un conjunto de clases en forma de modulo, de forma similar a un paquete en UML. Sus atributos más importantes son el nombre, el prefijo y la URI.

La URI es un identificador único gracias al cual el paquete puede ser identificado unívocamente.

Una de las interfaces claves en Ecore es EObject, la cual es conceptualmente equivalente a java.lang.Object. Todos los objetos modelados implementan esta interfaz para proporcionar varias características importantes:

De forma similar a Object.getClass(), utilizando el método eClass() se puede obtener los Metadatos de la instancia. En un objeto modelado ecore se pueden utilizar los

métodos reflexivos del API (eGet(), eSet()) para acceder a sus datos. Esto es conceptualmente equivalente al método de java java.lang.reflect.Method.invoke().

De cualquier instancia de objeto se puede obtener su contenedor (padre) utilizando el método eContainer(). EObject extiende Notifier, lo cual permite monitorizar todos los cambios en los datos de los objetos. La interfaz Notifier introduce una característica muy importante a cada elemento del modelo: notificaciones de cambios en el modelo como en el patrón de diseño Observador.

2.3.1.1. Creación del Ecore

Un modelo Ecore se puede crear siguiendo dos aproximaciones:

1. Importándolo de un modelo en uno de los siguientes formatos:

    a. Diagrama de Rational case.

    b. XML schema.

    c. Modelo UML.

    d. Java anotado.

2. Utilizando el editor gráfico de modelos Ecore proporcionado por GMF.

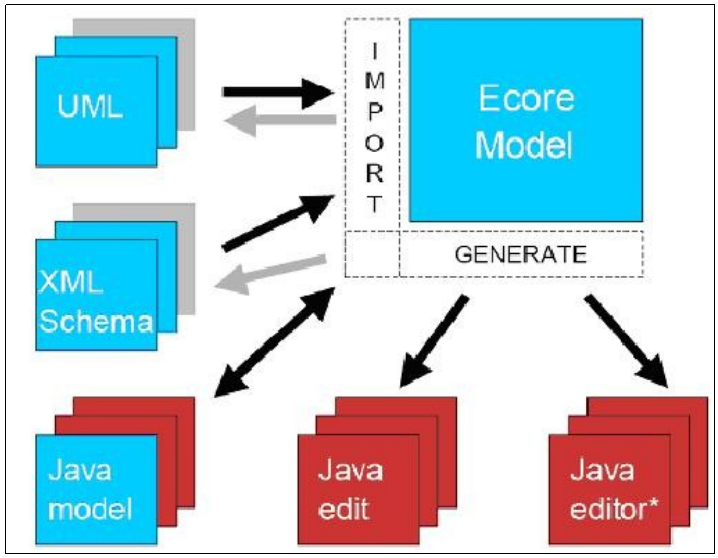


Figura 3. EMF: Importación de modelos

2.3.1.2. Generación de código

A partir de un modelo Ecore, EMF es capaz de generar de una forma simple y eficiente:

- Clases Java que implementan el MetaModelo.
- Clases para la edición independientes de la interfaz de usuario.
- Una vista de edición de modelos integradas en el IDE eclipse.
- Esqueletos para las clases de test de JUnit.

Además de lo anterior también se generan los elementos necesarios para encapsular cada uno de los puntos anteriores en plug-ins de eclipse:

- Manifiestos.
- Archivos properties.
- Iconos.
- Clases relacionadas con los plug-in.

2.4. Atlas Transformation Language

La transformación de modelos constituye una tecnología clave para el éxito de los diversos enfoques de desarrollo de software dirigido por modelos (DSDM). Desde la propuesta de MDA, se han definido varios lenguajes y herramientas de transformación de modelos como resultado de un intenso trabajo de investigación y desarrollo, tanto desde la industria como desde el mundo académico. Sin embargo, todavía es necesario seguir investigando para conocer mejor la naturaleza de las transformaciones y las características deseables de un lenguaje de transformación. Por ello, el interés actual está centrado principalmente en la experimentación con los lenguajes existentes a través de la escritura de definiciones de transformaciones para casos reales.

Existe una amplia oferta de tecnologías disponibles [11] para realizar transformaciones de modelos en el contexto de la plataforma Eclipse. Todas estas tecnologías tienen en común que manipulan modelos en formato Ecore, la tecnología de metamodelado utilizada en EMF. Esta tecnología puede considerarse como una implementación de un subconjunto de MOF, el estándar de OMG para metamodelado.

Atlas Transformation Language [5] (ATL) es la herramienta propuesta por el grupo de investigación ATLAS INRIA & LINA. Es un lenguaje de transformación de modelos especificado tanto como un metamodelo como una sintaxis textual concreta. Actualmente forma parte del subproyecto "M2M" dentro del proyecto "Eclipse Modeling Project".

2.4.1. Características del lenguaje

La manera de tratar los modelos es diferente, es decir, los modelos de entrada solo pueden ser leídos mientras que en los modelos de salida, solo se puede escribir.

El lenguaje es híbrido: declarativo e imperativo, sin embargo, el estilo más utilizado es el declarativo pudiendo expresar mappings fácilmente. En la parte declarativa se definen reglas de matching sobre los patrones de entrada y que generan los patrones de salida. En cambio, en la parte imperativa se definen reglas que podemos considerar básicamente como

métodos o procedimientos (pueden ser llamadas por su nombre y poseen argumentos).

Existen distintos tipos de ficheros ATL, todos ellos con la extensión *.atl*:

- ***ATL module*** - módulo en el que se definen el conjunto de reglas de la transformación.

- ***Library*** - se definen un conjunto de funciones de ayuda (conocidos como *helpers*). Debe ser importado por el módulo.
- ***Query*** - se definen consultas para obtener valores simples de un modelo.

El *patrón origen* está formado por:

- Un conjunto etiquetado de elementos de los metamodelos de origen.
- Una guarda (expresión booleana a modo de filtro)

Se producirá un matching si se encuentran elementos en los modelos origen que sean del tipo especificado en el patrón y satisfacen la guarda.

El *patrón destino* se compone de:

- Elementos etiquetados del metamodelo destino.
- Para cada elemento se inicializan sus propiedades.

Cuando se produce el matching y se satisface la guarda, se crean los elementos destino y se inicializan sus propiedades.

2.4.2. Sintaxis básica

ATL module

```
module module_name;
create output_models : output_metamodels from input_models : input_metamodels;

uses library_name;
```

Donde se especifica:

- El nombre del módulo (module_name)
- Los modelos y metamodelos especificados (OUT : DataBase, IN : UML)
- Si se utiliza alguna librería, se indica mediante la palabra reservada *uses*.

Helpers - Permiten extender el metamodelo con funciones y atributos derivados.

```
helper [context context_type] def : helper_name(parameters) : return_type = exp;
```

- Se consideran atributos si no se usan parámetros.
- Las expresiones se definen en OCL.
- El contexto es opcional.

Rules - Definen la transformación de cada elemento del modelo origen en elementos del modelo destino.

```
rule rule_name {
from
    in_var : in_type [(condition)]
[using {var1 : var_type1 = init_exp1; ... varn : var_typen = init_expn;}]
to
    out_var1 : out_type1 ( bindings1 )
    ...
    out_varn : out_typen ( bindingsn )
[do { statements }]
}
```

Donde se indica:

- El nombre de la regla (rule_name)
- Metamodelo del elemento origen (in_var) y tipo del elemento origen (in_type) con la palabra reservada *from*. Condition es la guarda que tiene que satisfacer el elemento origen para poder ejecutar la regla.
- Metamodelo del elemento destino (out_var) y tipo de elemento destino (out_type) con la palabra reservada *to*. Podemos crear tantos elementos de salida como queramos (out_var_n : out_type_n).
- Finalmente el bloque imperativo *do*.

2.5. Conclusiones

En este capítulo se han presentado las tecnologías y estándares en los que se sitúa el trabajo que se desarrolla para la tesis del máster, con la intención de facilitar el entendimiento de la solución propuesta.

En primer lugar se ha dado una visión general sobre el desarrollo de software dirigido por modelos. A continuación, se ha presentado el entorno de trabajo sobre el cuál se va a desarrollar todo el proyecto y el framework de modelado que se utiliza en dicho entorno. Finalmente, se ha presentado ATL como lenguaje de transformación de modelos.

Capítulo 3 – Configuración de transformaciones de modelos. Análisis y Diseño

Tal y como se presentó en la introducción, el principal objetivo de este trabajo es diseñar una infraestructura genérica para configurar las transformaciones entre modelos dentro de la herramienta MOSKitt.

Para ello, se presentan los conceptos de transformación de modelos y configuración de transformaciones. En términos generales, el concepto de configuración lo podemos entender como edición o modificación de características; en el campo de las transformaciones de modelos, se refiere a la capacidad para modificar dichas transformaciones de acuerdo a los requisitos del software a desarrollar o las necesidades del usuario. Así pues y como solución al problema de como configurar las transformaciones, surge la necesidad de diseñar una infraestructura para gestionar la configuración.

3.1. Transformación de modelos

El metamodelado es un mecanismo que permite construir formalmente lenguajes de modelado, como por ejemplo UML. La arquitectura de 4 capas de Modelado es la propuesta por la OMG orientada a estandarizar conceptos relacionados con el modelado, desde los más abstractos a los más concretos. Los niveles [10] definidos en esta arquitectura se denominan comúnmente: M3, M2, M1, M0:

- El **nivel M3** (el meta-metamodelo) es el nivel más abstracto, donde se encuentra Ecore, que permite definir metamodelos concretos (como el del lenguaje UML [3]).

- El **nivel M2** (el metamodelo), sus elementos son lenguajes de modelado, por ejemplo UML. Los conceptos a este nivel podrían ser Clase, Atributo, Asociación.

- El **nivel M1** (el modelo del sistema), sus elementos son modelos de datos, por ejemplo entidades como "Persona" o "Coche", atributos como "nombre", relaciones entre estas entidades.

- El **nivel M0** (instancias) modela al sistema real. Sus elementos son datos, por ejemplo "Juan López", que vive en "Av. Libertador 345".

En este contexto, la definición de lenguajes para transformación de modelos puede pensarse en la capa M3 de la arquitectura de modelado de 4 capas, ya que una instancia específica de transformación se ubica en la capa M2 para poder relacionar instancias genéricas de metamodelos concretos (que se ubican en M2) como el de UML y el de Base de Datos (BD), entre cuyas instancias se produce la transformación (por ejemplo una Class de UML y una Table de BD). Es decir, los modelos que concretamente están involucrados en la transformación (capa M1) son parámetros para el lenguaje de transformación.

Es lógico pensar que el metamodelo de transformaciones y su instanciación no pueden convivir en la misma capa, ya que representan distintos niveles de abstracción.

Formalmente, la transformación de modelos [9] implica que a partir de un modelo M(a) que cumple cierto metamodelo MM(a) se genera otro modelo M(b) de acuerdo a su metamodelo MM(b).

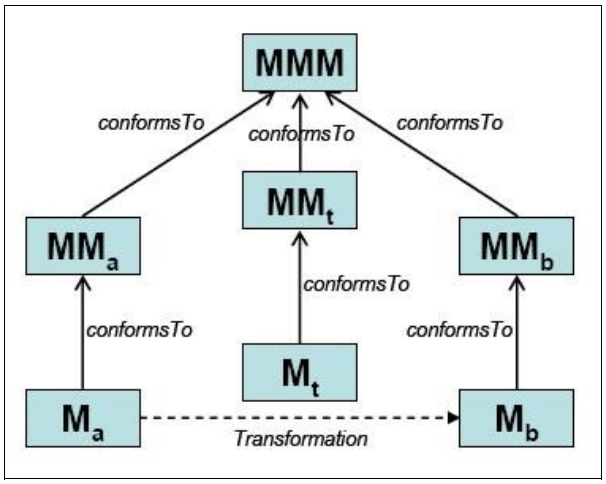


Figura 4. Visión general de la transformación de modelos

Como podemos observar en la figura, un modelo M(a), conforme a un metamodelo MM(a), se transforma en otro modelo M(b) que cumple cierto metamodelo MM(b). La transformación a su vez es definida por otro modelo M(t) que cumple también cierto metamodelo MM(t). Tanto el metamodelo de la transformación MM(t), como los otros dos metamodelos (MM(a) y MM(b)) tienen que cumplir un metamodelo más genérico o metametamodelo (Ecore).

3.2. Configuración de transformaciones

Tal y como se ha comentado en el punto anterior, la transformación de modelos consiste en obtener un modelo a partir de otro de acuerdo a cierta transformación definida previamente.

El problema que surge es que, a partir de un modelo de entrada y una transformación definida, el resultado siempre será el mismo, es decir, el usuario NO puede decidir cómo o qué se tiene que transformar.

Por ejemplo, para el caso concreto de la transformación entre modelos de diagrama de clases UML y modelos de Base de Datos, en ocasiones la transformación no siempre es la misma ya que depende de diferentes factores (requisitos del cliente, diseño o punto de vista del analista, restricciones o limitaciones del SGBD, etc.), y dependiendo de estos factores, un mismo modelo de entrada se puede transformar en diferentes modelos de salida.

Así pues, hay que considerar la taxonomía presentada en [11], donde se define una clasificación para las distintas aproximaciones sobre transformación de modelos y se identifican problemas y retos en dicho ámbito.

Uno de los retos dentro de esta taxonomía es el de la parametrización (*Parameterization*). Principalmente se basa en el uso de parámetros para especificar cual de las reglas de transformación se debe aplicar en cada caso.

Basándonos en este concepto, surge la idea de configurar las transformaciones, esto es, proporcionar mecanismos para que los lenguajes o herramientas de transformación de modelos puedan tomar en consideración distintas estrategias o criterios a la hora de transformar un modelo origen en otro modelo de destino. Además, dichas transformaciones mantendrán la sincronización entre modelos, es decir, ante cualquier cambio en el modelo de entrada, este se verá reflejado en el modelo de salida.

El concepto de sincronización también se contemplan dentro de la taxonomía [11], referenciado como trazabilidad (*Tracing*). En la transformación de modelos, el concepto de trazabilidad se entiende como una forma de mantener información que conecta o relaciona elementos del modelo origen con elementos del modelo destino.

A continuación, una vez analizados y presentados los conceptos de transformación y sincronización de modelos y la configuración de transformaciones, se presenta el diseño de una infraestructura aplicable a cualquier transformación de modelos, capaz de especificar las posibles opciones de configuración que el usuario crea necesarias.

3.3. Diseño de la infraestructura de configuración de transformaciones

Esta infraestructura se basa principalmente en la generación de *modelos de configuración* que se utilizarán como entrada para la transformación. Para poder crear estos modelos, nos hace falta un metamodelo para especificar esta configuración. A partir de este metamodelo se generan *modelos de catalógo de reglas*, encargados de especificar que patrones y reglas se pueden configurar. Una vez se hayan definido los modelos de configuración, a partir del catálogo de reglas y el modelo de entrada, se utilizarán como modelos de entrada en las transformaciones.

Como se ha comentado en el punto anterior, la configuración de transformaciones se basa en saber qué y cómo se tiene que transformar los elementos de un modelo.

- El qué, es lo que se ha definido como el patrón (*Pattern*). Este patrón deberá contener al elemento o elementos del modelo de entrada a configurar.

- El cómo, se ha definido como la regla (*Rule*). La regla o conjunto de reglas definen a qué elemento o elementos del modelo destino se transforma el patrón.

También, en algunas ocasiones se deberán configurar ciertas propiedades del modelo de salida, y para ello se ha definido el concepto de parámetro (*Parameter*).

A modo de ejemplo, utilizaremos el patrón de Generalization para crear un posible modelo de configuración. Así pues:

Cada vez que se encuentre una jerarquía de herencia sobre un modelo del diagrama de clases UML, se podrá elegir entre las distintas reglas:

Regla 1:

- Se crea una única tabla para la clase padre.
- Los atributos de las clases hijas aparecen como columnas de la clase padre.

Regla 2:

- Se generan tablas solo para las clases hijas.
- Los atributos de las clases hijas se convierten en campos en su correspondiente tabla.
- Los atributos de la clase padre se convierten en campos para las clases hijas.

Regla 3: (Regla por defecto)

- Se generan tablas tanto para la clase padre como para las clases hijas.
- Se incluye la correspondiente clave ajena (FK) de las tablas "hijas" a la tabla "padre".

Así pues para el patrón que denominaremos GeneralizationInto se podrán elegir las siguientes opciones:

- Patrón GeneralizationInto: OnlyParentTable/OnlyChildTable/AllTables

Regla 1	GeneralizationInto = OnlyParentTable
Regla 2	GeneralizationInto = OnlyChildTable
Regla 3	GeneralizationInto = AllTables

Modelo de Configuración para el patrón Generalization:

- Patrón de configuración
  - Patrón: *GeneralizationInto*
  - Reglas:
    - *AllTables*

- *OnlyParentTable*
- *OnlyChildTable*

- Parámetros (para el caso en el que alguna de las reglas genere una clave ajena):
  - *On Update*
  - *On Delete*

Además, para cada elemento del modelo de entrada se tendrá que saber que patrón de todos los que se han definido está cumpliendo. Para ello, el Patrón debe contener ciertas referencias a estos elementos y a sus propiedades.

Una vez presentada la estructura del modelo, solamente falta presentar como se define su metamodelo, al que hemos denominado Catálogo de reglas. Para crear dicho metamodelo, nos hemos basado en el apartado "2.3.1.1. Creación del Ecore", y para poder crear instancias del metamodelo, hemos tenido que crear su editor, siguiendo el apartado "2.3.1.2. Generación de código".

3.3.1. Catálogo de reglas

Como ya se ha indicado, este metamodelo contiene un patrón de configuración (*ConfigurationPattern*) para cada elemento o elementos (*Instance*) del metamodelo (*Metamodel*) referente al modelo de entrada que se quiere configurar y cada patrón de configuración a su vez, contiene las posibles reglas (*Rule*) de configuración y los parámetros (*Parameter*) que se quiera configurar.

De manera detallada:

Elemento	***Descripción***
Catalog	Elemento raíz del catálogo de reglas.
Metamodel	Representa tanto al metamodelo de entrada (relación *source* 0..1) como el de salida (relación *target* 0..1) en la transformación de modelos.
ConfigurationPattern	Representa el patrón de configuración para un elemento del metamodelo de entrada que se quiere configurar. Puede contener distintas Reglas (relación *rules* 1..*****) y parametros (relación *parameter* 0..*****), pero solamente un patrón configurado (relación *pattern* 1..1). Siempre deberá tener una regla por defecto (relación *defaultRule* 1..1), que será la que se ejecute en la transformación, en caso que el usuario no seleccione ninguna regla.
Rule	Representa la regla que se puede ejecutar para cierto patrón de configuración.

Parameter	Representa el parámetro que se quiere configurar para cierto patrón de configuración.
Pattern	Representa al patrón definido para un elemento o elementos del metamodelo de entrada que se quiere configurar. Por ello, contiene instancias (relación *elements* 1..*).
Instance	Representa el objeto del metamodelo de entrada a configurar (relación *type* al objeto EObject de Ecore).
InstanceParameter	Representa los posibles parámetros de una instancia.
ReferenceInstance Parameter	Representa las referencias entre instancias. Por ello contiene una relación *reference* a EObject y una relación *value* al elemento Instance.
AttributeInstance Parameter	Representa los atributos de una instancia. Por ello contiene una relación *attribute* a EObject y una propiedad *value* de tipo String en la cual se indica el valor del atributo.

Tabla 1. Elementos del metamodelo del catálogo de reglas

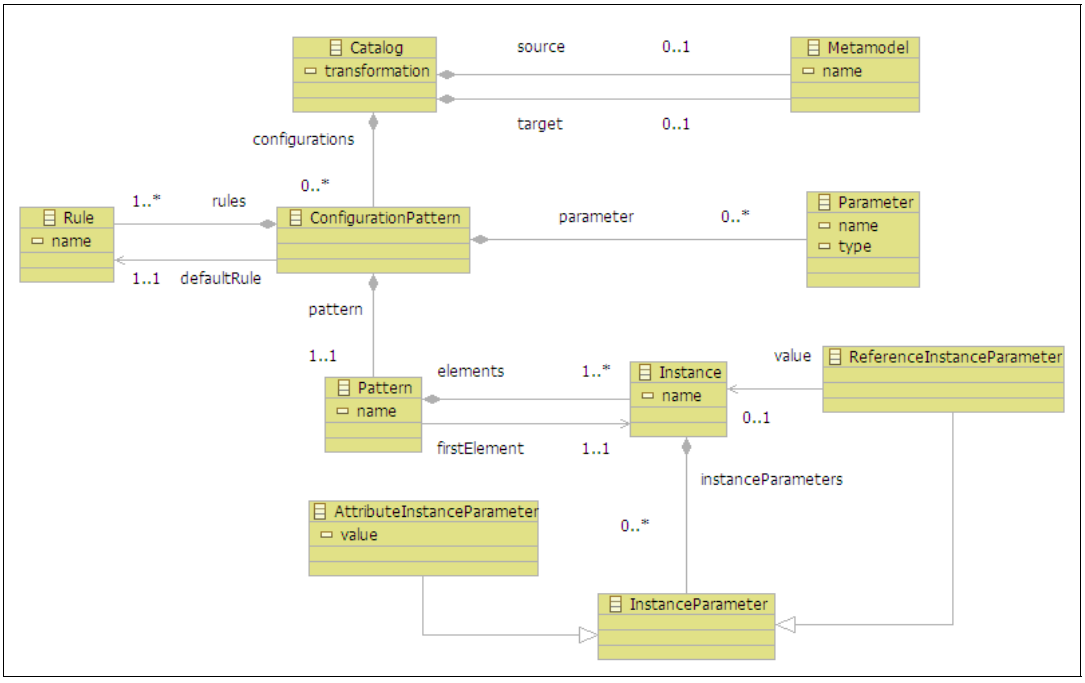


Figura 5. Metamodelo del catálogo de reglas

Así pues, a partir del metamodelo definido y haciendo uso de los requisitos a configurar para la transformación entre modelos de clases UML y modelos de Base de Datos, definidos en el capítulo siguiente, vamos a crear un modelo de catálogo de reglas en el que configuraremos la Generalization de UML. Para ello, primero se presenta la estructura típica de una generalización según la especificación de UML:

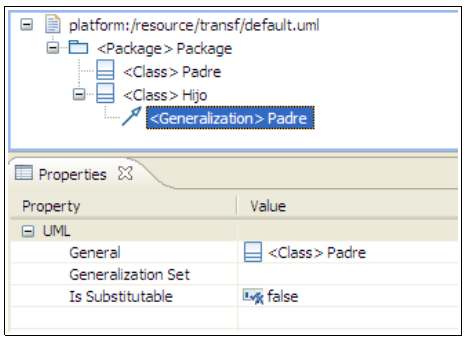


Figura 6. Estructura de la generalización en UML

Para el patrón del modelo se configuran sus instancias de acuerdo a la estructura del elemento Generalization del modelo UML:

- Las instancias (Instance) y sus parámetros (InstanceParameter)
  - *Instance specificClass* (type: *Class*)
    - *Reference Instance Parameter*
      - propiedad reference: *generalization : Generalization*
      - propiedad value: *Instance generalization*
  - *Instance generalClass* (type: *Class*)
  - *Instance generalization* (type: *Generalization*)
    - *Reference Instance Parameter*
      - propiedad reference: *general : Classifier*
      - propiedad value: *Instance generalClass*

Como podemos observar, las instancias del patrón *GeneralizationInto* tienen la misma estructura que la generalización de UML (figura):

- Class: Padre → Instance generalClass
- Class: Hijo → Instance specificClass
  - Generalization → Instance generalization, ReferenceInstanceParameter generalization
    - general: Class Padre→ ReferenceInstanceParameter general

Por otro lado, para el patrón de configuración (Configuration Pattern) indicamos:

- Las reglas (Rule)
  - *Rule OnlyParentTable*
  - *Rule OnlyChildTable*
  - *Rule AllTables*
- El patrón (Pattern)
  - *Pattern GeneralizationInto* (FirstElement: *Instance specificClass*)
- Los parámetros (Parameter)
  - *Parameter OnUpdate_generalization*
  - *Parameter OnDelete_generalization*

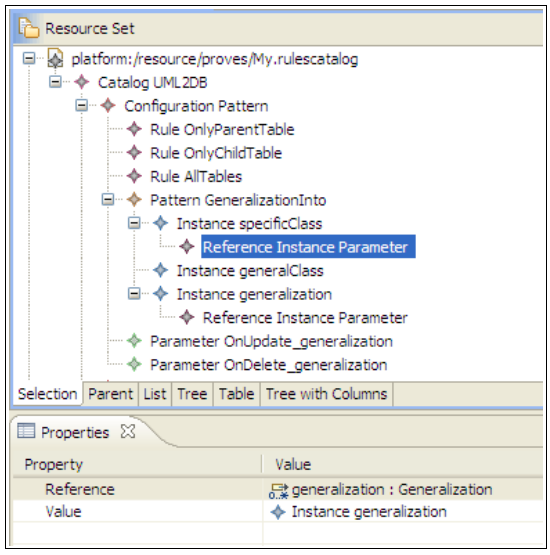


Figura 7. Modelo del catálogo de reglas

Por todo ello y siguiendo los requisitos de configuración para la transformación entre modelos de clases UML y modelos de Base de Datos podemos definir el correspondiente modelo de catálogo de reglas para cada uno de los patrones, reglas y parámetros especificados.

Pero el problema todavía no está resuelto, porque lo que se quiere alcanzar es un modelo de configuración específico para cada modelo de entrada, es decir, que contenga la configuración de dicho modelo. Este modelo de configuración se basa en el modelo de catálogo de reglas.

Por un lado, este modelo de configuración contiene los modelos (*Model*) de entrada y salida en la transformación y que son instancias de los metamodelos (*Metamodel*) del modelo

de catálogo de reglas.

Por otro, también contiene un patrón configurado (*ConfiguredPattern*) para cada uno de los elementos del modelo de entrada que cumplen algún patrón definido en el modelo de catálogo de reglas. Este patrón configurado contiene a su vez las instancias (*SpecificInstance*) de los elementos del modelo de entrada que cumplen el patrón del modelo de catálogo de reglas.

A continuación se presenta la estructura o metamodelo para el modelo de configuración.

3.3.2. Configuración de transformaciones

Así pues, el metamodelo de configuración contiene un patrón configurado (*ConfiguredPattern*) para cada uno de los elementos del modelo (*Model*) de entrada. Además, cada patrón configurado contiene las diferentes reglas (*Rule*) seleccionables por parte del usuario en el momento de la transformación y algunos parámetros configurados (*ParameterValue*) necesarios para el modelo de salida.

Elemento	***Descripción***
Configuration	Elemento raíz del metamodelo de configuración de transformaciones.
Model	Representa tanto al modelo de entrada (relación *source* 0..1) como el de salida (relación *target* 0..1) en la transformación. Además, tiene una referencia metamodel al elemento **Metamodel** del catálogo de reglas para saber a que metamodelo pertenece.
ConfiguredPattern	Representa cada uno de los patrones configurados que se han definido en el catálogo de reglas (**ConfigurationPattern**). Contiene una referencia (relación *pattern* 1..1) a las instancias **Pattern** del modelo del catálogo y una referencia (relación *rule* 1..1) a las instancias **Rule** del catálogo de reglas. También contiene una referencia (relación *defaultRule* 1..1) a cualquier instancia **Rule** que indicará la regla a ejecutar "por defecto" en caso de que el usuario no haya seleccionado ninguna durante la configuración y una referencia (relación *firstElement* 1..1) que contiene el elemento del modelo de entrada que cumple el patrón.
ParameterValue	Representa el valor que el usuario asignará a los parámetros (relación *parameter* 1..1) que son instancias **Parameter** del catálogo de reglas.
SpecificInstance	Representa cada uno de los elementos que cumplen el patrón. Para ello contiene una relación con la instancia del modelo de entrada (relación *element* a EObject) y una referencia (relación *instance* 1..1) con el elemento Instance del catálogo de reglas.

Tabla 2. Elementos del metamodelo de configuración de transformaciones

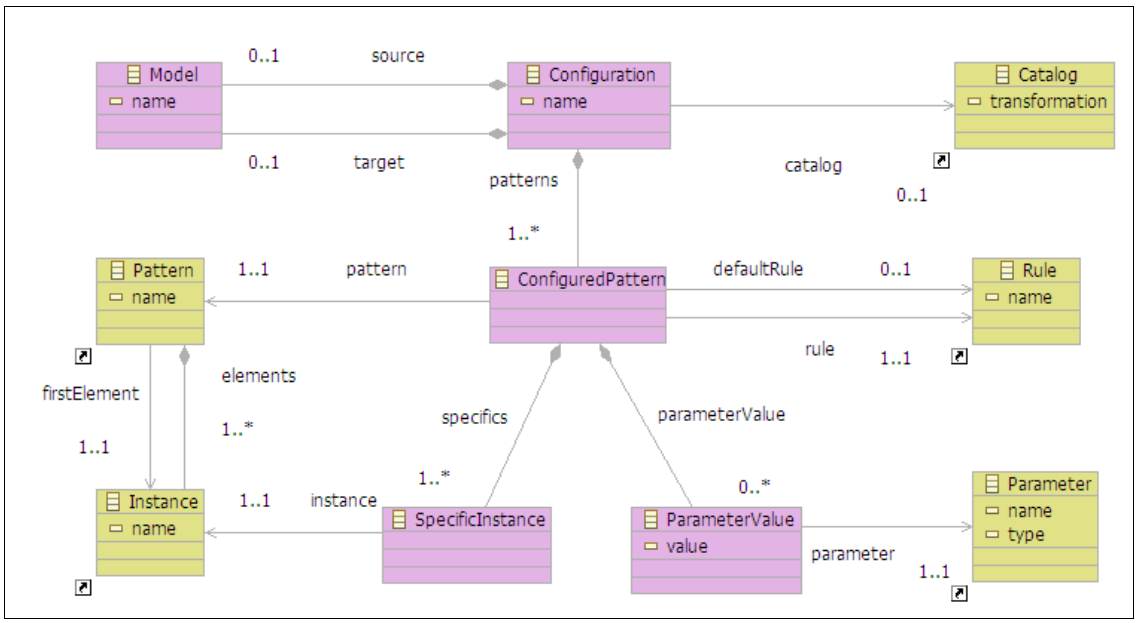


Figura 8. Metamodelo de configuración

Finalmente, y siguiendo con el ejemplo para configurar la Generalización, ahora ya se puede generar el modelo de configuración correspondiente. Para generar este modelo, se lanza una transformación cuyos modelos de entrada son: el modelo del catálogo de reglas y el modelo UML (los detalles de la transformación se presentan en el siguiente capítulo).

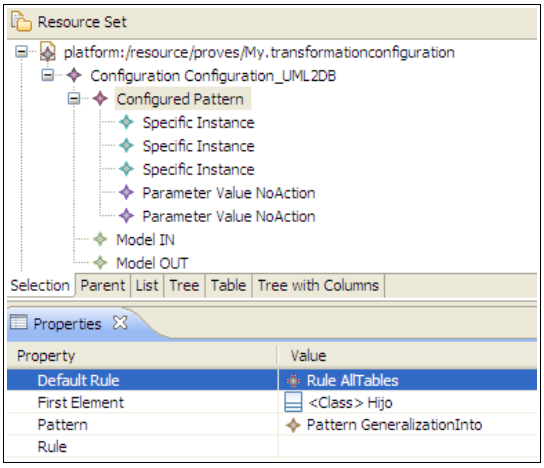


Figura 9. Modelo de configuración

Así pues, para el patrón configurado (*ConfiguredPattern*) se ha especificado:

- *Configured Pattern*
  - Propiedad Default Rule: *Rule AllTables*
  - Propiedad First Element: *Class Hijo*
  - Propiedad Pattern: *Pattern GeneralizationInto*
  - Propiedad Rule: *A seleccionar por parte del usuario*

- *Specific Instance*
  - Propiedad Element: *Class Hijo*
  - Propiedad Instance: *Instance specificClass*
- *Specific Instance*
  - Propiedad Element: *Generalization*
  - Propiedad Instance: *Instance generalization*
- *Specific Instance*
  - Propiedad Element: *Class Padre*
  - Propiedad Instance: *Instance generalClass*
- *Parameter Value*
  - Propiedad Parameter: *Parameter OnUpdate_generalization*
  - Propiedad Value: *A indicar por parte del usuario*
- *Parameter Value*
  - Propiedad Parameter: *Parameter OnDelete_generalization*
  - Propiedad Value: *A indicar por parte del usuario*

3.4. Conclusiones

Finalmente y como conclusión, en este capítulo se han presentado los conceptos de transformación y configuración de transformaciones sobre los que se basa este trabajo y a continuación, se ha detallado el diseño de la infraestructura para configurar transformaciones con un pequeño ejemplo para entender mejor la propuesta.

Capítulo 4 – Implementación de transformaciones ATL. Del diagrama de clases UML al modelo de Base de Datos

En este capítulo, tras analizar y diseñar la infraestructura para dar soporte a la configuración de transformaciones ya solamente nos queda implementar las transformaciones dando soporte a la configuración.

Primeramente, presentaremos los requisitos o criterios de configuración necesarios para el caso concreto de las transformaciones de UML a Base de Datos. Seguidamente y de manera general se comentan los pasos a seguir para llevar a cabo el proceso de configuración de transformaciones. A continuación, lo aplicaremos al caso concreto para las transformaciones de UML a Base de Datos y, siguiendo la estructura de módulos de la herramienta MOSKitt, entraremos en detalle en cada una de las transformaciones *ATL* requeridas para dar soporte a la configuración, transformación y sincronización.

A continuación, se especifican los requisitos de configuración de la CIT para el caso de la transformación entre modelos UML y modelos de Base de Datos.

4.1. Requisitos para la configuración de transformaciones UML a Base de Datos

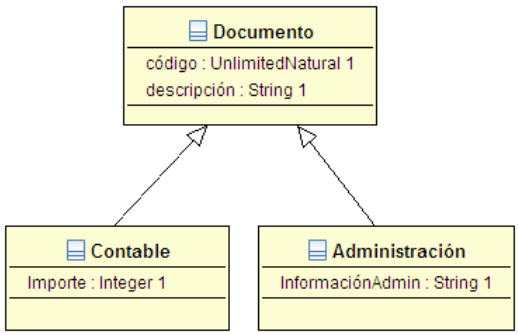
Tal y como se ha comentado, el objetivo de la configuración de las transformaciones es indicar en algún momento de la transformación, qué y cómo un elemento, o conjunto de elementos, del modelo de entrada se transforma en el modelo de salida. Así pues, hay que tener en cuenta 2 conceptos básicos para definir la configuración:

- Qué
    - Dentro de la configuración, este concepto se denomina Patrón (*Pattern*)
- Cómo
    - Este concepto se denomina Regla (*Rule*). Dependiendo de estas reglas se ejecutará una regla u otra en las transformaciones ATL.

Además, y para darle más funcionalidad a la configuración, se pretende añadir algunos parámetros para configurar ciertas propiedades de los elementos del modelo de salida.

A continuación presentamos cada uno de los patrones y parámetros a configurar, y para cada patrón, las posibles reglas a ejecutar:

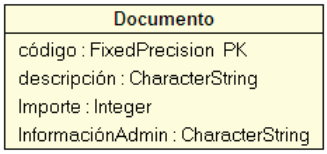
4.3.1. Patrón Generalization



Se definen las siguientes opciones para transformar una jerarquía de herencia definida en un diagrama de clases (DCL):

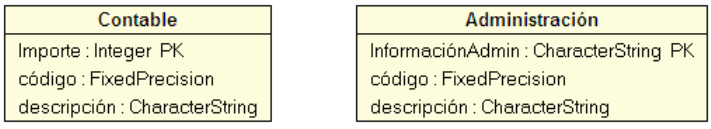
Regla 1:

- Se crea una única tabla para la clase padre.
- Los atributos de las clases hijas aparecen como columnas de la clase padre.



Regla 2:

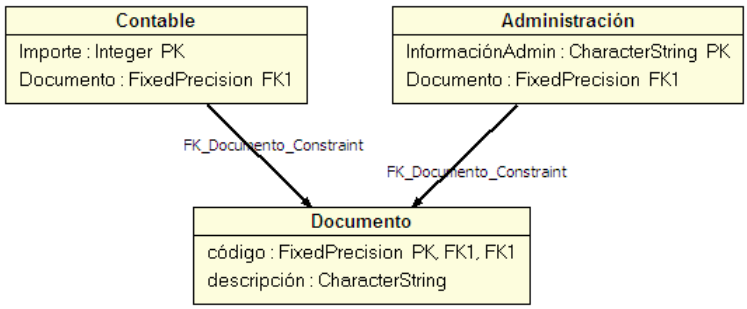
- Se generan tablas solo para las clases hijas.
- Los atributos de las clases hijas se convierten en campos en su correspondiente tabla.
- Los atributos de la clase padre se convierten en campos para las clases hijas.



Validaciones: No es aplicable cuando las clases hijas son clases abstractas.

Regla 3: (Regla por defecto)

- Se generan tablas tanto para la clase padre como para las clases hijas.
- Se incluye la correspondiente clave ajena (FK) de las tablas "hijas" a la tabla "padre".



Validaciones: No es aplicable cuando la clase padre es una clase abstracta.

Así pues para el patrón que denominaremos GeneralizationInto se podrán elegir las siguientes opciones:

- Patrón GeneralizationInto: OnlyParentTable/OnlyChildTable/AllTables

Regla 1	GeneralizationInto = OnlyParentTable
Regla 2	GeneralizationInto = OnlyChildTable
Regla 3	GeneralizationInto = AllTables

3.3.2. Patrón Association

Se definen las siguientes opciones para transformar una asociación definida en un DCL:

Regla 1:

Siempre que en una asociación alguno de los extremos de la asociación tenga la Propiedad UpperBound > 1 se crea una tabla para representar la asociación.

- La clave primaria (PK) de esta tabla estará compuesta por las claves primarias de las tablas en las que se han transformado las clases extremo de la asociación y sus correspondientes claves ajenas (FK).

Src/dst	UpperBound = 1	UpperBound > 1
UpperBound = 1	Una referencia por cada navegabilidad definida en la asociación.	Tabla
UpperBound > 1	Tabla	Tabla

Regla 2: (Regla por defecto)

Sólo se creará una tabla intermedia para representar la asociación en el caso de que ambos extremos de la asociación tengan en su propiedad UpperBound > 1.

Src/dst	UpperBound = 1	UpperBound > 1

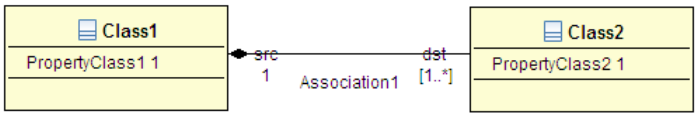
UpperBound = 1	Una referencia por cada navegabilidad definida en la asociación.	Una FK hacia la tabla cuya clase aparece en el extremo de la asociación cuya propiedad UpperBound = 1
UpperBound > 1	Una FK hacia la tabla cuya clase aparece en el extremo de la asociación cuya propiedad UpperBound = 1	Tabla

La estrategia a adoptar se podría definir haciendo uso del siguiente patrón de configuración:

- Patrón AssociationInto: Table/Reference

Regla 1	AssociationInto = Table
Regla 2	AssociationInto = Reference

3.3.3. Patrón Aggregation

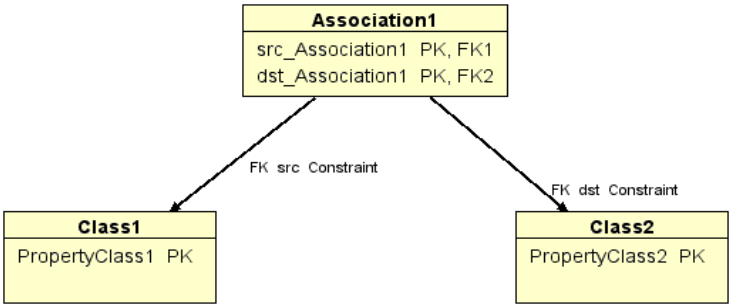


Se definen las siguientes opciones para transformar una agregación/composición definida en un DCL:

Regla 1:

- Se trata como cualquier asociación teniendo en cuenta la regla 1 (Table).

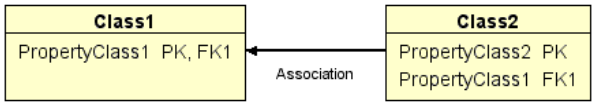
Ejemplo: Teniendo en cuenta la agregación/composición del DCL del ejemplo, si AssociationAs = Table, tendríamos:



Regla 2:

- Se trata como cualquier asociación teniendo en cuenta la regla 2 (Reference).

Ejemplo: Teniendo en cuenta el DCL del ejemplo, si AssociationAs = Reference, tendríamos:



Regla 3: (Regla por defecto)

- Se interpreta como una restricción de identificación en el Modelo de Base de Datos, es decir, la clave ajena es también clave primaria.

Ejemplo: Teniendo en cuenta el DCL del ejemplo, tendríamos:

La estrategia a adoptar se podría definir haciendo uso del siguiente patrón de configuración:

- Patrón CompositionAs: Table/Reference/IntegrityConstaint

Regla 1	CompositionAs = Table
Regla 2	CompositionAs = Reference
Regla 3	CompositionAs = IntegrityConstraint

3.3.4. Patrón DataType

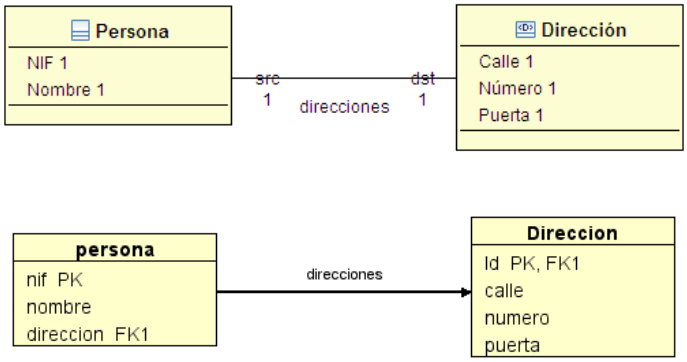
Se definen las siguientes opciones para transformar un elemento DataType definido en un DCL:

Regla 1: (Regla por defecto)

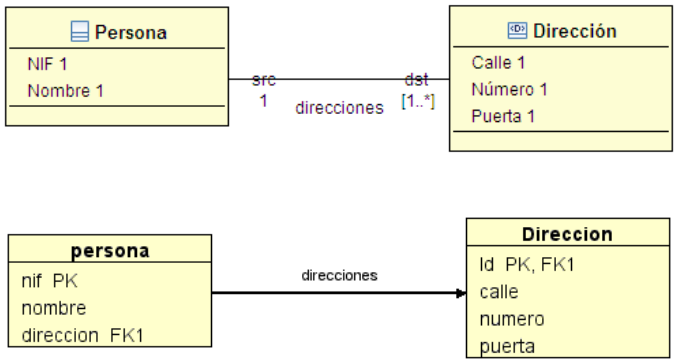
- La transformación es la misma que si se tratase de una clase.

Ejemplo 1:

Ejemplo 2:



Ejemplo 3:



Regla 2:

Sólo se transformará en una tabla en los siguientes casos:

- Cuando se haya definido alguna asociación entre el elemento DataType y un elemento Class en la que la propiedad UpperBound en el extremo de la asociación correspondiente al DataType sea > 1.

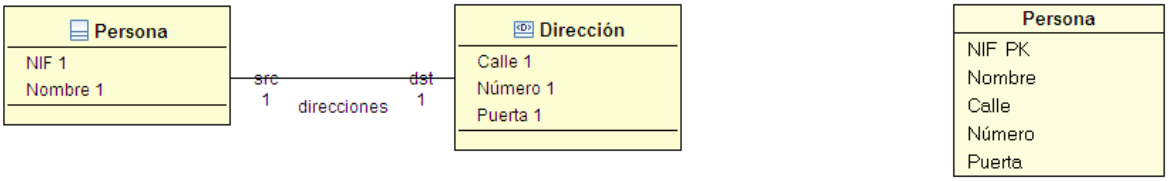
- Cuando se trate del atributo 'type' de un elemento Property, como propiedad de un elemento Class, con UpperBound > 1 (atributo objeto multivaluado).

Los atributos del elemento DataType migrarán a la clase asociada durante la transformación:

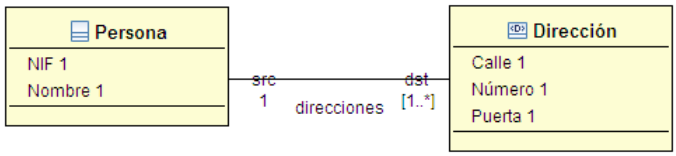
- Cuando se haya definido alguna asociación entre el elemento DataType y un elemento Class en la que la propiedad UpperBound en el extremo de la asociación correspondiente al DataType sea = 1.

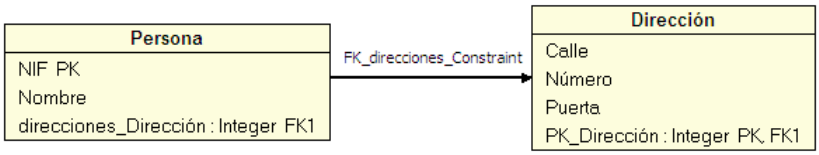
- Cuando se trate del atributo 'type' de un elemento Property, como propiedad de un elemento Class, con UpperBound = 1 (atributo objeto valuado).

Ejemplo 1:



Ejemplo 2:





La estrategia a adoptar se podría definir haciendo uso del siguiente patrón de configuración:

- Patrón DataTypeAs: Table/Property

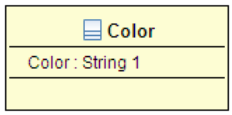
Regla 1	DataTypeAs = Table
Regla 2	DataTypeAs = Property

3.3.5. Patrón Enumeration

Se definen las siguientes opciones para transformar un elemento Enumeration definido en un DCL:

Regla 1:

- La transformación es la misma que si se tratase de una clase con una propiedad del mismo nombre que la Enumeración.



Consideraciones:

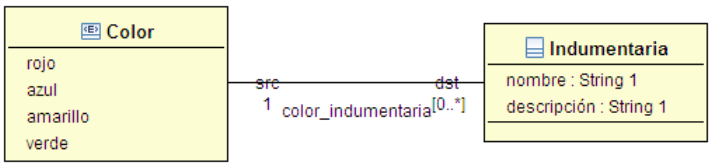
- Se incluirá una Check Constraint en el elemento PersistentTable resultante para indicar que los valores posibles para el elemento Column son los correspondientes a los Enumeration Literals.

- Se incluirá en la configuración una nueva propiedad para configurar el valor de la propiedad DataType de la columna resultante en el modelo destino (MBD). Se trata de la propiedad **ColumnDataType**:

  - El valor de la propiedad DataType del elemento Column resultante será *CharacterString* para el caso de que el parámetro de configuración ColumnDataType de la Enumeration no tenga valor.

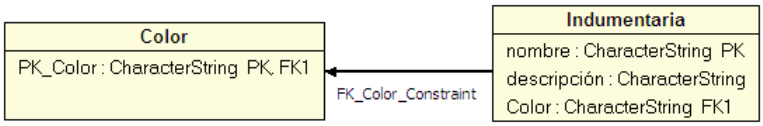
  - El valor de la propiedad DataType del elemento Column resultante será el indicado por el parámetro de configuración ColumnDataType cuando éste tenga valor.

    Ejemplo 1:



    Ejemplo 2:

En ambos casos la transformación será la misma:



    Regla 2: (Regla por defecto)

Sólo se transformará siguiendo la regla 1 en los siguientes casos:

- Cuando se haya definido alguna asociación entre el elemento Enumeration y una clase en la que la propiedad UpperBound en el extremo de la asociación correpondiente a elemento Enumeration sea > 1.

- Cuando se trate de la propiedad Type de un elemento Property con UpperBound>1 (atributo objeto multivaluado).

Consideraciones:

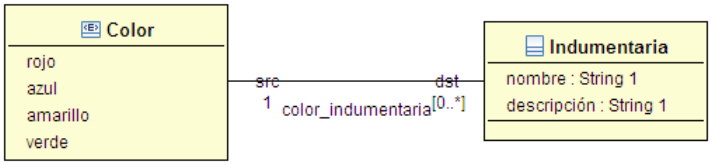
- La tabla que contiene al campo resultado de la transformación del elemento Property cuya propiedad Type era un enumerado, tendrá asociada una Check Constraint para indicar que los elementos Enumeration Literal del elemento Enumeration son los únicos valores posibles para ese campo.

- Se incluirá en la configuración una nueva propiedad para configurar el valor de la propiedad DataType de la columna resultante en el modelo destino (BD). Se trata de la propiedad **ColumnDataType**:

  - El valor de la propiedad DataType del elemento Column resultante será *CharacterString* para el caso de que el parámetro de configuración ColumnDataType de la Enumeration no tenga valor.

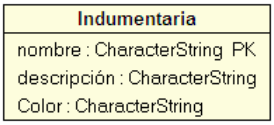
  - El valor de la propiedad DataType del elemento Column resultante será el indicado por el parámetro de configuración **ColumnDataType** cuando éste tenga valor.

    Ejemplo 1:



    Ejemplo 2:

    En ambos casos la transformación será la misma:



La estrategia a adoptar se podría definir haciendo uso del siguiente patrón de configuración:

- Patrón EnumerationAs: Table/Property

Regla 1	EnumerationAs = Table
Regla 2	EnumerationAs = Property

3.3.6. Propiedad Foreign Key

Los siguientes parámetros de configuración se tendrán en cuenta siempre que una regla de transformación de lugar a una clave ajena (Foreign Key) en el modelo de salida. Tienen como objetivo mantener la integridad referencial en el modelo de Base de Datos:

Se permitirá que sea el usuario el que decida qué tipo de regla se deberá seguir en el modelo de Base de Datos para mantener la Integridad Referencial en sus claves ajenas en los casos siguientes:

- Cada vez que se modifique el valor de una clave primaria (Primary Key) referenciada por una clave ajena (Foreign Key) de otra tabla relacionada.

- Cada vez que se elimine un valor en una clave primaria (Primary Key) referenciada por una clave ajena (Foreign Key) de otra tabla relacionada.

La estrategia a adoptar se podría definir haciendo uso de los siguientes parámetros de configuración:

- Parámetro OnUpdate = NoAction/Cascade/Restrict/SetNull/SetDefault

- Parámetro OnDelete = NoAction/Cascade/Restrict/SetNull/SetDefault

Posibles valores	OnUpdate = NoAction/Cascade/Restrict/SetNull/SetDefault ONDelete = NoAction/Cascade/Restrict/SetNull/SetDefault

4.2. Configuración de transformaciones

A continuación, podemos observar el proceso para transformar modelos aplicando configuración por parte del usuario. En la figura 10 se observa como, tras definir un modelo del catálogo de reglas y crear un modelo concreto, el usuario es quien configura la transformación, en el *wizard de configuración*, para posteriormente lanzar la transformación y mantener los modelos sincronizados.

Así pues, los pasos a seguir para llevar a cabo este proceso de transformaciones son los siguientes:

1. Creación de la transformación entre el catálogo de reglas y la configuración de transformaciones.

2. Creación de la transformación entre el modelo de configuración, el modelo de entrada y el modelo de salida.

3. Creación de la transformación para mantener sincronizados los modelos de entrada y salida.

1. Creación de la transformación para obtener el modelo de diferencias.

2. Creación de la transformación para sincronizar el modelo de salida con el modelo de diferencias.

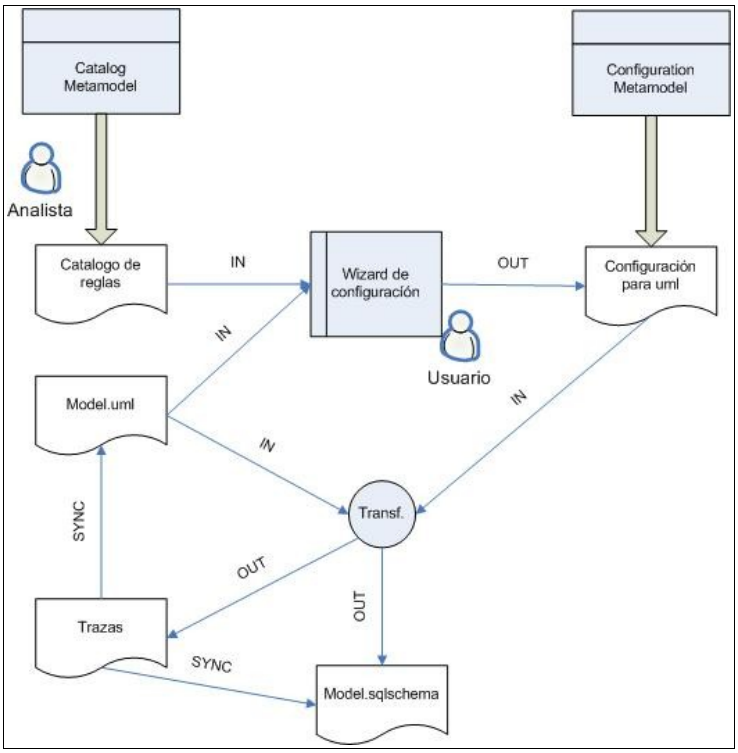


Figura 10. Visión general del proceso de configuración de transformaciones

Tal y como se comentó en la introducción, la herramienta MOSKitt está compuesta por módulos, cada uno de ellos dedicados a una función concreta. Para el caso de las transformaciones entre modelos UML y modelos de Base de Datos, los módulos o proyectos donde se encuentra dicha información son:

- *Proyecto de transformación (es.cv.gvcase.linkers.uml2db.transf)*
    - Dentro de este proyecto, cuyo diseño aparece detallado en el anexo 1 de este documento, se pueden observar 2 transformaciones y una librería de helpers:
        - Transformación de modelos del catálogo de reglas a modelos de configuración de transformaciones (catalog2configuration.atl)
        - Transformación de modelos UML2 a modelos Base de Datos (uml2db.atl)
        - Librería para la transformación (uml2db_library.atl)
- *Proyecto de sincronización (es.cv.gvcase.linkers.uml2db.sync)*
    - Dentro de este módulo, cuyo diseño aparece detallado en el anexo 2, también aparecen 2 transformaciones y una librería de helpers:

- Transformación para obtener el cálculo de diferencias (DiffCalc.atl)
- Transformación para obtener el modelo destino con la combinación de diferencias (DiffMerge.atl)
- Librería para la sincronización (DiffMerge_library.atl)

Antes de presentar cada una de las transformaciones, comentaremos como se ha estructurado cada una de las mismas con un pequeño ejemplo de la transformación del diagrama de clases a Base de Datos:

- **Cabecera** – contiene los modelos y metamodelos de origen y destino.
  - **Entradas** – IN : UML, modelConf : CONF
  - **Salidas** – OUT : DB, trace : Trace

```
create OUT : DB, trace : Trace from IN : UML, modelConf : CONF;
```

- **Librerías** – contiene los helpers. En este caso aparecen en otro fichero distinto para estructurar mejor la información.

```
uses uml2db_library;
```

- **Variables** – se definen variables auxiliares utilizadas en las regals. Como ejemplo se muestran la variable *database* que almacena el elemento Database y una lista que almacena las claves ajenas *Fkseq*.

```
...
helper def: database : SQLMODEL!Database =
  OclUndefined;

helper def: FKseq : Sequence(UML!ForeignKey) =
  Sequence {};
...
```

- **Helpers** – aparecen en el fichero auxiliar uml2db_library.atl. El helper que se muestra sirve para comprobar si un elemento Property de UML es multivaluado, es decir, su cardinalidad máxima es mayor que uno.

```
...
helper context UML!Property def: isMultivaluated() : Boolean =
  if self.upperValue <> OclUndefined then
```

```
    self.upperValue.value <> 1
  else
    false
  endif;
...
```

- **Reglas anónimas** – son las reglas que se ejecutan cuando se encuentra un elemento en el modelo origen, que cumple cierta condición y se transforma en otro u otros elementos del modelo destino.

  En este caso, esta regla se ejecuta cuando se encuentra un elemento Class de UML y cumple la condición de: ser la clase padre en una generalización (pertenece al patrón configurado *GeneralizationInto*) y la regla de configuración elegida por el usuario para este patrón es la de *OnlyParentTable*.

  Entonces, el elemento Class se transforma en un elemento PersistentTable y otro elemento PrimaryKey. Además se generan las trazas correspondientes (elementos TraceLink, TraceLinkEnd y ElementRef) para poder manterer sincronizados los modelos origen y destino.

```
...
rule GeneralizationParent2Table {
  from
    i : UML!Class (
          i.isAbstractClass() = false and
          i.oclIsTypeOf(UML!Class) = true and
          i.isPersistentClass() = true and
          i.executeParentTableRule('OnlyParentTable') = true
    )
  to
    ot : DB!PersistentTable (
      name <- i.name
    ),
    ocpk : DB!PrimaryKey (
      name <- 'PK_' + i.name + '_Constraint'
    ),
    __traceLink : Trace!TraceLink (
      name <- 'GeneralizationParent2Table',
      sourceElements <- Sequence {__LinkEnd_i},
      targetElements <- Sequence {__LinkEnd_ot, __LinkEnd_ocpk},
      model <- thisModule.__wmodel
    ),
    __LinkEnd_i : Trace!TraceLinkEnd (
      element <- __elementRef_i
    ),
    __elementRef_i : Trace!ElementRef (
      links <- __traceLink,
      modelRef <- thisModule.__model_IN
    ),
    __LinkEnd_ot : Trace!TraceLinkEnd (
      element <- __elementRef_ot
```

```
    ),
    __elementRef_ot : Trace!ElementRef (
      links <- __traceLink,
      modelRef <- thisModule.__model_OUT
    ),
    __LinkEnd_ocpk : Trace!TraceLinkEnd (
      element <- __elementRef_ocpk
    ),
    __elementRef_ocpk : Trace!ElementRef (
      links <- __traceLink,
      modelRef <- thisModule.__model_OUT
    )
  do {
    if(i.getPKCandidate() <> OclUndefined)
      ocpk.members <-
ocpk.members.append(thisModule.resolveTemp(i.getPKCandidate(), 'gc'));
    else
      thisModule.createPK(i, ocpk);

    thisModule.resolveTemp(i.eContainer(), 'os').tables <-
    thisModule.resolveTemp(i.eContainer(), 'os').tables.append(ot);
    thisModule.resolveTemp(i, 'ot').constraints <-
    thisModule.resolveTemp(i, 'ot').constraints.append(ocpk);

    __elementRef_i.ref <- i;
    __elementRef_ot.ref <- ot;
    __elementRef_ocpk.ref <- ocpk;
  }
}
...
```

- **Reglas nombradas** - estas reglas se ejecutan solamente cuando son llamadas por otras. A estas reglas se les puede pasar parámetros y no tienen cláusula from.

  En este caso, esta regla es la que se ejecuta cuando se crea un elemento PrimaryKey para alguna tabla y hay que crear su correspondiente elemento Column.

```
rule createPK(elem : UML!Element, pk : DB!PrimaryKey) {
  to
    oac : DB!Column (
      name <- 'PK_' + elem.name
    )
  do {
    thisModule.Integer2IntegerDataType(oac);
    thisModule.resolveTemp(elem, 'ot').columns <-
                  thisModule.resolveTemp(elem,
'ot').columns.append(oac);
    pk.members <- pk.members.append(oac);
  }
}
```

A continuación pasamos a presentar cada una de las transformaciones. El código ATL no ha sido añadido por cuestiones de legibilidad y extensión del mismo. Se ha detallado cada uno de los apartados de las transformaciones tal y como hemos indicado anteriormente. Si es necesario acceder al código, éste está disponible en la página web del proyecto MOSKitt (http://www.moskitt.org/cas/moskitt/), en el apartado de *descargas*.

4.2.1. Transformación de catálogo de reglas a la configuración de transformaciones

De acuerdo al diseño presentado en el capítulo anterior, esta transformación se basa en generar un modelo de configuración a partir de un modelo (*catálogo de reglas)* donde se especifican los parámetros y las reglas para un metamodelo específico (el metamodelo de UML) y un modelo cualquiera de un diagrama de clases UML. Este modelo generado, es el que se utilizará posteriormente para configurar la transformación entre el diagrama de clases UML y el modelo de Base de Datos.

- Cabecera:
  - Entradas:
    - Metamodelo UML2
    - Modelo de entrada UML2
    - Metamodelo del catálogo de reglas
    - Modelo de entrada del catálogo de reglas
  - Salidas:
    - Metamodelo de Configuración de transformaciones
    - Modelo de salida de Configuración de transformaciones
- Variables:
  - src: Elemento auxiliar Property del modelo UML2
  - added: variable booleana.
- Helpers:
  - isTransformPattern
    - Tipo: ConfigurationPattern

- Valor de retorno: Boolean
- Descripción: Comprueba si el patrón configurado ConfigurationPattern es *TransformElement.*

- isPersistentClassPattern
  - Tipo: ConfigurationPattern
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el patrón configurado ConfigurationPattern es *Persistent* y contiene un elemento Class de UML.
- isPersistentPropertyPattern
  - Tipo: ConfigurationPattern
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el patrón configurado ConfigurationPattern es *Persistent* y contiene un elemento Property de UML.
- isPersistentAssociationPattern
  - Tipo: ConfigurationPattern
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el patrón configurado ConfigurationPattern es *Persistent* y contiene un elemento Association de UML.
- isGeneralizationPattern
  - Tipo: ConfigurationPattern
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el patrón configurado ConfigurationPattern es *GeneralizationInto* y los elementos contenidos cumplen el patrón de Generalización de UML.
- isAssociationPattern
  - Tipo: ConfigurationPattern
  - Valor de retorno: Boolean

        - Descripción: Comprueba si el patrón configurado ConfigurationPattern es *AssociationInto* y los elementos contenidos cumplen el patrón de Asociación de UML.

    - isCompositionPattern

        - Tipo: ConfigurationPattern

        - Valor de retorno: Boolean

        - Descripción: Comprueba si el patrón configurado ConfigurationPattern es *CompositionAs* y los elementos contenidos cumplen el patrón de Agregación/Composición de UML.

    - isDataTypePattern

        - Tipo: ConfigurationPattern

        - Valor de retorno: Boolean

        - Descripción: Comprueba si el patrón configurado ConfigurationPattern es *DataTypeAs* y los elementos contenidos cumplen el patrón Tipo de Dato de UML.

    - isEnumerationPattern

        - Tipo: ConfigurationPattern

        - Valor de retorno: Boolean

        - Descripción: Comprueba si el patrón configurado ConfigurationPattern es Enumeration*As* y los elementos contenidos cumplen el patrón Enumeración de UML.

- Reglas anónimas:

    - Catalog2Configuration

        - Parte declarativa: crea el elemento Configuration a partir del elemento Catalog.

        - Parte procedural: asigna a la referencia *catalog* del elemento Configuration el elemento de entrada Catalog.

    - Metamodel2Model

        - Parte declarativa: crea el elemento Model a partir del elemento Metamodel.

        - Parte procedural: asigna a la referencia *metamodel* del elemento Model el

elemento de entrada Metamodel y lo situa dentro del elemento Configuration correspondiente.

- ConfigurationPattern2ConfiguredPattern
  - Parte procedural: a partir del elemento ConfigurationPattern comprueba si coincide con alguno de los patrones definidos y si es así llama a la regla correspondiente.

- Reglas nombradas:
  - TransformPatternElements
    - Parte declarativa: crea los elementos ConfiguredPattern y SpecificInstance.
    - Parte procedural: da valor a las propiedades de los elementos ConfiguredPattern y SpecificInstance a partir del elemento ConfigurationPattern y el elemento Element de UML2 que recibe como parámetros.
  - GeneralizationPatternElements
    - Parte declarativa: crea los elementos ConfiguredPattern y SpecificInstance.
    - Parte procedural: da valor a las propiedades de los elementos ConfiguredPattern y SpecificInstance a partir del elemento ConfigurationPattern y los elementos de UML2 (Class, Generalization, Class) que recibe como parámetros. Si existen parametros configurados llama a la regla generateParameterValue para asignarles valor.
  - AssociationPatternElements
    - Parte declarativa: crea los elementos ConfiguredPattern y SpecificInstance.
    - Parte procedural: da valor a las propiedades de los elementos ConfiguredPattern y SpecificInstance a partir del elemento ConfigurationPattern y los elementos de UML2 (Association, Property, Property) que recibe como parámetros. Si existen parametros configurados llama a la regla generateParameterValue para asignarles valor.
  - CompositionPatternElements
    - Parte declarativa: crea los elementos ConfiguredPattern y SpecificInstance.
    - Parte procedural: da valor a las propiedades de los elementos ConfiguredPattern y SpecificInstance a partir del elemento ConfigurationPattern y los elementos de UML2 (Association, Property, Property) que recibe como parámetros. Si existen parametros configurados llama a la regla generateParameterValue para

asignarles valor.

- DataTypePatternElements
  - Parte declarativa: crea los elementos ConfiguredPattern y SpecificInstance.
  - Parte procedural: da valor a las propiedades de los elementos ConfiguredPattern y SpecificInstance a partir del elemento ConfigurationPattern y el elemento de UML2 (DataType) que recibe como parámetros. Si existen parametros configurados llama a la regla generateParameterValue para asignarles valor.
- EnumerationPatternElements
  - Parte declarativa: crea los elementos ConfiguredPattern y SpecificInstance.
  - Parte procedural: da valor a las propiedades de los elementos ConfiguredPattern y SpecificInstance a partir del elemento ConfigurationPattern y el elemento de UML2 (Enumeration) que recibe como parámetros. Si existen parametros configurados llama a la regla generateParameterValue para asignarles valor.
- generateParameterValue
  - Parte declarativa: crea el elemento ParameterValue.
  - Parte procedural: da valor a las propiedades del elemento ParameterValue a partir de los elementos ConfigurationPattern y Parameter que recibe como parámetros.

4.2.2. Transformación del diagrama de clases UML a Base de Datos

Esta apartado, como se indica en el título, presenta la transformación del diagrama de clases UML a modelos de Base de Datos. A partir de un modelo UML y un modelo de configuración de acuerdo a este modelo UML, se genera un modelo de Base de Datos y un modelo de Trazas para mantener sincronizados ambos modelos.

- Cabecera:
  - Entradas:
    - Metamodelo UML2
    - Modelo de entrada UML2
    - Metamodelo de Configuración de transformaciones
    - Modelo de entrada de Configuración de transformaciones

- Salidas:
  - Metamodelo de Datatools
  - Modelo de base de datos conforme al metamodelo Datatools
  - Metamodelo de Trazas
  - Modelo de trazas conforme al metamodelo de trazas
- Librería:
  - Enlace a la librería uml2db_library.atl

- Variables:
  - __wmodel: Elemento raiz del modelo de trazas
  - __model_IN: Elemento que representa el modelo de entrada en el modelo de trazas
  - __model_OUT: Elemento que representa el modelo de salida en el modelo de trazas
  - database: Elemento raíz del modelo datatools
  - Fkseq: colección de claves ajenas (Foreign Keys) del modelo empleadas al final de la transformación para asignarles el tipo correcto de datos a sus miembros.
  - associationsOneToOne: colección de elementos Association con cardinalidad uno a uno utilizada en las reglas de transformación de dicho elemento.

- Regla Inicial:
  - initTransformation
    - Parte declarativa: crea el elemento raíz y los elementos que actúan como modelos entrada/salida del modelo de trazas.
    - Parte procedural: asigna los valores a las variables globales correspondientes.

- Regla Final:
  - GenerateFKTypes
    - Parte procedural: genera los tipos de datos correctos para las claves ajenas (Foreign Keys) generadas en la transformación.

- Reglas anónimas:
  - Model2Database
    - Parte declarativa: crea el elemento Database a partir del elemento Model y genera las trazas asociadas.
    - Parte procedural: asigna el nuevo elemento a la variable database. Si el elemento Model contiene classifiers, se genera un elemento Schema.
  - Model2DatabaseSchema
    - Parte declarativa: crea el elemento Database y el elemento Schema a partir del elemento Model y genera las trazas asociadas.
    - Parte procedural: asigna el nuevo elemento a la variable database y añade el elemento Schema creado al elemento Database.
  - Package2Schema
    - Parte declarativa: crea el elemento Schema a partir del elemento Package y genera las trazas asociadas.
    - Parte procedural: Añade el elemento Schema creado al elemento Database.
  - Class2Table
    - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Class (si NO cumple ningún patrón configurado para la Generalización) y genera las trazas asociadas.
    - Parte procedural: Intenta obtener una propiedad válida que actue como clave primaria, si no hay ninguna genera un elemento Column para ello.
  - DataType2Table
    - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento DataType (si cumple el patrón configurado *DataTypeAs = Table*) y genera las trazas asociadas.
    - Parte procedural: Genera un elemento Column para que actue como clave primaria.
  - Enumeration2Table
    - Parte declarativa: crea los elementos PersistentTable, PrimaryKey, Column, CheckConstraint y SearchConditionDefault a partir del elemento Enumeration (si cumple el patrón configurado Enumeration*As = Table*) y genera las trazas

asociadas.

- Parte procedural: da valor a las propiedades de los elementos creados.

- GeneralizationParent2Table

  - Parte declarativa: crea el elemento PersistentTable y PrimaryKey a partir del elemento Class (si cumple el patrón configurado *GeneralizationInto = OnlyParentTable*) y genera las trazas asociadas.

  - Parte procedural: Intenta obtener una propiedad válida que actue como clave primaria, si no hay ninguna genera un elemento Column para ello.

- ParentProperty2Column

  - Parte declarativa: crea el elemento Column asociado al elemento Property (si cumple el patrón configurado *GeneralizationInto = OnlyParentTable*) que pertenece a un elemento Class, generando las trazas necesarias.

  - Parte procedural: genera el tipo de datos primitivo correspondiente.

- GeneralizationChild2Table

  - Parte declarativa: crea el elemento PersistentTable y PrimaryKey a partir del elemento Class (si cumple el patrón configurado *GeneralizationInto = OnlyChildTable*) y genera las trazas asociadas.

  - Parte procedural: Intenta obtener una propiedad válida que actue como clave primaria, si no hay ninguna genera un elemento Column para ello.

- ChildProperty2Column

  - Parte declarativa: crea el elemento Column asociado al elemento Property (si cumple el patrón configurado *GeneralizationInto = OnlyChildTable*) que pertenece a un elemento Class, generando las trazas necesarias.

  - Parte procedural: genera el tipo de datos primitivo correspondiente.

- Property2Column

  - Parte declarativa: crea el elemento Column a partir del elemento Property que pertenece a un elemento Class o AssociationClass y genera las trazas asociadas.

  - Parte procedural: Llama a la regla GenerateDataType para generar el tipo de datos primitivo correspondiente.

- Reference2Column

- Parte declarativa: crea los elementos Column y ForeignKey a partir del elemento Property y genera las trazas asociadas.

- Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y almacena el elemento ForeignKey en la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- AssociationClass2Table

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento AssociationClass y genera las trazas asociadas.

  - Parte procedural: Intenta obtener una propiedad válida que actue como clave primaria, si no hay ninguna genera un elemento Column para ello.

- AssociationClassEnd2Column

  - Parte declarativa: crea los elementos Column y ForeignKey a partir del elemento Property que pertenece a un elemento AssociationClass y genera las trazas asociadas.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y almacena el elemento ForeignKey en la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- CompositionProperty2Column

  - Parte declarativa: crea los elementos Column y ForeignKey a partir del elemento Property (si cumple el patrón configurado *CompositionAs = IntegrityConstraint*) y genera las trazas asociadas.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- AssociationOneToOne2Column

  - Parte declarativa: crea los elementos Column y ForeignKey a partir del elemento Association con cardinalidad uno a uno y genera las trazas asociadas.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- DataTypeAssociationOneToOne2Column

  - Parte declarativa: crea los elementos Column y ForeignKey a partir del elemento Association con cardinalidad uno a uno y con un extremo de la asociación de tipo

DataType (si cumple el patrón configurado *DataTypeAs = Table*) y genera las trazas asociadas.

- Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- EnumerationAssociationOneToOne2Column

  - Parte declarativa: crea los elementos Column y ForeignKey a partir del elemento Association con cardinalidad uno a uno y con un extremo de la asociación de tipo Enumeration (si cumple el patrón configurado *EnumerationAs = Table*) y genera las trazas asociadas.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- CompositionOneToOne2Column

  - Parte declarativa: crea los elementos Column y ForeignKey a partir del elemento Association (que representa a una agregación/composición) con cardinalidad uno a uno (si NO cumple el patrón configurado *CompositionAs = IntegrityConstraint* y cumple el patrón configurado *DataTypeAs = Table*) y genera las trazas asociadas.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- Association2Table

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Association con cardinalidad uno a muchos (si cumple el patrón configurado *AssociationInto = Table*) y genera las trazas asociadas.

  - Parte procedural: Añade la PrimaryKey a la tabla generada.

- AssociationEnd2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con cardinalidad uno a muchos (si cumple el patrón configurado *AssociationInto = Table*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- AssociationNav2Table

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Association con cardinalidad uno a muchos y navegabilidad (si cumple el patrón configurado *AssociationInto = Table*) y genera las trazas necesarias.

  - Parte procedural: Añade la PrimaryKey a la tabla generada.

- AssociationEndNav2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con cardinalidad uno a muchos y navegabilidad (si cumple el patrón configurado *AssociationInto = Table*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y Foreignkey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- Association2Reference

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Association con cardinalidad muchos a muchos (si cumple el patrón configurado *AssociationInto = Reference*), asociando los elementos necesarios del modelo de trazas.

  - Parte procedural: Añade la PrimaryKey a la tabla generada.

- AssociationEnd2ReferenceColumn

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con cardinalidad muchos a muchos (si cumple el patrón configurado *AssociationInto = Reference*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- AssociationNav2Reference

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Association con cardinalidad muchos a muchos y navegabilidad (si cumple el patrón configurado *AssociationInto = Reference*) asociando los elementos necesarios del modelo de trazas.

  - Parte procedural: Añade la PrimaryKey a la tabla generada.

- AssociationEndNav2ReferenceColumn

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con cardinalidad muchos a muchos y navegabilidad (si cumple el patrón configurado *AssociationInto = Reference*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- AssociationUpperBound2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con cardinalidad uno a muchos (si cumple el patrón configurado *AssociationInto = Reference*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- Composition2Table

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Association (que es de tipo agregación/composición) con cardinalidad uno a muchos (si cumple el patrón configurado *CompositionAs = Table*) asociando los elementos necesarios del modelo de trazas.

  - Parte procedural: Añade la PrimaryKey a la tabla generada.

- CompositionEnd2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association (si cumple el patrón configurado *CompositionAs = Table*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- CompositionNav2Table

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Association con cardinalidad uno a muchos y navegabilidad (si cumple el patrón configurado *CompositionAs = Table*), generando los elementos necesarios del modelo de trazas.

- Parte procedural: Añade la PrimaryKey a la tabla generada.

- CompositionEndNav2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con cardinalidad uno a muchos y navegabilidad (si cumple el patrón configurado *CompositionAs = Table*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- Composition2Reference

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Association (que es de tipo agregación/composición) con cardinalidad muchos a muchos (si cumple el patrón configurado *CompositionAs = Reference*) asociando los elementos necesarios del modelo de trazas.

  - Parte procedural: Añade la PrimaryKey a la tabla generada.

- CompositionEnd2ReferenceColumn

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association (si cumple el patrón configurado *CompositionAs = Reference*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- CompositionNav2Reference

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento Association con cardinalidad muchos a muchos y navegabilidad (si cumple el patrón configurado *CompositionAs = Reference*), asociando los elementos necesarios del modelo de trazas.

  - Parte procedural: Añade la PrimaryKey a la tabla generada.

- CompositionEndNav2ReferenceColumn

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con cardinalidad muchos a muchos y navegabilidad (si cumple el patrón configurado *CompositionAs = Reference*), generando las trazas necesarias.

- Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- CompositionUpperBound2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con cardinalidad uno a muchos (si cumple el patrón configurado *CompositionAs = Reference*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- DataTypeProperties2Columns

  - Parte declarativa: crea el elemento Column asociado al elemento Property que pertenece a un elemento DataType (si cumple el patrón configurado *DataTypeAs = Table*), generando las trazas necesarias.

  - Parte procedural: genera el tipo de datos primitivo correspondiente.

- DataTypeAssociationEnd2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es parte de un elemento Association con un elemento DataType en algún extremo (si cumple el patrón configurado *DataTypeAs = Table*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- DataTypeMultivaluatedProperty2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Property que es un elemento multivaluado dentro de un elemento Class y cuya propiedad type es un elemento DataType (si cumple el patrón configurado *DataTypeAs = Table*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- DataTypeAssociationToMany2Table

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del

elemento DataType (si cumple el patrón configurado *DataTypeAs = Property*) y genera las trazas asociadas.

- Parte procedural: Añade la PrimaryKey a la tabla generada.

- DataTypeAssociationEndToMany2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociados al elemento Property que es parte de un elemento Association con un elemento DataType en alguno de los dos extremos (si cumple el patrón configurado *DataTypeAs = Property*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- DataTypeAssociationProperties2Columns

  - Parte declarativa: crea el elemento Column asociado al elemento Property que pertenece a un elemento DataType (si cumple el patrón configurado *DataTypeAs = Property*), generando las trazas necesarias.

  - Parte procedural: genera el tipo de datos primitivo correspondiente.

- DataTypeProperty2Table

  - Parte declarativa: crea los elementos PersistentTable y PrimaryKey a partir del elemento DataType (si cumple el patrón configurado *DataTypeAs = Property*) y genera las trazas asociadas.

  - Parte procedural: Añade la PrimaryKey a la tabla generada.

- DataTypeMultivaluatedProperties2Columns

  - Parte declarativa: crea el elemento Column asociado al elemento Property que pertenece a un elemento DataType (si cumple el patrón configurado *DataTypeAs = Property*), generando las trazas necesarias.

  - Parte procedural: genera el tipo de datos primitivo correspondiente.

- MultivaluatedDataTypeProperty2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociados al elemento Property que es multivaluado y su tipo es un DataType (si cumple el patrón configurado *DataTypeAs = Property*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para

transformar el tipo de datos en la última regla lanzada.

- DataTypeAssociation2Column
  - Parte procedural: para cada elemento Property de un DataType que pertenece a una asociación llama a generateDataTypeProperty2Column para crear un elemento de tipo Column (si cumple el patrón configurado *DataTypeAs = Property*).
- DataTypeProperty2Column
  - Parte procedural: para cada elemento Property de un DataType que es referenciado como objeto multivaluado, llama a la regla generateDataTypeProperty2Column para crear un elemento de tipo Column (si cumple el patrón configurado *DataTypeAs = Property*).
- EnumerationMultivaluatedProperty2Column
  - Parte declarativa: crea los elementos Column y ForeignKey asociados al elemento Property que es multivaluado y su tipo es un Enumeration (si cumple el patrón configurado *EnumerationAs = Table*), generando las trazas necesarias.
  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.
- EnumerationAssociationEnd2Column
  - Parte declarativa: crea los elementos Column y ForeignKey asociados al elemento Property que es parte de una asociación y su tipo es un Enumeration (si cumple el patrón configurado *EnumerationAs = Table*), generando las trazas necesarias.
  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.
- EnumerationAssociationOrProperty2Table
  - Parte declarativa: crea los elementos PersistentTable, PrimaryKey, Column, CheckConstraint y SearchConditionDefault a partir del elemento Enumeration (si cumple el patrón configurado EnumerationAs = *Property*) y genera las trazas asociadas.
  - Parte procedural: da valor a las propiedades de los elementos creados.
- EnumerationAssociationEndToMany2Column

- Parte declarativa: crea los elementos Column y ForeignKey asociados al elemento Property que es parte de una asociación y su tipo es un Enumeration (si cumple el patrón configurado *EnumerationAs = Property*), generando las trazas necesarias.

- Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- MultivaluatedEnumerationProperty2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociados al elemento Property que es multivaluado y su tipo es un Enumeration (si cumple el patrón configurado *EnumerationAs = Property*), generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y ForeignKey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- EnumerationAssociation2Column

  - Parte declarativa: crea el elemento Column asociado al elemento Property que es parte de una asociación y su tipo es un Enumeration (si cumple el patrón configurado *EnumerationAs = Property*), generando las trazas necesarias.

  - Parte procedural: añade el elemento Column en la tabla correspondiente y genera el tipo primitivo.

- EnumerationProperty2Column

  - Parte declarativa: crea el elemento Column asociado al elemento Property que referencia es multivaluado y su tipo es un Enumeration (si cumple el patrón configurado *EnumerationAs = Property*), generando las trazas necesarias.

  - Parte procedural: añade el elemento Column en la tabla correspondiente y genera el tipo primitivo.

- GeneralizationParentProperty2Column

  - Parte declarativa: crea las trazas necesarias para el elemento Generalization y las pasa como parámetro para cada elemento Property del elemento Class que participa como padre en la generalización.

  - Parte procedural: a partir del elemento Generalization (si cumple el patrón configurado *GeneralizationInto = OnlyParentTable*) se obtiene el elemento Class padre y para cada elemento Property contenido se llama a la regla createGeneralizationParentProperty2Column para crear un elemento de tipo Column.

- GeneralizationChildProperty2Column

  - Parte declarativa: crea las trazas necesarias para el elemento Generalization y las pasa como parámetro para cada elemento Property del elemento Class que participa como hijo en la generalización.

  - Parte procedural: a partir del elemento Generalization (si cumple el patrón configurado *GeneralizationInto = OnlyChildTable*) se obtiene el elemento Class hijo y para cada elemento Property contenido se llama a la regla createGeneralizationChildProperty2Column para crear un elemento de tipo Column.

- Generalization2Column

  - Parte declarativa: crea los elementos Column y ForeignKey asociado al elemento Generalization (si cumple el patrón configurado *GeneralizationInto = AllTables*) , generando las trazas necesarias.

  - Parte procedural: da valor a las propiedades de los elementos Column y Foreignkey y añade el elemento ForeignKey a la colección FKSeq para transformar el tipo de datos en la última regla lanzada.

- MultivaluatedProperty2Column

  - Parte declarativa: crea los elementos Column, PersistentTable, PrimaryKey y ForeignKey asociados al elemento Property, contenido dentro de un elemento Class cuyo tipo es distinto de DataType o Enumeration y con cardinalidad uno a muchos o muchos a muchos, generando las trazas necesarias.

  - Parte procedural: asigna los valores necesarios a las propiedades de los elementos PrimaryKey y ForeignKey creados.

- Reglas nombradas:

  - GenerateDatabase

    - Parte declarativa: crea el elemento raíz Database del modelo de base de datos.

    - Parte procedural: asigna el nuevo elemento a la variable database.

  - CreatePK

    - Parte declarativa: crea un elemento Column del modelo de base de datos.

    - Parte procedural: llama a la regla Integer2IntegerDataType para generar el tipo de datos y resuelve a partir de la clase en que tabla debe incluirse la columna

generada.

- GenerateDataType
  - Parte procedural: Obtiene el tipo de datos de la propiedad y lo discrimina para llamar a la regla correspondiente.
- String2CharacterDataString
  - Parte declarativa: crea un elemento CharacterStringDataType.
  - Parte procedural: lo asigna a la columna que recibe como parámetro.
- Integer2IntegerDataType
  - Parte declarativa: crea un elemento IntegerDataType.
  - Parte procedural: lo asigna a la columna que recibe como parámetro.
- UnlimitedNatural2NumericalDataType
  - Parte declarativa: crea un elemento NumericalDataType.
  - Parte procedural: lo asigna a la columna que recibe como parámetro.
- Boolean2BooleanDataType
  - Parte declarativa: crea un elemento BooleanDataType.
  - Parte procedural: lo asigna a la columna que recibe como parámetro.
- GenerateDataTypeProperty2Column
  - Parte declarativa: crea un elemento Column y los elementos necesarios del modelo de trazas.
  - Parte procedural: crea el tipo de datos adecuado y asigna la columna a la tabla que recibe como parámetro.
- createGeneralizationParentProperty2Column
  - Parte declarativa: crea un elemento Column y los elementos necesarios del modelo de trazas.
  - Parte procedural: genera el tipo de datos asociado al elemento Column y asigna la columna a la tabla que se obtiene a partir de la generalización que recibe como parámetro.

- createGeneralizationChildProperty2Column

  - Parte declarativa: crea un elemento de tipo Column y los elementos necesarios del modelo de trazas.

  - Parte procedural: genera el tipo de datos asociado al elemento Column y asigna la columna a la tabla que se obtiene a partir de la generalización que recibe como parámetro.

- relatedTable

  - Parte procedural: Comprueba si la clase recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla en la cuál tiene que situar los elementos Column y ForeignKey que recibe como parámetro, sino, la tabla en la que situar los elementos es la que se corresponde con la clase recibida como parámetro.

- relatedColumn2Table

  - Parte procedural: Comprueba si la clase recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla en la cuál tiene que situar el elemento Column que recibe como parámetro, sino, la tabla en la que situar el elemento Column es la que se corresponde con la clase recibida como parámetro.

- relatedReferences

  - Parte procedural: Comprueba si la clase recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla a la cuál tienen que referenciar los elementos Column y ForeignKey que recibe como parámetro, sino, la tabla a la que referenciar con los elementos es la que se corresponde con la clase recibida como parámetro.

- RelatedEnd2Column

  - Parte procedural: Comprueba si la clase recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla a la cuál tiene que referenciar el elemento ForeignKey que recibe como parámetro, sino, la tabla a la que referenciar con la ForeignKey es la que se corresponde con la clase recibida como parámetro.

- RelatedUpperBound2Column

  - Parte procedural:

    Comprueba si la clase *src* recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase

correspondiente y obtiene la tabla en la cuál tiene que situar los elementos Column y ForeignKey que recibe como parámetro, sino, la tabla en la que situar los elementos es la que se corresponde con la clase recibida como parámetro.

Comprueba si la clase *dst* recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla a la cuál tienen que referenciar los elementos Column y ForeignKey que recibe como parámetro, sino, la tabla a la que referenciar con los elementos es la que se corresponde con la clase recibida como parámetro.

Librería para la transformación de UML2 a Base de Datos

En este punto se presentan los helpers o funciones utilizadas en la transformación de UML a Base de Datos. Estos helpers son utilizados mayoritariamente para comprobar la condición de cada regla, es decir, si una regla se tiene que ejecutar o no.

- isOneToOne
  - Tipo: Association
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una asociación tiene cardinalidad uno a uno.
- isOneToOne
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una propiedad tiene cardinalidad uno a uno.
- isAssociation
  - Tipo: Association
  - Valor de retorno: Boolean
  - Descripción: Comprueba si los extremos del elemento asociación son de tipo Class.
- isAssociation
  - Tipo: Property
  - Valor de retorno: Boolean

- Descripción: Comprueba si los extremos del elemento asociación al cuál pertenece dicha propiedad son de tipo Class.

- isDataTypeAssociation
  - Tipo: Association
  - Valor de retorno: Boolean
  - Descripción: Comprueba si al menos un extremo del elemento asociación es de tipo DataType.
- isDataTypeAssociation
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si al menos un extremo del elemento asociación al cuál pertenece dicha propiedad es de tipo DataType.
- isEnumerationAssociation
  - Tipo: Association
  - Valor de retorno: Boolean
  - Descripción: Comprueba si al menos un extremo del elemento asociación es de tipo Enumeration.
- isEnumerationAssociation
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si al menos un extremo del elemento asociación al cuál pertenece dicha propiedad es de tipo Enumeration.
- isNavigable
  - Tipo: Association
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una asociación tiene navegabilidad asociada.
- isNavigable

- Tipo: Property
- Valor de retorno: Boolean
- Descripción: Comprueba si la asociación asociada a una propiedad tiene navegabilidad.

- isMultivaluated
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una propiedad tiene cardinalidad máxima mayor que uno.
- isNotNull
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una propiedad tiene cardinaliad mínima igual a cero.
- isReference
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una propiedad tiene como tipo de datos asociado otra clase del modelo.
- isPropertyClass
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una propiedad pertenece a un elemento de tipo Class.
- isPropertyAssociationClass
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una propiedad pertenece a un elemento de tipo AssociationClass.

- getPKCandidate
    - Tipo: Class
    - Valor de retorno: Property
    - Descripción: Obtiene, si es posible, la propiedad de la clase que pueda ser considerada clave primaria (no multivaluada, no nula, no navegable, no referencia).
- isAbstractClass
    - Tipo: Class
    - Valor de retorno: Boolean
    - Descripción: Comprueba si una clase es abstracta.
- hasAbstractClass
    - Tipo: Property
    - Valor de retorno: Boolean
    - Descripción: Comprueba si la propiedad forma parte de una clase abstracta.
- isAbstractAssociation
    - Tipo: Property
    - Valor de retorno: Boolean
    - Descripción: Comprueba si la propiedad que forma parte de una asociación contiene alguna clase abstracta.
- hasAbstractEnd
    - Tipo: Association
    - Valor de retorno: Boolean
    - Descripción: Comprueba si la asociación contiene alguna clase abstracta.
- isManyToMany
    - Tipo: Association
    - Valor de retorno: Boolean

- Descripción: Comprueba si una asociación tiene cardinalidad muchos a muchos.

- isManyToMany
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una asociación a la cuál pertenece dicha propiedad tiene cardinalidad muchos a muchos.

- isOneToMany
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una asociación a la cuál pertenece dicha propiedad tiene cardinalidad uno a muchos.

- associationWithUpperBound
  - Tipo: Element
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Element es de tipo DataType o Enumeration y pertenece a una asociación cuya cardinalidad máxima es mayor que uno.

- associationWithUpperBoundToOne
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si una propiedad de una asociación cuyo extremo es de tipo DataType o Enumeration y tiene como cardinalidad máxima uno.

- dataTypeWithUpperBound
  - Tipo: Element
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Element es de tipo DataType, pertenece a una propiedad de una clase y esta propiedad tiene cardinalidad máxima mayor que uno.

- enumerationWithUpperBound

- Tipo: Element
- Valor de retorno: Boolean
- Descripción: Comprueba si Element es de tipo Enumeration, pertenece a una propiedad de una clase y esta propiedad tiene cardinalidad máxima mayor que uno.

- dataTypeWithUpperBoundToOne
  - Tipo: Element
  - Valor de retorno: Boolean
  - Descripción: Comprueba que si Element es de tipo Property y su propiedad Type es un DataType, tiene cardinalidad máxima uno.

- enumerationWithUpperBoundToOne
  - Tipo: Element
  - Valor de retorno: Boolean
  - Descripción: Comprueba que si Element es de tipo Property y su propiedad Type es un Enumeration, tiene cardinalidad máxima uno.

- isDataTypeAssociationEnd
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Property pertenece a una asociación y su propiedad Type es un elemento DataType.

- isEnumerationAssociationEnd
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Property pertenece a una asociación y su propiedad Type es un elemento Enumeration.

- isDataTypeAssociationProperty
  - Tipo: Property
  - Valor de retorno: Boolean

- Descripción: Comprueba si Property pertenece a una asociación y en dicha asociación hay un elemento Property cuya propiedad Type es un elemento DataType.

- isEnumerationAssociationProperty
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Property pertenece a una asociación y en dicha asociación hay un elemento Property cuya propiedad Type es un elemento Enumeration.

- dataTypeProperties
  - Tipo: Element
  - Valor de retorno: Boolean
  - Descripción: Comprueba que Element es de tipo DataType y pertenece a una asociación con cardinalidad mayor que uno.

- isDataTypeReference
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Property tiene valor en su propiedad Type y es de tipo DataType.

- isEnumerationReference
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Property tiene valor en su propiedad Type y es de tipo Enumeration.

- isComposition
  - Tipo: Association
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Association es de tipo composition.

- isComposition
    - Tipo: Property
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Property pertenece a una asociación de tipo composition.
- isNotCompositionProperty
    - Tipo: Property
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Property pertenece a una asociación y ésta es de tipo aggregation/shared.
- transformOnlyParentTable
    - Tipo: Class
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Class que pertenece a una Generalización tiene algún patrón configurado con valor *GeneralizationAs = OnlyParentTable*.
- transformOnlyChildTable
    - Tipo: Class
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Class que pertenece a una Generalización tiene algún patrón configurado con valor *GeneralizationAs = OnlyChildTable*.
- executeGeneralizationRule (ruleName)
    - Tipo: Generalization
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Generalization cumple el patrón de Generalización y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto *AllTables*.

- executeParentOrChildTableRule (ruleName)
    - Tipo: Element
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Element cumple el patrón de Generalización y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *AllTables*.
- executeParentTableRule (ruleName)
    - Tipo: Element
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Element cumple el patrón de Generalización y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *AllTables*.
- executeChildTableRule (ruleName)
    - Tipo: Element
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si un elemento Element cumple el patrón de Generalización y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *AllTables*.
- executeAssociationRule (ruleName)
    - Tipo: Association
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Association cumple el patrón de Asociación y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *Reference*.

- executeAssociationRule (ruleName)
    - Tipo: Property
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Property cumple el patrón de Asociación y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *Reference*.
- executeCompositionRule (ruleName)
    - Tipo: Association
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Association cumple el patrón de Agregación/Composición y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *IntegrityConstraint*.
- executeCompositionRule (ruleName)
    - Tipo: Property
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Property cumple el patrón de Agregación/Composición y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *IntegrityConstraint*.
- executeDataTypeRule (ruleName)
    - Tipo: Element
    - Parámetro: String
    - Valor de retorno: Boolean
    - Descripción: Comprueba si Element cumple el patrón para el elemento DataType de UML y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no

tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *Table*.

- executeEnumerationRule (ruleName)
  - Tipo: Element
  - Parámetro: String
  - Valor de retorno: Boolean
  - Descripción: Comprueba si Element cumple el patrón para el elemento Enumeration de UML y ha sido configurado para transformarse de acuerdo a la regla *ruleName*. Si no tiene patrón configurado, comprueba si *ruleName* coincide con la regla por defecto, *Property*.

4.2.3.1. Cálculo y generación del modelo de diferencias

Esta transformación se lanza cuando se produce algún cambio en el modelo UML y existe un modelo de Base de Datos. El objetivo de esta transformación es generar un modelo de diferencias sobre el modelo de Base de Datos, que servirá para lanzar la sincronización, a partir del modelo UML, el modelo de Base de Datos, el modelo de Trazas y un modelo de diferencias que se genera cuando se producen cambios en el modelo UML.

- Cabecera:
  - Entradas:
    - Metamodelo de diferencias de EMFCompare
    - Modelo de diferencias del modelo UML2 de entrada
    - Metamodelo de trazas
    - Modelo de trazas
    - Metamodelo de base de datos de Datatools
    - Modelo de base de datos de Datatools
    - Metamodelo UML2
    - Modelo de entrada UML2
    - Metamodelo de Configuración de transformaciones
    - Modelo de salida de Configuración de transformaciones

- Salidas:
  - Metamodelo de diferencias de EMFCompare modificado (GVDiffModel)
  - Modelo de diferencias de salida sobre el modelo de base de datos

- Variables:
  - diffmodel: referencia al elemento raíz del modelo de salida
  - associationsSync: colección auxiliar de elementos Association de UML2
  - transformClass: colección auxiliar de elementos Class de UML2

- Helpers:
  - isReference
    - Tipo: EObject
    - Valor de retorno: Boolean
    - Descripción: Comprueba si un tipo de datos es una clase del modelo.
  - isOneToOne
    - Tipo: EObject
    - Valor de retorno: Boolean
    - Descripción: Comprueba si el elemento Association tiene cardinalidad uno a uno.
  - isManyToMany
    - Tipo: EObject
    - Valor de retorno: Boolean
    - Descripción: Comprueba si el elemento Association tiene cardinalidad muchos a muchos.
  - isDataTypePropertyPattern
    - Tipo: DataType

        - Valor de retorno: Boolean
        - Descripción: Comprueba si para el elemento DataType existe algún patrón configurado cuya regla sea *DataTypeAs = Property*.
    - isEnumerationPropertyPattern
        - Tipo: Enumeration
        - Valor de retorno: Boolean
        - Descripción: Comprueba si para el elemento Enumeration existe algún patrón configurado cuya regla sea *EnumerationAs = Property*.
    - isGeneralizationParentPattern
        - Tipo: Class
        - Valor de retorno: Boolean
        - Descripción: Comprueba si para el elemento Class existe algún patrón configurado cuya regla sea *GeneralizationInto = OnlyParentTable o AllTables*.
    - isGeneralizationChildPattern
        - Tipo: Class
        - Valor de retorno: Boolean
        - Descripción: Comprueba si para el elemento Class existe algún patrón configurado cuya regla sea *GeneralizationInto = OnlyChildTable*.
    - HasGeneralizationConfiguredPattern
        - Tipo: Element
        - Valor de retorno: Boolean
        - Descripción: Comprueba para el elemento Element existe algún patrón configurado de tipo *GeneralizationInto*.

- Reglas anónimas:
    - Diffmodel
        - Parte declarativa: crea el elemento DiffModel del metamodelo GVDiffModel a

partir del elemento DiffModel del metamodelo de diferencias de EMFCompare.

- Parte procedural: asigna el elemento creado a la variable global diffmodel.

- NameChanged

  - Parte declarativa: se provoca el disparo de la regla con elementos UpdateAttribute cuya propiedad name tenga el valor 'name'.

  - Parte procedural: genera un elemento UpdateAttribute para cada elemento que se encuentre en la traza como elemento destino.

- AbstractChanged

  - Parte declarativa: se provoca el disparo de la regla con elementos UpdateAttribute cuya propiedad name tenga el valor 'abstract' sobre elementos de tipo Class.

  - Parte procedural: se comprueba que la clase no forme parte de alguna jerarquía de herencia. En caso negativo, se generan elementos de borrado para los elementos del metamodelo destino para la clase, sus nodos hijos y las relaciones en las que participe. En caso afirmativo se generan elementos para añadir la clase, sus nodos hijos y las posibles relaciones en las que participe.

- PropertyBoundElementAdded

  - Parte declarativa: se provoca el disparo de la regla con elementos AddModelElement cuya propiedad rightElement sea de tipo UnlimitedNatural sobre un elemento Property de una clase.

  - Parte procedural: elimina los nodos hijos de las trazas en las que participe la clase como elemento origen. Genera un elemento AddModelElement de la propiedad.

    Si la propiedad Type del elemento Property es Enumeration se borran las trazas correspondientes al elemento Enumeration y se vuelve a generar un elemento AddModelElement de Enumeration.

    Si la propiedad Type del elemento Property es DataType se borran las trazas correspondientes al elemento DataType y se vuelve a generar un elemento AddModelElement de DataType.

- PropertyBoundChanged

  - Parte declarativa: se provoca el disparo de la regla con elementos UpdateAttribute cuya propiedad name tenga el valor 'value' sobre elementos contenidos en una propiedad de una clase.

  - Parte procedural:

Si es un elemento Property de una Asociación navegable, se eliminan los nodos hijos de las trazas en las que participe la asociación como elemento origen y se vuelve a generar un elemento AddModelElement de Association.

Sino, se eliminan los nodos hijos de las trazas en las que participe la clase como elemento origen y genera un elemento AddModelElement para la propiedad.

Si la propiedad Type del elemento Property es Enumeration se borran las trazas correspondientes al elemento Enumeration y se vuelve a generar un elemento AddModelElement de Enumeration.

Si la propiedad Type del elemento Property es DataType se borran las trazas correspondientes al elemento DataType y se vuelve a generar un elemento AddModelElement de DataType.

- AssociationBoundChanged
  - Parte declarativa: se provoca el disparo de la regla con elementos UpdateAttribute cuya propiedad name tenga el valor 'value' sobre elementos contenidos en una propiedad de una asociación.
  - Parte procedural:

    Si tanto para el elemento Association como para cada uno de sus elementos Property contenidos existen trazas, se eliminan los nodos hijos de las trazas en las que participe la asociación como elemento origen y se genera un elemento AddModelElement para la asociación.

    Si la propiedad Type del elemento Property, contenido en la asociación, es Enumeration se borran las trazas correspondientes al elemento Enumeration y se vuelve a generar un elemento AddModelElement de Enumeration.

    Si la propiedad Type del elemento Property, contenido en la asociación, es DataType se borran las trazas correspondientes al elemento DataType y se vuelve a generar un elemento AddModelElement de DataType.
- PrimitiveTypeAdded
  - Parte declarativa: Copia el elemento de entrada en la salida.
- PrimitiveTypeUpdated
  - Parte declarativa: Copia el elemento de entrada en la salida.
- ReferenceTypeAdded
  - Parte declarativa: Copia el elemento de entrada en la salida.

- ReferenceTypeUpdated

  - Parte declarativa: genera un elemento AddModelElement a partir de un elemento UpdateUniqueReferenceValue cuya propiedad reference es de tipo 'type' y sea una referencia.

  - Parte procedural: elimina los nodos hijos de las trazas en las que participe la asociación como elemento origen. Genera un elemento AddModelElement para el elemento.

- ElementAdded

  - Parte declarativa: se provoca el disparo de la regla con elementos AddModelElement cuya propiedad rightElement no sea de tipo LiteralUnlimitedNatural, propiedad o clase.

  - Parte procedural:

    Si la propiedad rightElement es de tipo Package, Class, DataType o AssociationClass genera todos sus nodos hijos.

    Si la propiedad rightElement es de tipo Property y la propiedad Type es de tipo Enumeration o DataType y no tienen trazas, genera un elemento AddModelElement para el elemento Enumeration o DataType y todos sus nodos hijos.

    Si la propiedad rightElement es de tipo Association y en alguno de sus elementos Property la propiedad Type es de tipo Enumeration o DataType y no tienen trazas, genera un elemento AddModelElement para el elemento Enumeration o DataType y todos sus nodos hijos.

    Si la propiedad rightElement es de tipo Generalization, genera un elemento AddModelElement para el elemento Generalization. Si las clases relacionadas (general y specific) están configuradas con patrón *GeneralizationInto = OnlyParentTable* o *GeneralizationInto = OnlyChildTable* y tienen trazas, las eliminan y se vuelve a generar un elemento AddModelElement para cada elemento de la generalización.

- ElementRemoved

  - Parte declarativa: copia la entrada en la salida.

  - Parte procedural: elimina los nodos hijos de las trazas en las que participe la propiedad leftElement como elemento origen.

    Para cada elemento Enumeration, si tiene patrón configurado *EnumerationAs=Property*, no aparece en la propiedad Type de ningún elemento Property y tiene trazas, elimina los nodos hijos de las trazas en las

que participe el elemento Enumeration como elemento origen.

Para cada elemento DataType, si tiene patrón configurado *DataTypeAs=Property*, no aparece en la propiedad Type de ningún elemento Property y tiene trazas, elimina los nodos hijos de las trazas en las que participe el elemento DataType como elemento origen y cada uno de sus elementos contenidos.

Para cada elemento Class, si no pertenece a ninguna generalización y no tiene traza, se genera un elemento AddModelElement del elemento Class. En cambio, si tiene traza y tiene algún patrón configurado *GeneralizationInto=OnlyParentTable o AllTables*, hay que eliminar cada una de sus propiedades y volverlas a generar.

Si la propiedad rightParent es un elemento Class, esta clase participa en alguna generalización como elemento hijo y no tiene traza alguna, se genera un elemento AddModelElement del elemento Class. Además, para cada una de sus propiedades se tienen que eliminar y volver a generar.

- Reglas nombradas:
  - removeHierarchy
    - Parte procedural: para cada nodo hijo del elemento genera un elemento RemoveModelElement y elimina su jerarquía.
  - AddHierarchy
    - Parte procedural: para cada nodo hijo del elemento genera un elemento AddModelElement y añade su jerarquía.
  - GenerateUpdateAttribute
    - Parte declarativa: genera un elemento de tipo UpdateAtribute con los valores pasados como parámetros.
    - Parte procedural: añade el elemento creado al modelo.
  - GenerateRemoveModelElement
    - Parte declarativa: genera un elemento de tipo RemoveModelElement con los valores pasados como parámetros.
    - Parte procedural: añade el elemento creado al modelo.
  - GenerateAddModelElement

- Parte declarativa: genera un elemento de tipo AddModelElement con los valores pasados como parámetros.

- Parte procedural: añade el elemento creado al modelo.

4.2.3.2. Combinación de diferencias en el modelo destino

A partir del modelo de diferencias generado en el apartado anterior, se va a regenerar el modelo de Base de Datos y el modelo de Trazas sobre el cuál se aplicarán los cambios de acuerdo a las modificaciones realizadas en el modelo UML.

- Cabecera:
  - Entradas:
    - Metamodelo de base de datos de Datatools
    - Modelo de base de datos de Datatools
    - Modelo de diferencias del modelo de base de datos basado en GVDiffModel
    - Metamodelo de diferencias de GVDiffModel
    - Modelo de trazas
    - Metamodelo de trazas
    - Metamodelo UML2
    - Modelo de entrada UML2
    - Metamodelo de Configuración de transformaciones
    - Modelo de salida de Configuración de transformaciones
  - Salidas:
    - Metamodelo de base de datos de Datatools
    - Modelo de base de datos
    - Metamodelo de Trazas
    - Modelo de trazas

- Librería:
  - Enlace a la librería DiffMerge_library.atl

- Variables:
  - __wmodel: Elemento raiz del modelo de trazas
  - __model_IN: Elemento que representa el modelo de entrada en el modelo de trazas
  - __model_OUT: Elemento que representa el modelo de salida en el modelo de trazas
  - associations: colección de elementos Association de UML2
  - propertiesOfAssociation: colección de elementos Property de UML2
  - database: elemento Database temporal del modelo SQLMODEL
  - tmp: elemento temporal del modelo SQLMODEL
  - src: elemento temporal del modelo SQLMODEL
  - dst: elemento temporal del modelo SQLMODEL
  - data: elemento temporal del modelo SQLMODEL
  - dataTypeTransformed: variable booleana

- Reglas anónimas:
  - TraceModel
    - Parte declarativa: copia el elemento de entrada en la salida si no ha sido eliminado.
    - Parte procedural: inicializa la variable __wmodel.
  - TraceLink
    - Parte declarativa: copia el elemento de entrada en la salida si no ha sido eliminado.
  - TraceLinkEnd
    - Parte declarativa: copia el elemento de entrada en la salida si no ha sido eliminado.

- TraceModelRef
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: inicializa la variables __model_IN y __model_OUT.
- ElementRef
  - Parte declarativa: copia el elemento de entrada en la salida si no ha sido eliminado.
- Database
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica.
- Role
  - Parte declarativa: copia el elemento de entrada en la salida.
- User
  - Parte declarativa: copia el elemento de entrada en la salida.
- Group
  - Parte declarativa: copia el elemento de entrada en la salida.
- Schema
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica.
- Index
  - Parte declarativa: copia el elemento de entrada en la salida.
- ViewTable
  - Parte declarativa: copia el elemento de entrada en la salida.
- Tables
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica.

Actualiza las claves ajenas que referencian la tabla.

- Columns
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica.
- IntegerContainedType
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica.
- CharacterStringContainedType
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica.
- FixedPrecisionContainedType
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica.
- BooleanDataTypeContainedType
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica.
- PrimaryKeys
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica. Actualiza columnas que la forman.
- ForeignKeys
  - Parte declarativa: copia el elemento de entrada en la salida.
  - Parte procedural: actualiza los valores de sus propiedades de forma genérica. Actualiza las columnas que la forman.
- UniqueConstraint

- Parte declarativa: copia el elemento de entrada en la salida.

- CheckConstraint
  - Parte declarativa: copia el elemento de entrada en la salida.
- QueryExpressionDefault
  - Parte declarativa: copia el elemento de entrada en la salida.
- SearchConditionDefault
  - Parte declarativa: copia el elemento de entrada en la salida.
- ValueExpressionDefault
  - Parte declarativa: copia el elemento de entrada en la salida.
- RoleAuthorization
  - Parte declarativa: copia el elemento de entrada en la salida.
- Privilege
  - Parte declarativa: copia el elemento de entrada en la salida.
- TransformClass
  - Parte declarativa: genera una tabla y las trazas necesarias a partir de un elemento AddModelElement que debe contener una clase, si la clase no pertenece a ninguna generalización que tenga algún patrón configurado.
  - Parte procedural: busca en el modelo de trazas el elemento Schema a partir del elemento origen paquete para añadir la tabla. Crea una clave primaria para el elemento creado.
- TransformGeneralizationParentClass
  - Parte declarativa: si existe un patrón configurado *GeneralizationInto = OnlyParentTable*, genera una tabla y las trazas necesarias a partir de un elemento AddModelElement que debe contener una clase.
  - Parte procedural: busca en el modelo de trazas el elemento Squema a partir del elemento origen paquete para añadir la tabla. Crea una clave primaria para el elemento creado.
- TransformUnivaluatedParentProperty

- Parte declarativa: genera una columna y las trazas necesarias a partir de un elemento AddModelElement que debe contener una propiedad contenida dentro de un elemento Class o AssociationClass y este tiene a su vez un patrón configurado *GeneralizationInto = OnlyParentTable*.

- Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la columna. Transforma el tipo de datos asociado.

- TransformGeneralizationChildClass

  - Parte declarativa: si existe un patrón configurado *GeneralizationInto = OnlyChildTable*, genera una tabla y las trazas necesarias a partir de un elemento AddModelElement que debe contener una clase.

  - Parte procedural: busca en el modelo de trazas el elemento Squema a partir del elemento origen paquete para añadir la tabla. Crea una clave primaria para el elemento creado.

- TransformUnivaluatedChildProperty

  - Parte declarativa: genera una columna y las trazas necesarias a partir de un elemento AddModelElement que debe contener una propiedad contenida dentro de un elemento Class o AssociationClass y este tiene a su vez un patrón configurado *GeneralizationInto = OnlyChildTable*.

  - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la columna. Transforma el tipo de datos asociado.

- TransformDataType

  - Parte declarativa: si existe un patrón configurado *DataTypeAs = Table*, genera una tabla y las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento DataType.

  - Parte procedural: busca en el modelo de trazas el elemento Squema a partir del elemento origen paquete para añadir la tabla. Crea una clave primaria para el elemento creado.

- TransformEnumeration

  - Parte declarativa: si existe un patrón configurado *EnumerationAs = Table*, genera una tabla y las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Enumeration.

  - Parte procedural: busca en el modelo de trazas el elemento Squema a partir del elemento origen paquete para añadir la tabla. Crea una clave primaria para el

elemento creado.

- TransformAssociationOneToOne

  - Parte declarativa: si existe algún patrón configurado *AssociationInto = Table* o *AssociationInto = Reference*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una asociación uno a uno.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformAssociationManyToMany

  - Parte declarativa: si existe algún patrón configurado *AssociationInto = Table* o *AssociationInto = Reference*, genera una tabla, su clave primaria, la columna que la forma, dos claves ajena y las columnas que las forman además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una asociación muchos a muchos.

  - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave primaria y la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformAssociationOneToMany2Reference

  - Parte declarativa: si existe algún patrón configurado *AssociationInto = Reference*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una asociación uno a muchos.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformAssociationOneToMany2Table

  - Parte declarativa: si existe algún patrón configurado *AssociationInto = Table*, genera una tabla, su clave primaria, la columna que la forma, dos claves ajena y las columnas que las forman además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una asociación uno a muchos.

- Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave primaria y la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformComposition

  - Parte declarativa: si existe algún patrón configurado *CompositionAs = IntegrityConstraint*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una agregación/composición.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformCompositionOneToOne

  - Parte declarativa: si existe algún patrón configurado *CompositionAs = Table o CompositionAs = Reference*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una agregación/composición uno a uno.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformCompositionManyToMany

  - Parte declarativa: si existe algún patrón configurado *CompositionAs = Table* o *CompositionAs = Reference*, genera una tabla, su clave primaria, la columna que la forma, dos claves ajena y las columnas que las forman además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una agregación/composición muchos a muchos.

  - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave primaria y la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformCompositionOneToMany2Reference

  - Parte declarativa: si existe algún patrón configurado *CompositionAs = Reference*, genera una clave ajena y la columna que la forma además de las trazas necesarias

a partir de un elemento AddModelElement que debe contener una agregación/composición uno a muchos.

- Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformCompositionOneToMany2Table

  - Parte declarativa: si existe algún patrón configurado *CompositionAs = Table*, genera una tabla, su clave primaria, la columna que la forma, dos claves ajena y las columnas que las forman además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una agregación/composición uno a muchos.

  - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave primaria y la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformGeneralization

  - Parte declarativa: si existe algún patrón configurado *GeneralizationInto = AllTables*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una generalización.

  - Parte procedural: busca en el modelo de trazas los elementos tabla general y especifíca a partir de los elementos clase que forman la generalización para añadir la clave ajena y la columna que la forma. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformGeneralizationParentProperty2Column

  - Parte declarativa: genera las trazas necesarias para el elemento Generalization y para cada elemento Property del elemento Class que participa en la generalización como padre, realiza una llamada a la regla createGeneralizationParentProperty2Column.

  - Parte procedural: si existe algún patrón configurado *GeneralizationInto = OnlyParentTable*, para cada atributo de la clase específica de la genealización llama a la regla createGeneralizationParentProperty2Column.

- TransformGeneralizationChildProperty2Column

  - Parte declarativa: genera las trazas necesarias para el elemento Generalization y

para cada elemento Property del elemento Class que participa en la generalización como hijo, realiza una llamada a la regla createGeneralizationChildProperty2Column.

- Parte procedural: si existe algún patrón configurado *GeneralizationInto = OnlyChildTable*, para cada atributo de la clase general de la genealización llama a la regla createGeneralizationChildProperty2Column.

- TransformAssociationClass

  - Parte declarativa: genera una tabla, su clave primaria, la columna que la forma, dos claves ajena y las columnas que las forman además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una clase asociación.

  - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave primaria y la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformPrimitiveType

  - Parte declarativa: dispara la regla ante elementos de tipo AddReferenceValue cuya propiedad reference sea del tipo 'type'.

  - Parte procedural: busca en el modelo de trazas la columna a la que afecta partir de la propiedad y genera el tipo de dato correspondiente.

- TransformDataTypePropertyToMany

  - Parte declarativa: si existe algún patrón configurado *DataTypeAs = Property*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Property cuya propiedad Type es un elemento DataType como objeto multivaluado con cardinalidad > 1.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y DataType relacionados mediante la propiedad multivaluada para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformDataTypeAssociationOneToMany

  - Parte declarativa: si existe algún patrón configurado *DataTypeAs = Property*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Association con cardinalidad uno a muchos y uno de los extremos es un

elemento DataType.

- Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y DataType que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformDataTypePropertyToOne2Table

  - Parte declarativa: si existe algún patrón configurado *DataTypeAs = Table*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Property cuya propiedad Type es un elemento DataType como objeto multivaluado con cardinalidad = 1.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y DataType relacionados mediante la propiedad para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformDataTypeAssociationOneToOne2Table

  - Parte declarativa: si existe algún patrón configurado *DataTypeAs = Table*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Association con cardinalidad uno a uno y uno de los extremos es un elemento DataType.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y DataType que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformDataTypePropertyToMany2Table

  - Parte declarativa: si existe algún patrón configurado *DataTypeAs = Table*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Property cuya propiedad Type es un elemento DataType como objeto multivaluado con cardinalidad > 1.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y DataType relacionados mediante la propiedad para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los

tipos de datos para las columnas creadas.

- TransformDataTypeAssociationOneToMany2Table

  - Parte declarativa: si existe algún patrón configurado *DataTypeAs = Table*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Association con cardinalidad uno a muchos y uno de los extremos es un elemento DataType.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y DataType que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformEnumerationPropertyToMany

  - Parte declarativa: si existe algún patrón configurado *EnumerationAs = Property*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Property cuya propiedad Type es un elemento Enumeration como objeto multivaluado con cardinalidad > 1.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y Enumeration relacionados mediante la propiedad multivaluada para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformEnumerationAssociationOneToMany

  - Parte declarativa: si existe algún patrón configurado *EnumerationAs = Property*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Association con cardinalidad uno a muchos y uno de los extremos es un elemento Enumeration.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y DataType que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformEnumerationPropertyToOne

  - Parte declarativa: si existe algún patrón configurado *EnumerationAs = Property*, genera una clave ajena y la columna que la forma además de las trazas necesarias

a partir de un elemento AddModelElement que debe contener un elemento Property cuya propiedad Type es un elemento Enumeration como objeto multivaluado con cardinalidad = 1.

- Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y Enumeration relacionados mediante la propiedad multivaluada para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformEnumerationAssociationOneToOne

  - Parte declarativa: si existe algún patrón configurado *EnumerationAs = Property*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Association con cardinalidad uno a muchos y uno de los extremos es un elemento Enumeration.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y Enumeration que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformEnumerationPropertyToOne2Table

  - Parte declarativa: si existe algún patrón configurado *EnumerationAs = Table*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Property cuya propiedad Type es un elemento Enumeration como objeto multivaluado con cardinalidad = 1.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y Enumeration relacionados mediante la propiedad para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformEnumerationAssociationOneToOne2Table

  - Parte declarativa: si existe algún patrón configurado *EnumerationAs = Table*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Association con cardinalidad uno a uno y uno de los extremos es un elemento Enumeration.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y Enumeration que forman la asociación

para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformEnumerationPropertyToMany2Table

  - Parte declarativa: si existe algún patrón configurado *EnumerationAs = Table*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Property cuya propiedad Type es un elemento Enumeration como objeto multivaluado con cardinalidad > 1.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y Enumeration relacionados mediante la propiedad para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformEnumerationAssociationOneToMany2Table

  - Parte declarativa: si existe algún patrón configurado *EnumerationAs = Table*, genera una clave ajena y la columna que la forma además de las trazas necesarias a partir de un elemento AddModelElement que debe contener un elemento Association con cardinalidad uno a muchos y uno de los extremos es un elemento Enumeration.

  - Parte procedural: busca en el modelo de trazas los elementos tabla origen y destino a partir de los elementos clase y Enumeration que forman la asociación para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave ajena. Genera los tipos de datos para las columnas creadas.

- TransformDataTypeUnivaluatedProperty

  - Parte declarativa: si existe algún patrón configurado *DataTypeAs = Table*, genera una columna y las trazas necesarias a partir de un elemento AddModelElement que debe contener una propiedad contenida dentro de un elemento DataType.

  - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen DataType para añadir la columna. Transforma el tipo de datos asociado.

- TransformDataTypeUnivaluatedProperty2Column

  - Parte declarativa: si existe algún patrón configurado *DataTypeAs = Property*, genera una columna y las trazas necesarias a partir de un elemento AddModelElement que debe contener una propiedad contenida dentro de un elemento DataType.

        - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen DataType para añadir la columna. Transforma el tipo de datos asociado.

    - TransformUnivaluatedProperty

        - Parte declarativa: genera una columna y las trazas necesarias a partir de un elemento AddModelElement que debe contener una propiedad contenida dentro de un elemento Class o AssociationClass.

        - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la columna. Transforma el tipo de datos asociado.

    - TransformMultivaluatedProperty

        - Parte declarativa: genera una tabla, su clave primaria, la columna que la forma, la clave ajena, la columna que la forma y la columna que representa la propiedad además de las trazas necesarias a partir de un elemento AddModelElement que debe contener una propiedad uno a muchos/muchos a muchos contenida dentro de un elemento Class o AssociationClass.

        - Parte procedural: busca en el modelo de trazas el elemento tabla a partir del elemento origen clase para añadir la clave ajena y la columna que la forma. Transforma el tipo de datos asociado. Asigna los valores correctos a la clave primaria y la clave ajena. Genera los tipos de datos para las columnas creadas.

- Reglas nombradas:

    - TransformPackage

        - Parte declarativa: genera un esquema y las trazas necesarias a partir de un elemento Element UML2 que recibe como parámetro cuyo padre es un paquete con elementos.

        - Parte procedural: busca en el modelo de trazas el elemento Database a partir del elemento origen para añadir el esquema y añade al esquema la tabla que recibe como parámetro.

    - TransformDataType2Table

        - Parte declarativa: genera una tabla y las trazas necesarias a partir de un elemento Element UML2 que recibe como parámetro.

        - Parte procedural: busca en el modelo de trazas el elemento Schema a partir del elemento origen paquete para añadir la tabla. Crea una clave primaria para el elemento creado. Da valor a las propiedades de los elementos Column y

ForeignKey que recibe como parámetros.

- TransformEnumeration2Table
  - Parte declarativa: genera una tabla y las trazas necesarias a partir de un elemento Element UML2 que recibe como parámetro.
  - Parte procedural: busca en el modelo de trazas el elemento Schema a partir del elemento origen paquete para añadir la tabla. Crea una clave primaria para el elemento creado. Da valor a las propiedades de los elementos Column y ForeignKey que recibe como parámetros.
- CreatePK
  - Parte declarativa: genera una columna que actuará como miembro de la clave primaria obtenida como parámetro.
  - Parte procedural: Inicializa los valores de las propiedades de la clave primaria y genera el tipo de datos correcto para la columna creada.
- GeneratePK
  - Parte declarativa: genera una columna que actuará como miembro de la clave primaria obtenida como parámetro.
  - Parte procedural: Inicializa los valores de las propiedades de la clave primaria y genera el tipo de datos correcto para la columna creada.
- GenerateFK
  - Parte declarativa: genera una columna que actuará como miembro de la clave ajena obtenida como parámetro.
  - Parte procedural: Inicializa los valores de las propiedades de la clave ajena y genera el tipo de datos correcto para la columna creada.
- GenerateDatatype
  - Parte procedural: discrimina que tipo de datos, tanto básico como referencia, debe generar.
- GenerateReference
  - Parte declarativa: genera una clave ajena.
  - Parte procedural: busca en el modelo de trazas el elemento tabla que participan en la referencia a partir de los elementos clase del modelo. Inicializa las propiuedades de la clave ajena creada y el tipo de datos correcto.

- GenerateCharacterStringDataType
  - Parte declarativa: crea un elemento de tipo CharacterStringDataType.
  - Parte procedural: lo asigna a la columna que recibe como parámetro.
- GenerateIntegerDataType
  - Parte declarativa: crea un elemento de tipo IntegerDataType.
  - Parte procedural: lo asigna a la columna que recibe como parámetro.
- GenerateFixedPrecisionDataType
  - Parte declarativa: crea un elemento de tipo FixedPrecisionDataType.
  - Parte procedural: lo asigna a la columna que recibe como parámetro.
- GenerateBooleanDataType
  - Parte declarativa: crea un elemento de tipo BooleanDataType.
  - Parte procedural: lo asigna a la columna que recibe como parámetro.
- relatedTable
  - Parte procedural: Comprueba si la clase recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla en la cuál tiene que situar los elementos Column y ForeignKey que recibe como parámetro, sino, la tabla en la que situar los elementos es la que se corresponde con la clase recibida como parámetro.
- relatedReferences
  - Parte procedural: Comprueba si la clase recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla a la cuál tienen que referenciar los elementos Column y ForeignKey que recibe como parámetro, sino, la tabla a la que referenciar con los elementos es la que se corresponde con la clase recibida como parámetro.
- RelatedTableAndReferences
  - Parte procedural: Comprueba si la clase recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla a la cuál tienen que referenciar el elemento ForeignKey que recibe como parámetro, sino, la tabla a la que referenciar con los elementos es la que se corresponde con la clase recibida como parámetro.

- relatedColumn2Table

  - Parte procedural: Comprueba si la clase recibida como parámetro pertenece a alguna Generalización y existe algún patrón configurado. Si lo hay, busca la clase correspondiente y obtiene la tabla en la cuál tiene que situar el elemento Column que recibe como parámetro, sino, la tabla en la que situar el elemento Column es la que se corresponde con la clase recibida como parámetro.

- createGeneralizationParentProperty2Column

  - Parte declarativa: genera una columna y las trazas necesarias a partir del elemento AddModelElement pasado como parámetro que debe contener un elemento Generalization.

  - Parte procedural: busca en el modelo de trazas el elemento tabla a partir de la referencia general del elemento origen generalization para añadir la columna. Transforma el tipo de datos asociado.

- createGeneralizationChildProperty2Column

  - Parte declarativa: genera una columna y las trazas necesarias a partir del elemento AddModelElement pasado como parámetro que debe contener un elemento Generalization.

  - Parte procedural: busca en el modelo de trazas el elemento tabla a partir de la referencia specific del elemento origen generalization para añadir la columna. Transforma el tipo de datos asociado.

Librería para la sincronización

Al igual que en la transformación de UML a Base de Datos, la transformación para mantener los modelos sincronizados también posee una librería auxiliar donde se encuentran los helpers con los que comprobar cuál de las reglas se debe ejecutar.

- isRemoved

  - Tipo: TraceLink

  - Valor de retorno: Boolean

  - Descripción: Comprueba si debe borrarse una traza por efecto del borrado de sus elementos.

- isRemoved

  - Tipo: TraceLinkEnd

- Valor de retorno: Boolean
- Descripción: Comprueba si debe borrarse un elemento de una traza por efecto del borrado del elemento que referencia.

- isRemoved
  - Tipo: ElementRef
  - Valor de retorno: Boolean
  - Descripción: Comprueba si debe borrarse una referencia por efecto del borrado de la referencia que contiene.

- isRemoved
  - Tipo: Eobject
  - Valor de retorno: Boolean
  - Descripción: Comprueba si debe borrarse un elemento por contener un elemento de tipo RemoveModelElement en el modelo de diferencias.

- isAbstract
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una clase abstracta.

- isPackageWithElems
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es un paquete que contiene elementos.

- isClass
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una clase.

- isProperty
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una propiedad.
- isClassOrAssociationClassProperty
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una propiedad y su padre es una clase o una clase asociación.
- isDataTypeProperty
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una propiedad y su padre es un elemento DataType.
- isGeneralization
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una generalización.
- isAssociation
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una asociación.
- isComposition
  - Tipo: AddModelElement
  - Valor de retorno: Boolean

- Descripción: Comprueba si la propiedad rightElement es una agregación/composición.

- isDataTypeAssociation
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una asociación y alguno de sus extremos es un elemento DataType.

- isEnumerationAssociation
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una asociación y alguno de sus extremos es un elemento Enumeration.

- isAssociationClass
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es una clase asociación.

- isBound
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement presenta la cardinalidad de una propiedad.

- isBound
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento Property pertenece a una asociación.

- isMultivaluated

- Tipo: AddModelElement
- Valor de retorno: Boolean
- Descripción: Comprueba si la propiedad rightElement que presenta una propiedad tiene cardinalidad uno a muchos/muchos a muchos.

- isMultivaluated
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento Property tiene cardinalidad uno a muchos/muchos a muchos.
- isOneToOne
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement que presenta una asociación tiene cardinalidad uno a uno.
- isManyToMany
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement que presenta una asociación tiene cardinalidad muchos a muchos.
- isDataType
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo DataType.
- isDataTypeReference
  - Tipo: AddModelElement
  - Valor de retorno: Boolean

- Descripción: Comprueba si la propiedad rightElement es de tipo Property y su propiedad Type es un DataType.

- isDataTypeReference
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad Type del elemento Property es un DataType.

- isDataTypePropertyToOne
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Property, su propiedad Type es un DataType y su cardinalidad es = 1.

- isDataTypePropertyToMany
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Property, su propiedad Type es un DataType y su cardinalidad es > 1.

- isDataTypeAssociationToOne
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Association, alguno de sus extremos es un DataType y su cardinalidad es = 1.

- isDataTypeAssociationToMany
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Association, alguno de sus extremos es un DataType y su cardinalidad es > 1.

- isEnumeration
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Enumeration.
- isEnumerationReference
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Property y su propiedad Type es un Enumeration.
- isEnumerationReference
  - Tipo: Property
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad Type del elemento Property es un Enumeration.
- isEnumerationPropertyToOne
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Property, su propiedad Type es un Enumeration y su cardinalidad es = 1.
- is EnumerationPropertyToMany
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Property, su propiedad Type es un Enumeration y su cardinalidad es > 1.
- isEnumerationAssociationToOne
  - Tipo: AddModelElement

- Valor de retorno: Boolean
- Descripción: Comprueba si la propiedad rightElement es de tipo Association, alguno de sus extremos es un Enumeration y su cardinalidad es = 1.

- isEnumerationAssociationToMany
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Association, alguno de sus extremos es un Enumeration y su cardinalidad es > 1.

- existsDataTypeReference
  - Tipo: AddModelElement
  - Valor de retorno: Boolean
  - Descripción: Comprueba si la propiedad rightElement es de tipo Property, si la propiedad leftParent es de tipo DataType y si existe algún elemento Property cuya propiedad Type sea igual a la propiedad leftParent.

- transformOnlyParentTable
  - Tipo: Class
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento Class, que participa como hijo en una Generalización, tiene un patrón configurado con el valor *GeneralizationInto = OnlyParentTable.*

- transformOnlyChildTable
  - Tipo: Class
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento Class, que participa como padre en una Generalización, tiene un patrón configurado con el valor *GeneralizationInto = OnlyChildTable.*

- executeGeneralizationRule (ruleName)
  - Tipo: AddModelElement

- Parámetro: String
- Valor de retorno: Boolean
- Descripción: Comprueba si el elemento AddModelElement cumple el patrón de Generalización y ha sido configurado para transformarse de acuerdo a la regla ruleName. Si no tiene patrón configurado, comprueba si ruleName coincide con la regla por defecto, AllTables.

- executeParentOrChildTableRule (ruleName)
  - Tipo: AddModelElement
  - Parámetro: String
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento AddModelElement cumple el patrón de Generalización y ha sido configurado para transformarse de acuerdo a la regla ruleName. Si no tiene patrón configurado, comprueba si ruleName coincide con la regla por defecto, AllTables.
- executeParentTableRule (ruleName)
  - Tipo: AddModelElement
  - Parámetro: String
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento AddModelElement cumple el patrón de Generalización y ha sido configurado para transformarse de acuerdo a la regla ruleName. Si no tiene patrón configurado, comprueba si ruleName coincide con la regla por defecto, AllTables.
- executeChildTableRule (ruleName)
  - Tipo: AddModelElement
  - Parámetro: String
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento AddModelElement cumple el patrón de Generalización y ha sido configurado para transformarse de acuerdo a la regla ruleName. Si no tiene patrón configurado, comprueba si ruleName coincide con la regla por defecto, AllTables.

- executeAssociationRule (ruleName)
  - Tipo: AddModelElement
  - Parámetro: String
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento AddModelElement cumple el patrón de Associación y ha sido configurado para transformarse de acuerdo a la regla ruleName. Si no tiene patrón configurado, comprueba si ruleName coincide con la regla por defecto, Reference.
- executeCompositionRule (ruleName)
  - Tipo: AddModelElement
  - Parámetro: String
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento AddModelElement cumple el patrón de Agregación/Composición y ha sido configurado para transformarse de acuerdo a la regla ruleName. Si no tiene patrón configurado, comprueba si ruleName coincide con la regla por defecto, IntegrityConstraint.
- executeDataTypeRule (ruleName)
  - Tipo: AddModelElement
  - Parámetro: String
  - Valor de retorno: Boolean
  - Descripción: Comprueba si el elemento AddModelElement cumple el patrón de DataType y ha sido configurado para transformarse de acuerdo a la regla ruleName. Si no tiene patrón configurado, comprueba si ruleName coincide con la regla por defecto, Table.
- executeEnumerationRule (ruleName)
  - Tipo: AddModelElement
  - Parámetro: String
  - Valor de retorno: Boolean
- Descripción: Comprueba si el elemento AddModelElement cumple el patrón de

Enumeration y ha sido configurado para transformarse de acuerdo a la regla ruleName. Si no tiene patrón configurado, comprueba si ruleName coincide con la regla por defecto, Property.

4.3. Conclusiones

Como conclusión a este capítulo decir que en él se ha visto, todo el proceso que se ha de llevar a cabo para implementar las transformaciones entre modelos, y de manera detallada, cada una de las transformaciones ATL implementadas para un caso concreto; la transformación entre el catálogo de reglas y el modelo de configuración, la transformación entre el modelo UML y el modelo de Base de Datos y la transformación para mantener sincronizados ambos modelos.

Capítulo 5 – Editor de Configuraciones

Tras la implementación de la infraestructura genérica para dar soporte a la configuración de transformaciones se ha decidido crear un editor de configuraciones.

Este editor tiene como objetivo agilizar y hacer más intuitivo el proceso de edición de configuraciones por parte del usuario. Así pues, a continuación se presenta el diseño y la implementación del editor.

5.1. Diseño del Editor de Configuraciones

Este editor está basado en una infraestructura, denominada FEFEM (*Forms-based Editors Framework for Ecore Models*), que, como su nombre indica, permite crear editores basados en formularios. Por ello, cada uno de los elementos del formulario obtiene su funcionalidad básica extendiendo alguna de las clases del proyecto FEFEM. Esta infraestructura queda fuera del alcance de este trabajo, por ello el diseño e implementación se pueden encontrar dentro del proyecto (*es.cv.gvcase.fefem.common*) en la misma página web del proyecto MOSKitt.

Así pues, el diseño o estructura para el editor de configuraciones es el siguiente:

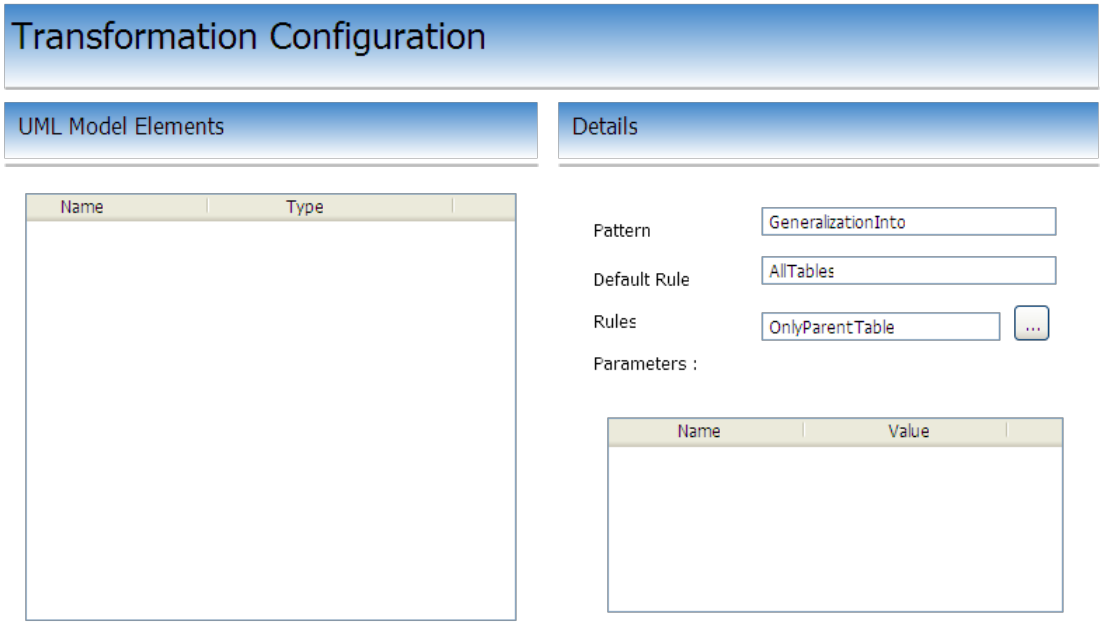


Figura 11. Diseño del editor de configuraciones

En la parte izquierda del editor aparece una tabla o lista con los patrones

configurados y en la parte derecha, para cada elemento seleccionado de la lista de la izquierda, aparecerán los detalles de sus atributos o referencias dependiendo de la funcionalidad que se le quiera dar al editor.

A continuación se detallan cada una de las partes:

- Tabla de elementos Configurados
  - Tipo: **EMFContainedCollectionEditionComposite**
  - Contenido: Cada uno de los elementos ConfiguredPattern del modelo de configuración.
  - Columnas:
    - "Nombre"
      - contenido: propiedad "name" del elemento EObject referenciado por la relación firstElement del elemento ConfiguredPattern.
      - decorador: ninguno
      - editable: no
    - "Tipo"
      - contenido: propiedad "type" del elemento EObject referenciado por la relación firstElement del elemento ConfiguredPattern.
      - decorador: icono del elemento UML referenciado.
      - editable: no
    - "Patrón"
      - contenido: propiedad "name" del elemento Pattern referenciado por la relación pattern del elemento ConfiguredPattern.
      - decorador: ninguno
      - editable: no

- Detalles de los elementos configurados
  - Tipo: **EMFEObjectComposite**

- Contenido: muestra las propiedades de los elementos relacionados con el elemento ConfiguredPattern seleccionado.
- Identificador: ninguno

- Patrón Configurado
  - Tipo: **EMFPropertyEReferenceComposite**
  - Contenido: referencia "pattern" del elemento configuredPattern. Muestra la propiedad "name" del elemento Pattern referenciado.
  - editable: no

- Regla por defecto
  - Tipo: **EMFPropertyEReferenceComposite**
  - Contenido: referencia "defaultRule" del elemento configuredPattern. Muestra la propiedad "name" del elemento Rule referenciado.
  - editable: no

- Regla relacionada
  - Tipo: **EMFPropertyEReferenceComposite**
  - Contenido: referencia "rule" del elemento configuredPattern. Muestra la propiedad "name" del elemento rule referenciado.
  - editable: sí, para cada patrón configurado, aparecerá la lista de reglas seleccionables.

- Tabla de Parámetros Configurados
  - Tipo: **EMFContainedCollectionEditionComposite**
  - Contenido: propiedad "parameterValue" del elemento ConfiguredPattern seleccionado.
  - Columnas:
    - "Name"
      - contenido: propiedad "name" del elemento Parameter referenciado por la relación parameterValue del elemento ConfiguredPattern.
      - decorador: ninguno

- editable: no

- "Valor"

  - contenido: propiedad "value" del elemento ParameterValue referenciado por la relación parameterValue del elemento ConfiguredPattern.

  - decorador: ninguno

  - editable: sí

5.2. Implementación del Editor de Configuraciones

Tal y como se presentó en la introducción, la herramienta MOSKitt está basada en un conjunto de módulos relacionados entre sí. Para el caso del editor de configuraciones, dentro del módulo *es.cv.gvcase.modelsync*, también se encuentra la infraestructura para la configuración y el editor (*es.cv.gvcase.modelsync.configuration.**).

Dentro del proyecto que contiene la implementación del editor de configuraciones (*es.cv.gvcase.modelsync.configuration.formseditor*) la funcionalidad se encuentra estructurada de la siguiente manera:

- Carpeta: src

  - Paquete: *es.cv.gvcase.modelsync.configuration.formseditor*

    - Hereda la funcionalidad de la clase FEFEMEditor de la infraestructura genérica y contiene el editor de formularios.

  - Paquete: *es.cv.gvcase.modelsync.configuration.formseditor.cellModifiers*

    - En el se encuentran cada una de las clases cuya función es la de poder modificar el contenido de las celdas de las tablas del editor.

  - Paquete: *es.cv.gvcase.modelsync.configuration.formseditor.composites.fefem*

    - Contiene cada uno de los composites que heredan su funcionalidad de la infraestructura genérica:

      - TableOfElementsComposite

        - Extiende a EMFContainedCollectionEditionComposite

        - Contiene la tabla donde se muestran los elementos ConfiguredPattern del modelo de configuración.

- ElementsDetailsComposite
    - Extiende a EMFEObjectComposite
    - Contiene los detalles del elemento ConfiguredPattern seleccionado.
- PatternComposite
    - Extiende a EMFPropertyEReferenceComposite
    - Contiene la referencia "pattern" del elemento ConfiguredPattern seleccionado y la presenta al usuario dentro de la sección de detalles.
- DefaultRuleComposite
    - Extiende a EMFPropertyEReferenceComposite
    - Contiene la referencia "defaultRule" del elemento ConfiguredPattern seleccionado y la presenta al usuario dentro de la sección de detalles.
- RulesComposite
    - Extiende a EMFPropertyEReferenceComposite
    - Contiene la referencia "rule" del elemento ConfiguredPattern seleccionado y la presenta al usuario dentro de la sección de detalles.
- ParametersComposite
    - Extiende a EMFContainedCollectionEditionComposite
    - Contiene la tabla donde se muestran los elementos ParameterValue del elemento ConfiguredPattern seleccionado.

- Paquete: *es.cv.gvcase.modelsync.configuration.formseditor.pages*
    - Hereda la funcionalidad de FEFEMPage que contiene la página principal a mostrar dentro del editor de formularios.
- Paquete: *es.cv.gvcase.modelsync.configuration.formseditor.providers*
    - Contiene las clases con la información a mostrar dentro de las celdas de las tablas.
- Paquete: *es.cv.modelsync.configuration.formseditor.sorters*
    - Contiene las clases que se encargan de ordenar los elementos de las celdas de las tablas de acuerdo a cierta propiedad del elemento.

En el anexo 3, se encuentra detalladamente cada una de las clases, propiedades y métodos del editor de configuraciones.

Siguiendo el ejemplo del capítulo anterior y haciendo uso del modelo de configuración ya creado "*Figura 9. Modelo de Configuración*" vamos a utilizar el editor que hemos presentado para editar dicho modelo.

Así pues, solamente hay que abrir el modelo con el nuevo editor, para ello, nos situamos sobre el modelo de configuración y hacemos clic con el botón derecho > *Open with* > *Configuration Multi-page Editor*.

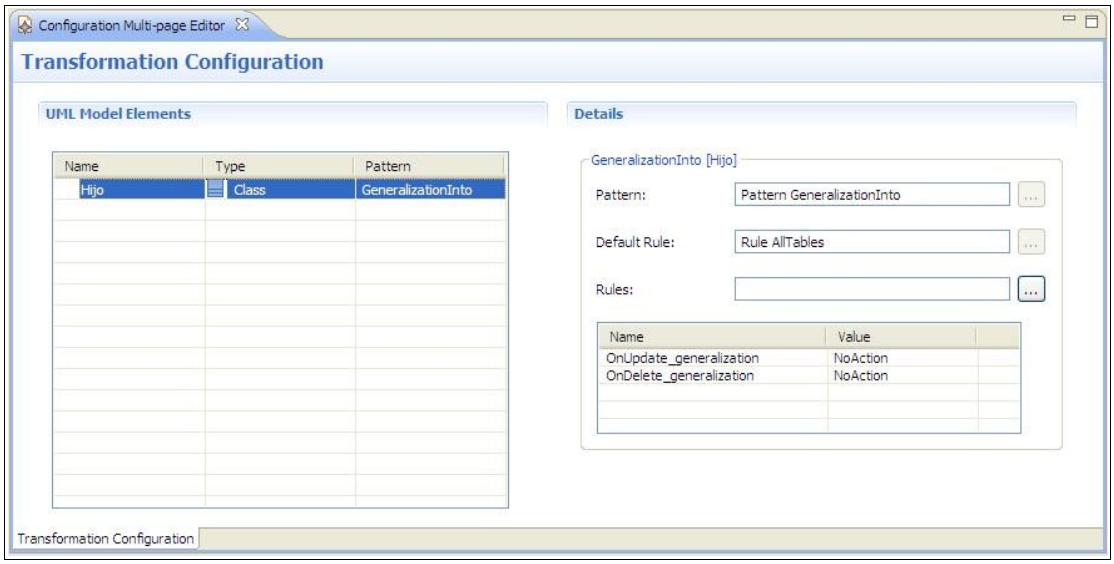


Figura 12. Editor de configuraciones

En él podemos observar, en la parte izquierda del editor los patrones configurados, y al seleccionar cualquier patrón configurado, en la parte derecha aparecen las propiedades del mismo, tanto las que se pueden editar (*Rules* y *Parameters*) como las que no (*Pattern* y *DefaultRule*).

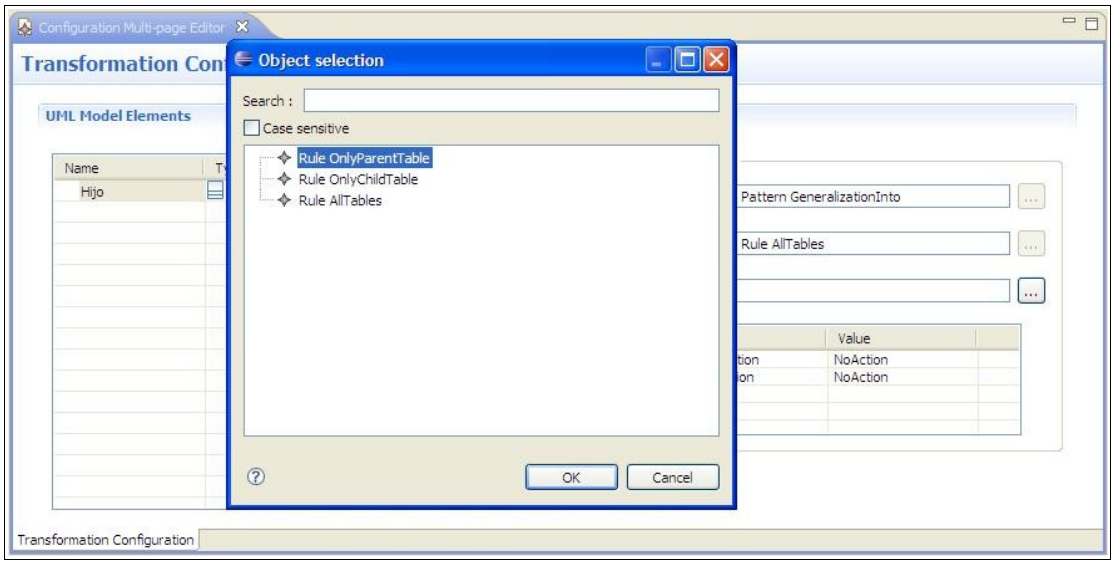


Figura 13. Selección de reglas

De las propiedades editables, por un lado es posible seleccionar la regla que el usuario crea oportuna (*Figura 13*), y por otro, para los parámetros definidos se les puede dar valor editando el campo de texto (*Figura 14*).

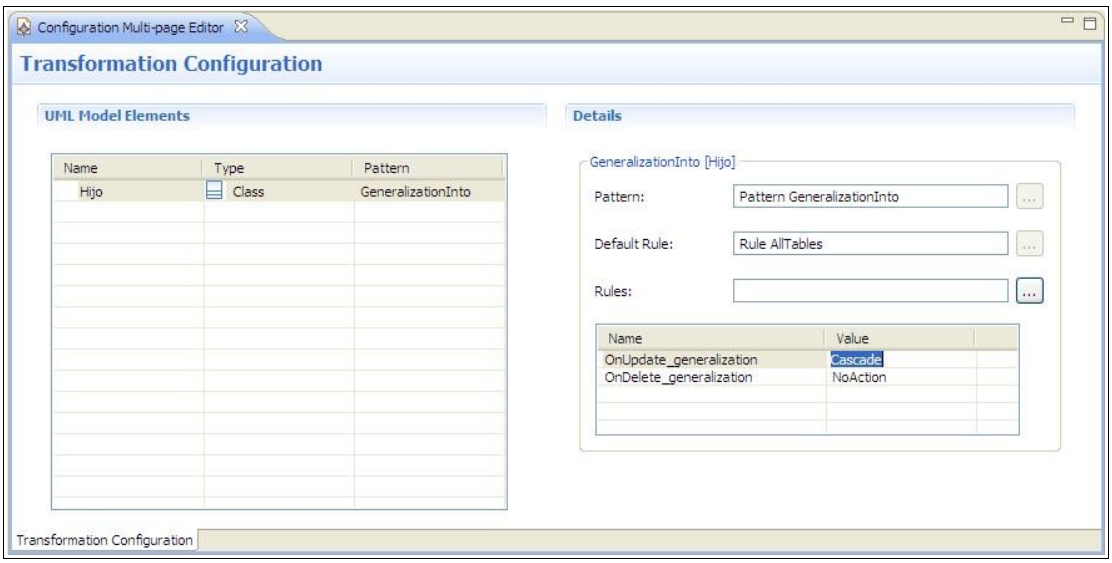


Figura 14. Edición de los parámetros

5.3. Conclusiones

En este capítulo se ha presentado el diseño y la implementación de un editor basado en formularios y cuya funcionalidad básica se obtiene de una infraestructura para la creación de editores.

Lo que se persigue es homogeneizar cada uno de los editores basados en formularios para que, con más o menos detalles, todos tengan la mismas características gracias a la infraestructura genérica. Además también se consigue hacer más intuitiva la edición de la configuración por parte del usuario.

Capítulo 6 – Configuración de transformaciones. Del diagrama de clases UML al modelo de Base de Datos

Tras la presentación del trabajo realizado, a continuación se muestra un breve ejemplo de como crear una configuración y lanzar las transformaciones configuradas para obtener un modelo de Base de Datos a partir de un modelo UML basado en un modelo real de un proyecto de la CIT.

Los pasos a seguir para crear la configuración y lanzar las transformaciones son:

1. Creación del un modelo del catálogo de reglas de acuerdo al metamodelo (UML) de entrada en la transformación configurable.

2. Creación de un modelo UML basado en un caso real.

3. Creación/generación del modelo de configuración a partir del modelo del catálogo de reglas y el modelo UML.

4. Transformación del modelo de UML al modelo de Base de Datos de acuerdo al modelo de Configuración.

    1. Transformación usando el modelo de configuración por defecto.

    2. Transformación tras modificar el modelo de configuración.

6.1. Creación del modelo de catálogo de reglas

Para crear el modelo de catálogo de reglas debemos saber, por un lado el modelo sobre el cuál vamos a realizar las transformaciones, en este caso el metamodelo UML y por otro, la configuración que hay que aplicar a cada elemento transformable, ya definida también en el apartado "requisitos para la transformación de modelos UML a Base de Datos" del tercer capítulo.

De manera general, los elementos que aparecen en las tablas tienen el siguiente significado:

La propiedad *DefaultRule* indica la regla que se ejecutará por defecto si el usuario no selecciona ninguna en el instante de la configuración.

Los elementos *Rule* indican las posibles reglas que el usuario va a poder seleccionar en el momento de configuración de la transformación.

Por otro lado, el elemento *Pattern* y sus correspondientes elementos *Instance* están haciendo referencia al patrón que deben cumplir los elementos dentro del modelo UML para que se puedan configurar de acuerdo a este patrón.

Por último, los elementos *Parameter*, configuran parámetros para cada patrón. En este caso, los parámetros que el usuario puede configurar son las propiedades '*On Update*' y '*On Delete*' del elemento Foreign Key del modelo de Base de Datos, si en dicho patrón, los elementos del modelo destino implican la creación de un elemento Foreign Key.

Así pues, los elementos del modelo de catálogo de regla son los siguientes:

Patrón de configuración para el elemento UML ***Generalization***

DefaultRule	*AllTables*		
Rule	*AllTables*		
Rule	*OnlyParentTable*		
Rule	*OnlyChildTable*		
Pattern	*GeneralizationInto*		
	Instance	*specificClass*	
		Type	*Class*
		ReferenceInstanceParameter	*generalization: Generalization*
			Instance generalization
	Instance	*generalClass*	
		Type	*Class*
	Instance	*Generalization*	
		Type	*Generalization*
		ReferenceInstanceParameter	*general: Classifier*
			Instance generalClass
Parameter	*OnUpdate_generalization*		
Parameter	*OnDelete_generalization*		

Tabla 3. Patrón de configuración para el elemento UML Generalization

Patrón de configuración para el elemento ***Association***

DefaultRule	*Reference*		
Rule	*Reference*		
Rule	*Table*		
Pattern	*AssociationInto*		
	Instance	*association*	
		Type	*Association*
		ReferenceInstanceParameter	*memberEnd: Property*

			Instance src_association
		ReferenceInstanceParameter	*memberEnd: Property*
			Instance dst_association
	Instance	*src_association*	
		Type	*Property*
		AttributeInstanceParameter	*aggregation: AggregationKind*
			none
		ReferenceInstanceParameter	*association: Association*
			Instance association
	Instance	*dst_association*	
		Type	*Property*
		AttributeInstanceParameter	*aggregation: AggregationKind*
			none
		ReferenceInstanceParameter	*association: Association*
			Instance association
Parameter	*OnUpdate_association*		
Parameter	*OnDelete_association*		

Tabla 4. Patrón de configuración para el elemento Association

Patrón de configuración para el elemento ***Aggregation***

DefaultRule	*IntegrityConstraint*		
Rule	*IntegrityConstraint*		
Rule	*Reference*		
Rule	*Table*		
Pattern	*CompositionAs*		
	Instance	*Composition*	
		Type	*Association*
		ReferenceInstanceParameter	*memberEnd: Property*
			Instance src_composition
		ReferenceInstanceParameter	*memberEnd: Property*
			Instance dst_composition
	Instance	*src_composition*	
		Type	*Property*
		AttributeInstanceParameter	*aggregation: AggregationKind*

			none
		ReferenceInstanceParameter	*association: Association*
			Instance composition
	Instance	*dst_composition*	
		Type	*Property*
		AttributeInstanceParameter	*aggregation: AggregationKind*
			composite
		ReferenceInstanceParameter	*association : Association*
			Instance composition
Parameter	*OnUpdate_composition*		
Parameter	*OnDelete_composition*		

Tabla 5. Patrón de configuración para el elemento Aggregation

Patrón de configuración para el elemento ***DataType***

DefaultRule	*Table*		
Rule	*Table*		
Rule	*Property*		
Pattern	*DatatypeAs*		
	Instance	*dataType*	
		Type	*DataType*
Parameter	*OnUpdate_dataType*		
Parameter	*OnDelete_dataType*		

Tabla 6. Patrón de configuración para el elemento DataType

Patrón de configuración para el elemento ***Enumeration***

DefaultRule	*Property*		
Rule	*Property*		
Rule	*Table*		
Pattern	*EnumerationAs*		
	Instance	*enumeration*	
		Type	*Enumeration*
Parameter	*OnUpdate_enumeration*		
Parameter	*OnDelete_enumeration*		

Tabla 7. Patrón de configuración para el elemento Enumeration

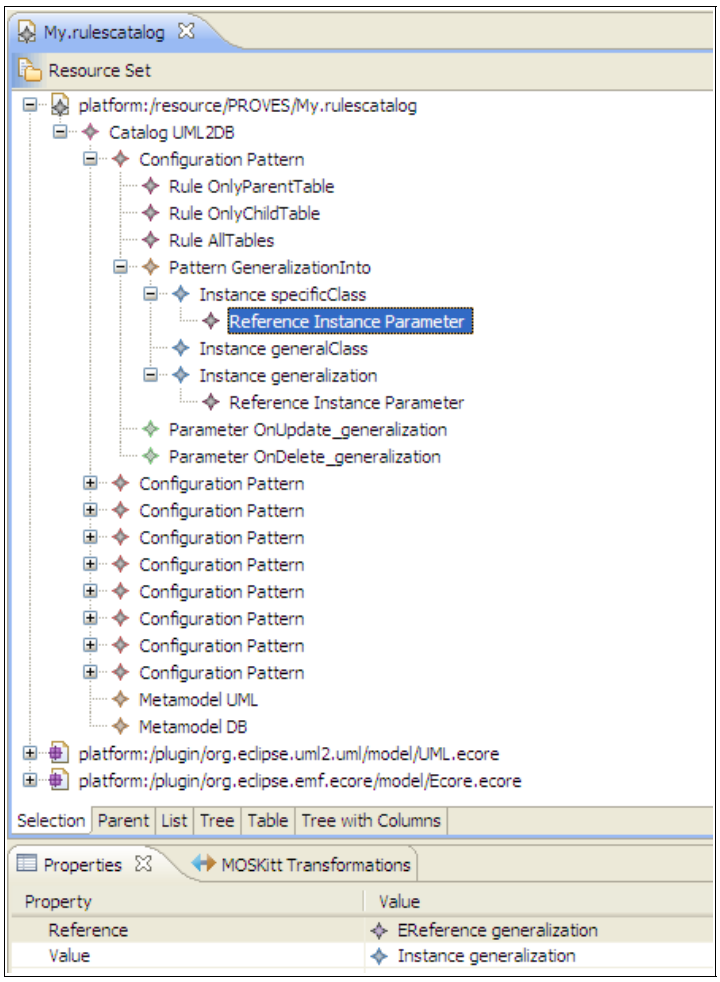


Figura 15. Modelo del catálogo de reglas

6.2. Creación de un modelo UML para la gestión de permisos de la CIT

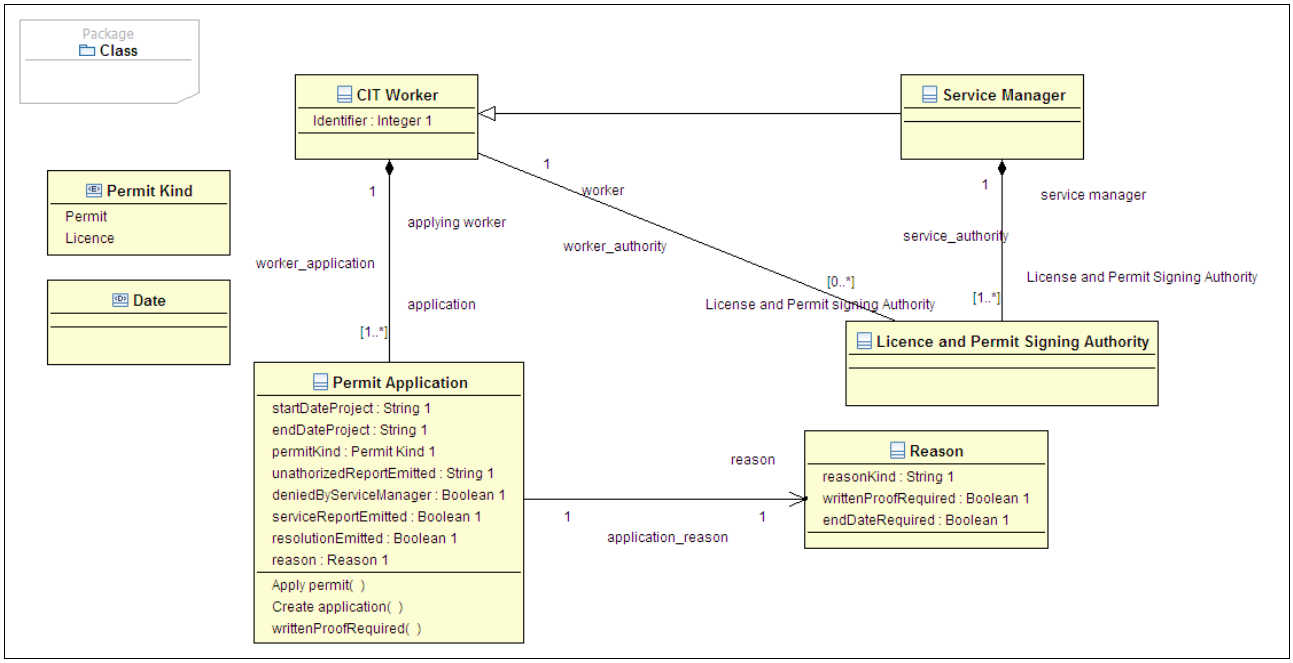


Figura 16. Modelo UML para la gestión de permisos

Sobre este modelo se pueden observar los distintos elementos UML que serán transformados:

- Elemento *Generalization* entre las clases 'CIT Worker' (elemento padre) y 'Service Manager' (elemento hijo).
- Elemento *Aggregation* entre la clase 'CIT Worker' y la clase 'Permit Application'.
- Elemento *Aggregation* entre la clase 'Service Manager' y la clase 'License and Permit Signing Authority'.
- Elemento *Association* entre las clases 'CIT Worker' y 'License and Permit Signing Authority'.
- Elemento *Association* entre las clases 'Permit Application' y 'Reason'.
- Elemento *DataType* nombrado como 'Date'.
- Elemento *Enumeration* cuya nombre es 'Permit Kind'.

6.3. Creación/generación del modelo de configuración a partir del modelo del catálogo de reglas y el modelo UML

Mediante el gestor de transformaciones de ATL incluido en ECLIPSE y con los modelos del catálogo de reglas y el modelo UML se genera el correspondiente modelo de configuración de transformaciones sobre el cuál el usuario tiene que decidir que reglas se aplica para cada patrón. La figura 17 muestra el editor con el modelo de configuración por defecto.

Para el primer patrón configurado podemos observar sus propiedades:

- Default Rule – Rule AllTables (regla que se ejecuta por defecto)
- First Element – Class Service Manager (es el elemento raíz del patrón)
- Pattern – GeneralizationInto (identificador del patrón configurado)
- Rule – a seleccionar por el usuario

Pero también los parámetros configurados:

- Parameter Value NoAction – cuya propiedad *parameter* hace referencia al elemento (Parameter OnUpdate_generalization) del modelo del catálogo y la propiedad *value* en la cuál el usuario da valor (Por defecto su valor es NoAction).
- Parameter Value NoAction – cuya propiedad *parameter* hace referencia al elemento (Parameter OnDelete_generalization) del modelo del catálogo y la propiedad *value* en la cuál el usuario da valor (Por defecto su valor es NoAction).

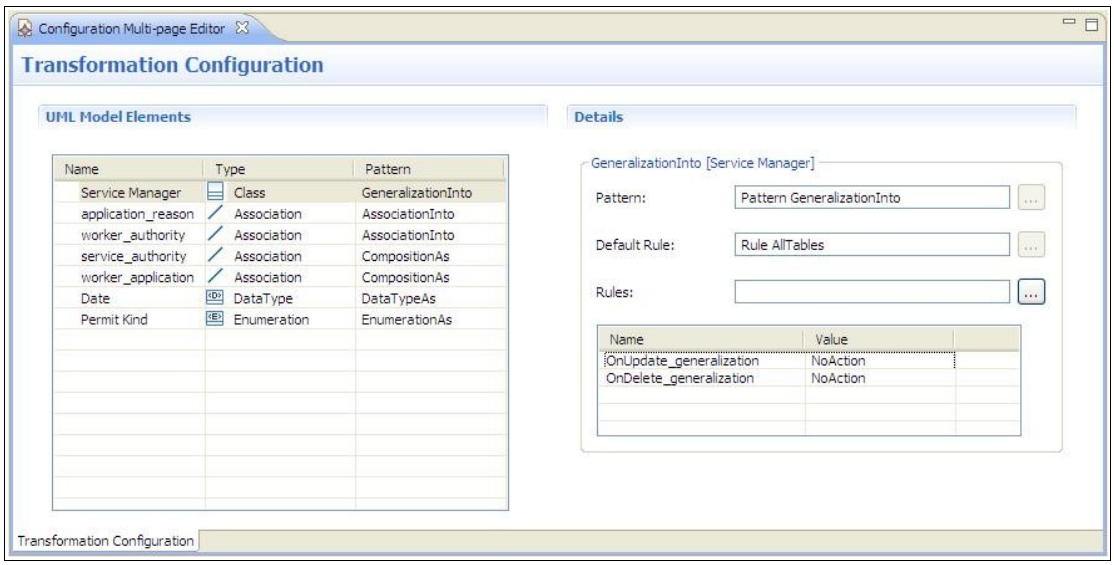


Figura 17. Modelo de configuración de transformaciones

6.4. Transformación del modelo de UML al modelo de Base de Datos

Para lanzar la transformación UML a Base de Datos se puede utilizar el gestor de transformaciones de MOSKitt. Par ello, hay que abrir la vista '*MOSKitt Transformations*' y seleccionar la entrada '*UML2 to Database Transformation*'.

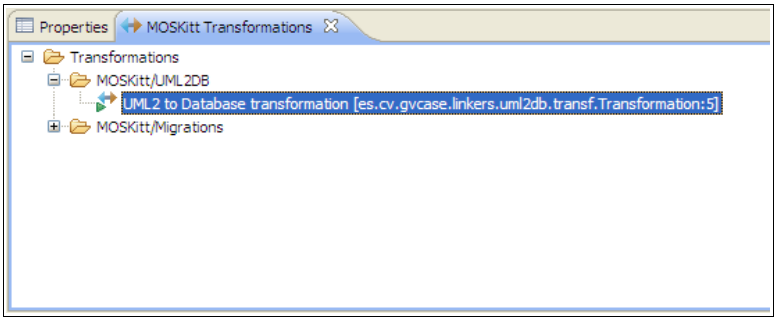


Figura 18. Vista para lanzar transformaciones en MOSKitt

A continuación, mediante el gestor de transformaciones se selecciona el modelo UML a transformar y el modelo de Base de Datos donde almacenar la transformación.

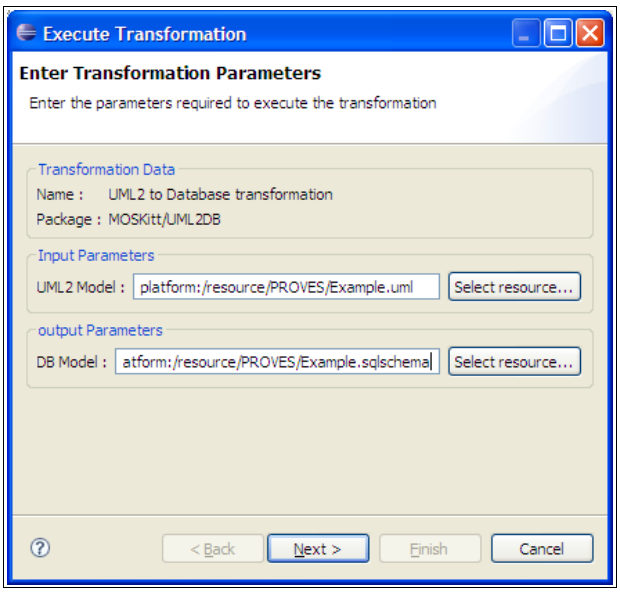


Figura 19. Gestor de transformaciones

Finalmente, se inicializa el diagrama a partir del modelo de Base de Datos generado, tal y como se indica en la figura 20, y se observa como resultado un modelo de Base de Datos de acuerdo a los requisitos para la transformación por defecto. Así pues:

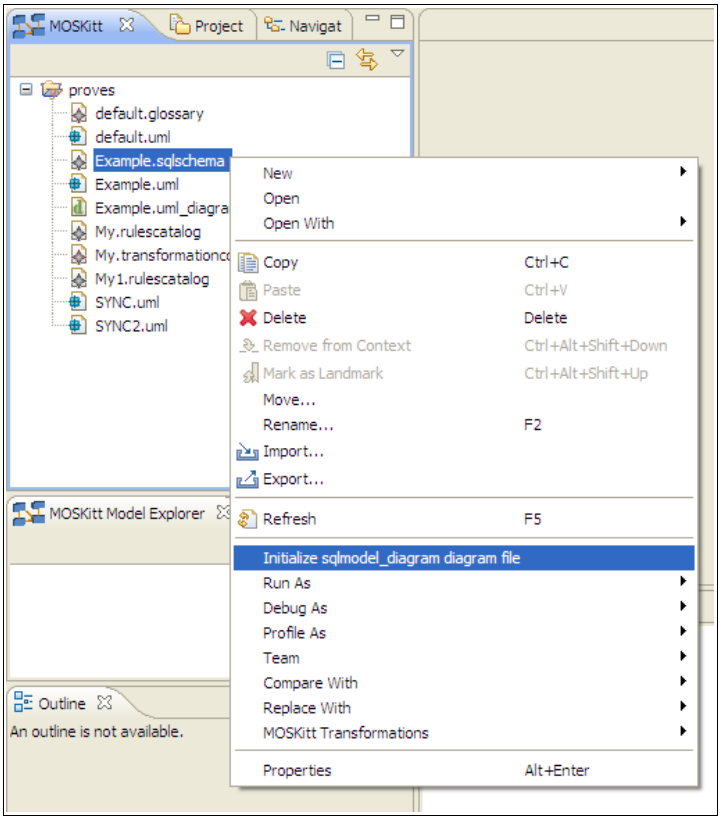


Figura 20. Inicialización del diagrama a partir del modelo de Base de Datos

- Para el elemento *Generalization* entre las clases 'CIT Worker' y 'Service Manager':
  - La clase 'CIT Worker' se transforma en una tabla con el mismo nombre.

- La clase 'Service Manager' se transforma en otra tabla 'Service Manager'.
- Y la generalización se transforma en una clave ajena (Foreign Key) dentro de la tabla 'Service Manager' que apunta a la tabla 'CIT Worker'.

- Para el elemento *Aggregation* entre las clases 'CIT Worker' y 'Permit Application':
  - La clase 'CIT Worker' se transforma en una tabla.
  - La clase 'Permit Application' también se transforma en una tabla.
  - La Agregación/composición se transforma en una clave ajena (Foreign Key) dentro de la tabla 'Permit Application' que apunta a la tabla 'CIT Worker'.
- Para el elemento *Aggregation* entre la clase 'Service Manager' y la clase 'License and Permit Signing Authority':
  - La clase 'Service Manager' se transforma en una tabla.
  - La clase 'License and Permit Signing Authority' también se transforma en una tabla.
  - La Agregación/composición se transforma en una clave ajena (Foreign Key) dentro de la tabla 'License and Permit Signing Authority' que apunta a la tabla 'Service Manager'.
- Para el elemento *Association* entre las clases 'CIT Worker' y 'License and Permit Signing Authority':
  - La clase 'CIT Worker' se transforma en una tabla.
  - La clase 'License and Permit Signing Authority' también se transforma en una tabla.
  - La Asociación se transforma en una clave ajena (Foreign Key) dentro de la tabla 'License and Permit Signing Authority' que apunta a la tabla 'CIT Worker'.
- Para el elemento *Association* entre las clases 'Permit Application' y 'Reason':
  - La clase 'Permit Application' se transforma en una tabla.
  - La clase 'Reason' también se transforma en una tabla.
  - La Asociación se transforma en una clave ajena (Foreign Key) dentro de la tabla 'Permit Application' que apunta a la tabla 'Reason'.
- Para el elemento *DataType* nombrado como 'Date' se transforma en una tabla con el mismo nombre.

- Por último, el elemento *Enumeration* cuya nombre es 'Permit Kind' no se transforma.

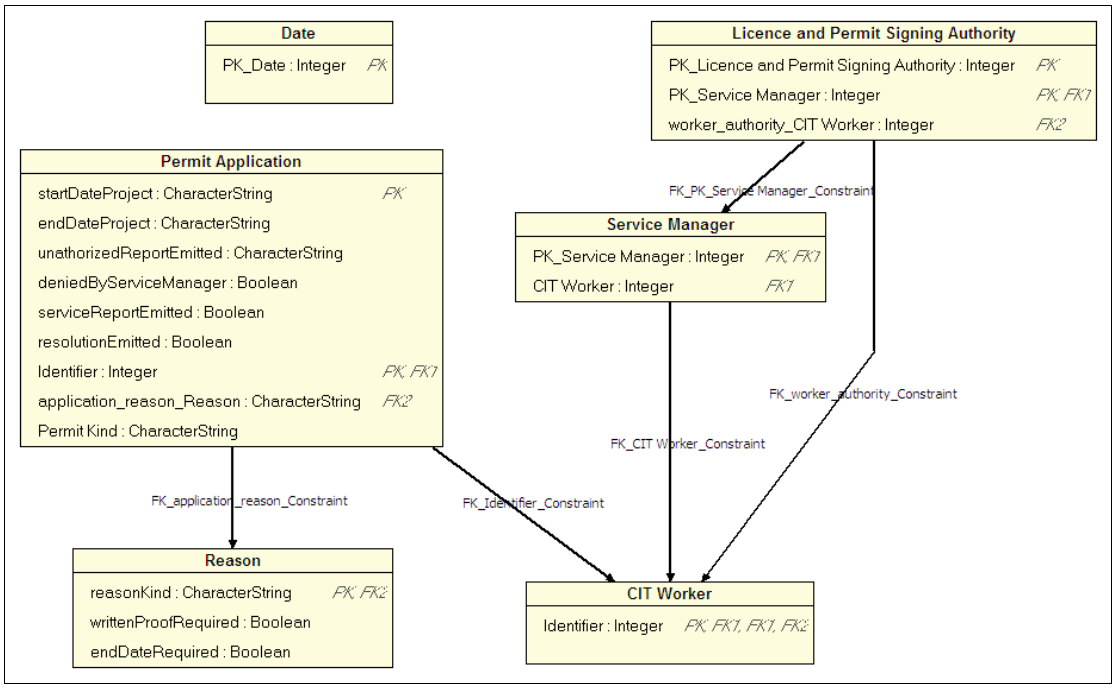


Figura 21. Modelo de Base de Datos con la configuración por defecto

6.4.1. Modificación de la configuración

Tras configurar el patrón de configuración GeneralizationInto para que ejecute la regla OnlyParentTable observamos que la tabla 'Service Manager' no se ha generado y las relaciones que tenia este elemento se le han asignado a la tabla 'CIT Worker', puesto que hemos configurado la transformación para que solamente se transforme el elemento padre de la generalización.

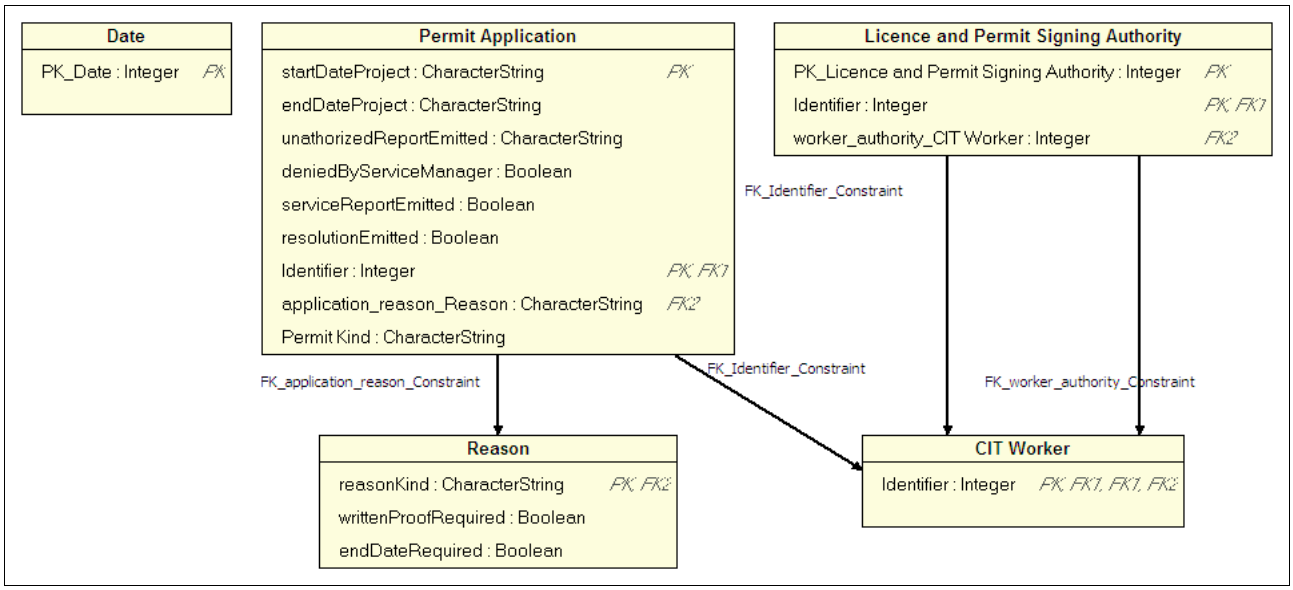


Figura 22. Modelo de Base de Datos tras modificar la configuración

6.5. Conclusiones

Como conclusiones a este capítulo, tenemos que decir que gracias a las transformaciones de modelos, el tiempo invertido en desarrollo de software es menor que en los casos en que los modelos son generados a mano.

Además, para el caso concreto de este trabajo, en el que el principal objetivo es configurar las transformaciones, observamos que la infraestructura desarrollada permite al usuario no solo obtener un modelo a partir de otro, sino que también decidir como quiere que sea el modelo de salida dependiendo de ciertos parámetros (requisitos del cliente, restricciones del modelo de salida, ...). Por último, decir también que el editor de configuraciones también agiliza y hace más fácil la edición de las configuraciones gracias a su interfaz más intuitiva que la del editor en forma de árbol.

Capítulo 7 – Conclusiones y trabajos futuros

En este trabajo del Máster en Ingeniería de Software, Métodos Formales y Sistemas de Información se ha presentado cómo se está abordando los conceptos de configuración de transformaciones, transformación de modelos y sincronización para la herramienta MOSKitt de la Consellería de Infraestructura y Transporte de la Generalitat Valenciana.

Esta propuesta presenta una infraestructura genérica para poder configurar transformaciones entre modelos, es decir, dar al usuario la libertad de elegir COMO se tienen que transformar los elementos de un modelo y EN QUE otros elementos se han de transformar. Como base para entender mejor esta propuesta, se ha utilizado una transformación definida anteriormente para este proyecto.

Toda esta configuración/transformación se ha podido desarrollar gracias a la tecnología basada en el concepto de Desarrollo de Software Dirigido por Modelos, como es EMF para el caso del metamodelo o ATL para el caso de las transformaciones.

Así pues, tanto la configuración por un lado como la transformación y sincronización por otro, permitirán al usuario de la herramienta MOSKitt agilizar en ciertas fases del proceso de desarrollo de software.

Además de todo este trabajo, se pretenden realizar distintas ampliaciones tales como :

- Modificar la *sincronización* en sentido *bidireccional*, es decir, que no solamente se modifique el modelo de Base de Datos frente a cambios en el modelo de UML2, sino que la sincronización se produzca tanto si hay cambios en un modelo como en el otro.

- Creación de un *editor* para el modelo de *catálogo de reglas* basándose también en la infraestructura FEFEM para hacer más intuitiva la edición de estos modelos por parte del usuario.

- Creación de *nuevas transformaciones* entre modelos, manteniendo también su sincronización y en algunos casos, utilizar la infraestructura creada en este trabajo para configurar dichas transformaciones.

Capítulo 8 – Bibliografía

[1] MOSKitt. MOdeling Software Kitt. http://www.moskitt.org/

[2] MDA. Model Driven Architecture http://www.omg.org/mda/

[3] UML 2.0 OCL Specification http://www.omg.org/docs/ptc/03-10-14.pdf

[4] Unified Modeling Language: Superstructure version 2.0
http://www.omg.org/docs/formal/05-07-04.pdf

[5] Atlas group LINA & INRIA. Atlas Transformation Language: ATL User Manual version 0.7
http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual%5Bv0.7%5D.pdf

[6] [EclipseOverview2003] (Febrero de 2003) "Eclipse Platform Technical Overview." Object Technology International, Inc. http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[7] [Budinsky2003] Budinsky, Frank. Steinberg, David. Merks, Ed. Ellersick, Raymond. J. Grose, Timothy (11 de Agosto de 2003). "Eclipse Modeling Framework: A Developer's Guide". Addison-Wesley, ISBN: 0-13-142542-0.

[8] [Gamma1995] Gamma, Erich. Helm, Richard. Johnson, Ralph. Vlissides, John (Año 1995). "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, ISBN: 0-20-163361-2.

[9] Emilio A. Sánchez, Gabriel Merín, Javier Muñoz. "Hacia un Marco Práctico de Evaluación de Tecnologías de Trasformación de Modelos en la Plataforma Eclipse".
http://www.sistedes.es/TJISBD/Vol-1/No-6/articles/dsdm-07-sanchez-M2M.pdf

[10] Roxana S. Giandini y Claudia F. Pons (2006). "Un Lenguaje para Transformación de Modelos basado en MOF y OCL".
http://www.lifia.info.unlp.edu.ar/papers/2006/Giandini2006.pdf

[11] Krzysztof Czarnecki, Simon Helsen. "Classification of Model Transformation Approaches". OOPSLA'03 Workshop on Generative Techniques in the Context of Model Driven Architecture.
http://www.swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf

Anexos

Anexo 1 – Diseño e implementación de las transformaciones de modelos UML2 con modelos de Base de Datos.

El objetivo de este documento de diseño es mostrar la infraestructura y el funcionamiento de los módulos (*es.cv.gvcase.linkers.uml2db.transf* y *es.cv.gvcase.linkers.uml2db.sync*) que contienen los mecanismos necesarios para generar la transformación entre modelos UML2 y modelos de BD, así como para mantener su sincronización dentro de la herramienta MOSKitt para la CIT.

Tareas y responsabilidades

- Proporcionar mecanismos para la generación de modelos de Base de Datos a partir de modelos UML2.
- Proporcionar mecanismos para la sincronización de los modelos derivados con los modelos originarios.

Tecnologías/Proyectos utilizados

- ATLAS Transformation Language (ATL)

  Especificación y ejecución de transformaciones entre modelos

- ANT TASKS FOR AMMA

  Conjunto de tareas para lanzar transformaciones ATL desde scripts ANT

Dependencias con otros componentes/módulos

- es.cv.gvcase.trmanager: Para registrar y ejecutar las distintas transformaciones.
- es.cv.gvcase.mdt.uml2: Para conocer el metamodelo de los modelos UML2.
- es.cv.gvcase.mdt.db: Para conocer el metamodelo de los modelos de Base de Datos.
- org.eclipse.emf.compare.diff: conocer el metamodelo de diferencias.

- es.cv.gvcase.modelsync.core: infraestructura de sincronización, metamodelo de trazas y modelos de diferencia.

Ficheros Producidos

- Modelos de BBDD (*.sqlschema)
- Modelos de trazabilidad (*.gvtrace)

Diseño

Organización y Estructura

es.cv.gvcase.linkers.uml2db.transf

- Tipo: Plug-in.
- No generado
- Responsabilidad: transformar un modelo de clases UML2 a un modelo de base de datos.
- Árbol de carpetas
  - src
    - Tipo: Source folder.
    - Responsabilidad: contiene el código fuente.
  - META-INF
    - Tipo: Folder
    - Responsabilidad: contiene el archivo MANIFEST.MF
- Dependencias
  - es.cv.gvcase.trmanager

es.cv.gvcase.linkers.uml2db.sync

- Tipo: Plug-in.
- No generado
- Responsabilidad: sincronizar modelos de clases UML2 con sus correspondientes modelos de base de datos
- Árbol de carpetas
  - src
    - Tipo: Source folder.
    - Responsabilidad: contiene el código fuente.
  - META-INF
    - Tipo: Folder
    - Responsabilidad: contiene el archivo MANIFEST.MF
- Dependencias
  - es.cv.gvcase.modelsync.core

Código Fuente es.cv.gvcase.linkers.uml2db.transf

- **UML2DBTransformation.java**
  - Extends:
    - Transformation
  - Carpeta: src
  - Paquete: es.cv.gvcase.linkers.uml2db.transf
  - Campos:
  - Métodos:
    - `public boolean inputsValid(HashMap<String, TransformedResource> inputs, List<String> errorList)`
      - `Override: no.`
      - `Funcionalidad: validar los modelos de entrada e informar del resultado de la validación al gestor de transformaciones.`

- **public boolean** transform(HashMap<String, TransformedResource> inputs, HashMap<String, TransformedResource> outputs, List<String> messageList)
  - Override: no.
  - Funcionalidad: obtener las rutas de los modelos de entrada y salida y llamar a la clase que realiza la transformacion.

- **ATL_Launcher.java**
  - Carpeta: src
  - Paquete: es.cv.gvcase.linkers.uml2db.transf.launcher
  - Campos:
    - **private** String inputPath
    - **private** String outputPath
    - **private** String confModel
    - **private** String traceModelName
    - **private** String outputModelName
    - **private** String inputModelName
    - **private final** String plugin = "es.cv.gvcase.linkers.uml2db.transf"
    - **private final** String buildPath = "es/cv/gvcase/linker/uml2db/transf/launcher"
  - Métodos:
    - **public** ATL_Launcher(String model, String output)
      - Override: no.
      - Funcionalidad: convertir las rutas de los modelos al formato requerido por ANT.
    - **public void** run()
      - Override: no.
      - Funcionalidad: inizializar y lanzar el script ANT que ejecuta la transformación.
    - **private void** setReadOnly(String trace)
      - Override: no.
      - Funcionalidad: convertir el fichero de trazas en modo solo lectura.
    - **private void** setWriteAllowed(String trace)
      - Override: no.
      - Funcionalidad: convertir el fichero de trazas en modo lectura/escritura.

- **catalog2configuration.atl**
    - Carpeta: src
    - Paquete: es.cv.gvcase.linkers.uml2db.transf.launcher.Transformations
    - Contenido:
        - Contiene las transformaciones entre los elementos del modelo del catálogo de reglas y el modelo para la configuración de transformaciones detalladas en el capítulo 4 "Implementación de transformaciones configuradas para un caso concreto"

- **uml2db.atl**
    - Carpeta: src
    - Paquete: es.cv.gvcase.linkers.uml2db.transf.launcher.Transformations
    - Contenido:
        - Contiene las transformaciones entre los elementos del modelo de UML2 y el modelo de Base de Datos también detalladas en el capítulo 4 "Implementación de transformaciones configuradas para un caso concreto"

- **uml2db_library.atl**
    - Carpeta: src
    - Paquete: es.cv.gvcase.linkers.uml2db.transf.launcher.Transformations
    - Contenido:
        - Contiene los helpers utilizados en la transformación uml2db.atl.

Código Fuente es.cv.gvcase.linkers.uml2db.sync

- **UML2DBSynchronizer.java**
    - Extends: Synchronizer
    - Carpeta: src
    - Paquete: es.cv.gvcase.linkers.uml2db.sync

- Campos:
  - `private final String plugin = "es.cv.gvcase.linkers.uml2db.sync"`
  - `private final String buildPath = "es/cv/gvcase/linkers/uml2db/sync"`

- Métodos:
  - `public boolean elementChanged(String source, String trace, String diff, String target, String project) {`
    - Override: no.
    - Funcionalidad: obtiene las rutas de los modelos lanza la transformación, elimina el modelo de diferencias y actualiza el entorno de trabajo.
  - `private void run(String source, String trace, String diff, String sql, String project)TransformedResource> outputs, List<String> messageList)`
    - Override: no.
    - Funcionalidad: inicializa y lanza el script ANT encargado de ejecutar las transformaciones ATL.

Anexo 2 – Diseño de la infraestructura del sincronizador de modelos.

El objetivo de este documento es mostrar la infraestructura genérica del módulo (*es.cv.gvcase.modelsync.core*) de sincronización entre modelos utilizado por el enlazador de modelos UML con modelos de Base de Datos en la herramienta MOSKitt.

Tareas y responsabilidades

- Registrar el metamodelo de trazas.
- Proporcionar el metamodelo de diferencias (gvdiff)
- Monitorizar el espacio de trabajo.
- Restablecer de la consistencia del modelo de trazas ante cambios provocados por el usuario.
- Informar de forma precisa y concisa a los componentes específicos, cuando sea necesario, para que sincronicen los cambios.
- Proporcionar un modelo de diferencias que contenga los cambios realizados en el modelo origen.

Tecnologías/Proyectos utilizados

- Eclipse (EMF)
    - Framework de modelado de la plataforma Eclipse

Funcionalidad Proporcionada

Proceso

El siguiente algoritmo resume el proceso que realiza el componente:

```
Si existe modelo de trazas asociado al recurso modificado:
        Si el tipo de cambio es eliminar:
                Eliminar el modelo de trazas asociado.
        Si el tipo de cambio es mover:
                Reconstruir las referencias afectadas en el modelo de trazas.
        Si el tipo de cambio es modificar:
                Obtener el modelo de diferencias del modelo origen.
                Lanzar el evento de sincronización al componente adecuado.
```

APIs utilizados

- EMF Compare

Datos de Salida

Si el tipo de cambio es la modificación de un modelo el componente proporciona los siguientes datos de salida:

- Modelo de trazas.
- Modelo de diferencias.
- Modelo antes del cambio.
- Modelo después del cambio.

Evento de Salida

Si el tipo de cambio es la modificación el componente dispara el evento de sincronización al componente adecuado.

Diseño

Organización y Estructura

- **es.cv.gvcase.modelsync.core**
  - Tipo: Plug-in.
  - No generado
  - Responsabilidad: proveer la infraestructura básica de sincronización entre modelos
  - Árbol de carpetas
    - src
      - Tipo: Source folder.
      - Responsabilidad: contiene el código fuente.
    - model
      - Tipo: Folder
      - Responsabilidad: contiene los metamodelos de trazas y diferencias.
    - META-INF
      - Tipo: Folder
      - Responsabilidad: contiene el archivo MANIFEST.MF
    - Schema
      - Tipo: Folder
      - Responsabilidad: contiene la especificación de la extensión que provee el plugin
  - Dependencias

- No tiene

Código Fuente

- **Start.java**
  - Interfaces:
    - IStartup
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.core
  - Campos:
    - `private final String plugin = "es.cv.gvcase.modelsync.core";`
    - `private final String location = "model";`
    - `private final String diffmmodel = "mmw_traceability.ecore";`
    - `private PostChange_ResourceChangeListener post_listener;`
  - Métodos:
    - `public void earlyStartup()`
      - Override: no.
      - Funcionalidad: realizar la llamada al método que registra el metamodelo de trazas y construir el escuchador del workspace que permitirá analizar los cambios.
    - `private void registerMetamodel()`
      - Override: no.
      - Funcionalidad: construir la url del metamodelos de trazas y crear la clase que lo registrará.
- **RegisterMetamodel.java**
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.core
  - Métodos:
    - `public RegisterMetamodel(String metamodel)`
      - Override: no.
      - Funcionalidad: Constructor de la clase y llama al método init.ç
    - `private void init(String URLmetaModel)`
      - Override: no.

        - Funcionalidad: toma el metamodelo de entrada y lo recorre registrándolo en el registro EMF.

    - `private Set getElementsByType(Resource extent,String type)`
        - Override: no.
        - Funcionalidad: Obtiene todos los elementos de un tipo dado de un modelo.

- **Synchronizer.java**

    - Carpeta: src

    - Paquete: es.cv.gvcase.modelsync.core.listener

    - Métodos:

        - `public abstract boolean elementChanged(String source, String trace, String diff, String targetModel, String project)`
            - Override: no.
            - Funcionalidad: define la clase y la signatura del método que deben ser extendidos por los plugin que se conecten a la extensión.

- **PostChange_ResourceChangeListener.java**

    - Interfaces:

        - IResourceChangeListener

    - Carpeta: src

    - Paquete: es.cv.gvcase.modelsync.core.listener

    - Métodos:

        - `public void resourceChanged(IResourceChangeEvent event)`
            - Override: no.
            - Funcionalidad: explorar el evento dado para obtener los cambios y según su naturaleza (modificar, elminar, mover) llamar a los métodos adecuados.

        - `private Resource getOldModel(IResourceDelta delta)`
            - Override: no.
            - Funcionalidad: obtener el recurso antes de su modificación a través del registro histórico de Eclipse.

        - `private Resource loadModel(IResourceDelta res)`
            - Override: no.
            - Funcionalidad: cargar el modelo asociado a un recurso.

        - `private void synchronizeDelete(IResource trace)`
            - Override: no.
            - Funcionalidad: eliminar el recurso dado del workspace.

- **private void** synchronizeMove(IResource trace, IResourceDelta delta)
  - Override: no.
  - Funcionalidad: cambiar el nombre del fichero que contiene las trazas para mantener la coherencia con los cambios de nombre de los modelos y llamar al método synchronizeTrace para actualizar sus referencias internas.

- **private void** synchronizeTrace(IResource trace, String srcname, String dstname
  - Override: no.
  - Funcionalidad: cargar el modelo de trazas y actualizar las referencias a modelos externas con su nuevo nombre.

- **private** IResource getTargetModel(IProject project, IResource trace)
  - Override: no.
  - Funcionalidad: obtener el modelo destino a partir del modelo de trazas en el contexto del proyecto.

- **private boolean** isRemoveOp(IResourceDelta delta)
  - Override: no.
  - Funcionalidad: comprueba si se trata de un evento de borrado.

- **private boolean** isMoveOp(IResourceDelta delta)
  - Override: no.
  - Funcionalidad: comprueba si se trata de un evento en el que se mueven los modelos.

- **private boolean** isModifyOp(IResourceDelta delta)
  - Override: no.
  - Funcionalidad: comprueba si se trata de un evento de modificación.

- **private** IResource getTrace(IProject project, String name)
  - Override: no.
  - Funcionalidad: busca el modelo de trazas asociado al modelo dado en el proyecto, si no existe devuelve nulo.

- **ListenerDesc.java**
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.core.listener.registry
  - Campos:
    - **private** String source;
    - **private** String target;
    - **private** String traceid;
    - **private** IConfigurationElement configElement;

- `private String class_;`
- `private String package_;`

- Métodos:
  - `public ListenerDesc(String source, String target, String trace,String class_)`
    - Override: no.
    - Funcionalidad: constructor de la clase, inicializa los campos con los parámetros dados.
  - `public void setConfigElement(IConfigurationElement configElement)`
    - Override: no.
    - Funcionalidad: modificador del valor del campo configElement.
  - `public IConfigurationElement getConfigElement()`
    - Override: no.
    - Funcionalidad: consultor del valor del campo configElement.
  - `public String getSource()`
    - Override: no.
    - Funcionalidad: consultor del valor del campo Source.
  - `public String getTarget()`
    - Override: no.
    - Funcionalidad: consultor del valor del campo Target.
  - `public String getTraceID()`
    - Override: no.
    - Funcionalidad: consultor del valor del campo traceid.
  - `public String getClass_)`
    - Override: no.
    - Funcionalidad: consultor del valor del campo class_.
  - `public String getPackage()`
    - Override: no.
    - Funcionalidad: consultor del valor del campo package_.
  - `public void setPackage(String package_)`
    - Override: no.
    - Funcionalidad: modificador del valor del campo package_.

- **ListenerRegistry.java**
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.core.listener.registry
  - Campos:
    - `private static String SOURCE_ATTRIBUTE = "Source";`

- **private static** String *TARGET_ATTRIBUTE* = "Target";
- **private static** String *TRACE_ATTRIBUTE* = "Trace";
- **private static** String *CLASS_ATTRIBUTE* = "class";
- **private static** String *PACKAGE_ATTRIBUTE* = "package";
- **private static final** String *extensionPointId* = "es.cv.gvcase.modelsync.core.Listener";

- Métodos:

  - **public static boolean** Synchronize(IResource trace, Resource old, Resource delta, IResource target)
    - Override: no.
    - Funcionalidad: recorrer el registro de plugins conectados al punto de extensión para obtener un manejador adecuado para la modificación del modelo dado junto con sus trazas.
  - **private static boolean** isSyncHandler(Resource source,IResource trace, ListenerDesc lsDesc)
    - Override: no.
    - Funcionalidad: comprobar si el plugin es un manejador válido para el modelo y las trazas dadas.
  - **public static** ArrayList<ListenerDesc> getAllListeners()
    - Override: no.
    - Funcionalidad: obtieen una lista de todos los plugins conectados al punto de extensión.
  - **private static** ArrayList<ListenerDesc> processExtension(IExtension extension)
    - Override: no.
    - Funcionalidad: convierte los plugin conectados en una lista de objetos de tipo listenerDesc.
  - **private static** DiffModel createDiffModel(Resource left, Resource right)
    - Override: no.
    - Funcionalidad: crea el modelo de diferencias del modelo origen.
  - **private static** String saveModel(EObject diff, URI uri)
    - Override: no.
    - Funcionalidad: persiste el modelo dado en la uri indicada.
  - **private static boolean** Synchronize(ListenerDesc lsdesc, IResource traceElem, Resource old, Resource delta, IResource target)
    - Override: no.
    - Funcionalidad: obtiene el modelo de diferencias, lo persiste, obteien las rutas de los modelos e inicializa el evento de sincronización del plugin manejador.

Anexo 3 – Diseño de la infraestructura para la configuración de transformaciones.

El objetivo de este documento es mostrar el diseño y la implementación del módulo (es.cv.gvcase.modelsync.configuration.*) en el que se encuentra la infraestructura necesaria para generar modelos de configuración. Además también se da soporte a la edición de los modelos mediante un editor de configuraciones basado en la infraestructura FEFEM. Con toda esta infraestructura se permite configurar las transformaciones, para el caso de la herramienta MOSKitt, entre modelos UML y modelos de Base de Datos.

Tareas y responsabilidades

- Proporcionar el metamodelo para el catálogo de reglas.
- Proporcionar el metamodelo para la configuración de transformaciones.
- Proporcionar un editor para la creación de ambos modelos.
- Proporcionar mecanismos para la edición mediante formularios del editor de configuración de transformaciones.

Tecnologías/Proyectos utilizados

- Eclipse (EMF)
  - framework de modelado de la plataforma Eclipse.
  - Uso del EMFDataBinding para la relación con la interfaz de usuario.
- Eclipse Forms
  - Desarrollo de la interfaz de usuario basada en formularios.

Dependencias con otros componentes

- es.cv.gvcase.fefem.common

Diseño

Organización y Estructura

- **es.cv.gvcase.modelsync.configuration**
    - Tipo: Plug-in.
    - No generado
    - Responsabilidad: proveer la infraestructura básica para la configuración de transformaciones entre modelos.
    - Árbol de carpetas
        - src
            - Tipo: Source folder.
            - Responsabilidad: contiene el código fuente.
        - model
            - Tipo: Folder
            - Responsabilidad: contiene los metamodelos del catalogo de reglas y de la configuración para las transformaciones y el genmodel para los editores.
        - META-INF
            - Tipo: Folder
            - Responsabilidad: contiene el archivo MANIFEST.MF

- **es.cv.gvcase.modelsync.configuration.edit**
    - Tipo: Plug-in.
    - Generado a partir de transformationConfiguration.genmodel.
    - Responsabilidad: proveer la infraestructura básica para construir modelos a partir de los metamodelos del catálogo de reglas y de la configuración de transformaciones.
    - Árbol de carpetas
        - src

        - Tipo: Source folder.
        - Responsabilidad: contiene el código fuente.
    - META-INF
        - Tipo: Folder
        - Responsabilidad: contiene el archivo MANIFEST.MF
    - icons
        - Tipo: Folder
        - Responsabilidad: contiene los iconos del editor.

- **es.cv.gvcase.modelsync.configuration.editor**
    - Tipo: Plug-in.
    - Generado a partir de transformationConfiguration.genmodel.
    - Responsabilidad: proveer la infraestructura básica para construir modelos a partir de los metamodelos del catálogo de reglas y de la configuración de transformaciones.
    - Árbol de carpetas
        - src
            - Tipo: Source folder.
            - Responsabilidad: contiene el código fuente.
        - META-INF
            - Tipo: Folder
            - Responsabilidad: contiene el archivo MANIFEST.MF
        - icons
            - Tipo: Folder
            - Responsabilidad: contiene los iconos del editor.

- **es.cv.gvcase.modelsync.configuration.formseditor**

- Tipo: Java
- Generado automáticamente
- Responsabilidad: implementación del editor de configuración de transformaciones basado en formularios.
- Árbol de carpetas
  - src
    - Tipo: Source Folder
    - Responsabilidad: contiene el código del editor.
  - icons
    - Tipo: Folder
    - Responsabilidad: contiene archivos de imágenes correspondientes a diversos iconos del editor.
  - META-INF
    - Tipo: Folder.
    - Responsabilidad: contiene el archivo MANIFEST.MF

Descripción del código

- **ParameterCellModifier**
  - Tipo: Cell modifier
  - Interfaz que implementa: org.eclipse.jface.viewers.ICellModifier
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.cellmodifiers
  - Modo de utilización: auxiliar.
  - Campos
    - `private es.cv.gvcase.fefem.common.FEFEMPage page;`

- `private static final String NAME_COLUMN = "Name";`

- Métodos

  - `public boolean canModify(Object element, String property)`

    - Override: Sí.

    - Funcionalidad: comprueba si la propiedad 'property' del elemento 'element' puede ser modificada por el cell modifier.

  - `public Object getValue(Object element, String property)`

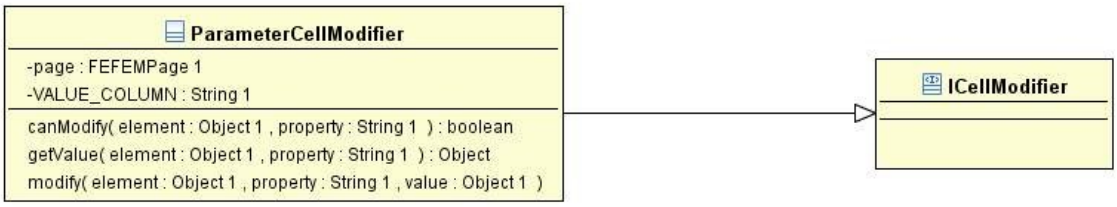
    - Override: Sí.

    - Funcionalidad: devuelve el valor de la propiedad 'property' para el elemento 'element'.

  - `public void modify(Object element, String property, Object value)`

    - Override: Sí.

    - Funcionalidad: Contiene el código necesario para modificar el valor de la propiedad 'property' para el elemento 'element'.



- **TableOfElementsComposite**

  - Tipo: Composite

  - Superclase: es.cv.gvcase.fefem.common.composites.EMFContainedCollectionEditionComposite

  - Carpeta: src

  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.composites.fefem

  - Modo de utilización: auxiliar.

  - Campos

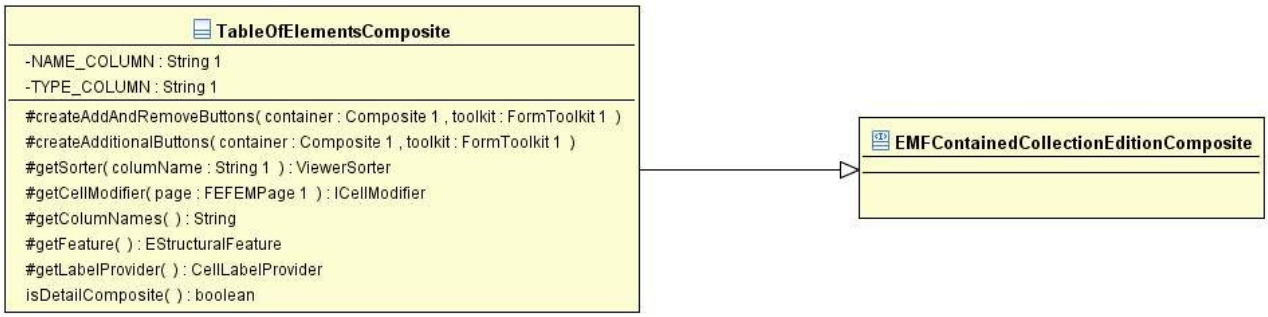
    - `private static final String NAME_COLUMN = "Name";`

- `private static final String TYPE_COLUMN = "Type";`

- Métodos

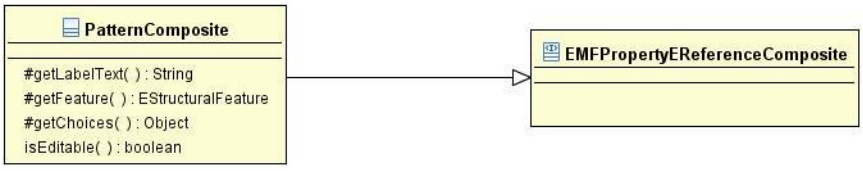
  - `protected void createAddAndRemoveButtons(Composite container, FormToolkit toolkit)`
    - Override: Sí.
    - Funcionalidad: Crea los botones que añadirán y eliminarán elementos de la tabla.
  - `protected void createAdditionalButtons(Composite container, FormToolkit toolkit)`
    - Override: Sí.
    - Funcionalidad: Crea botones adicionales si hace falta.
  - `protected ViewerSorter getSorter(String columName)`
    - Override: Sí.
    - Funcionalidad: Devuelve el sorter adecuado para la columna que recibe como parámetro.
  - `protected ICellModifier getCellModifier(FEFEMPage page)`
    - Override: Sí.
    - Funcionalidad: Devuelve un objeto ICellModifier para poder modificar la celda de la tabla.
  - `protected String[] getColumNames()`
    - Override: Sí.
    - Funcionalidad: Devuelve una lista con los nombres de las columnas.
  - `protected EStructuralFeature getFeature()`
    - Override: Sí.
    - Funcionalidad: Devuelve la feature correspondiente para mostrar en la tabla.
  - `protected CellLabelProvider getLabelProvider()`
    - Override: Sí.
    - Funcionalidad: Devuelve un objeto de tipo ConfigurationLabelProvider.
  - `public boolean isDetailComposite()`

- Override: Sí.
- Funcionalidad: Devuelve true o false si es un composite donde almacenar los detalles de la tabla o no.



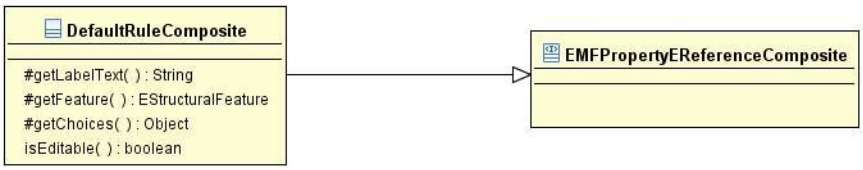
- ElementsDetailsComposite
  - Tipo: Composite
  - Superclase: es.cv.gvcase.fefem.common.composites.EMFEObjectComposite
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.composites.fefem
  - Modo de utilización: auxiliar.
  - Métodos
    - `protected EStructuralFeature getFeature()`
      - Override: Sí.
      - Funcionalidad: Devuelve una feature del modelo de configuración.
    - `protected String updatedGroupText()`
      - Override: Sí.
      - Funcionalidad: Devuelve un String que representa el nombre del Group a partir del elemento viewer del composite.

- PatternComposite
  - Tipo: Composite
  - Superclase: es.cv.gvcase.fefem.common.composites.EMFPropertyEReferenceComposite
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.composites.fefem
  - Modo de utilización: auxiliar.
  - Métodos
    - `protected String getLabelText()`
      - Override: Sí.
      - Funcionalidad: Devuelve la etiqueta a mostrar para el elemento Pattern.
    - `protected EStructuralFeature getFeature()`
      - Override: Sí.
      - Funcionalidad: Devuelve la feature Pattern asociada al elemento ConfiguredPattern del modelo de configuración.
    - `protected Object[] getChoices()`
      - Override: Sí.
      - Funcionalidad: Devuelve una lista con los objetos que se mostrarán en el seleccionable.
    - `public boolean isEditable()`
      - Override: Sí.
      - Funcionalidad: Devuelve true o false dependiendo si el composite es editable o no. En este caso devuelve false.



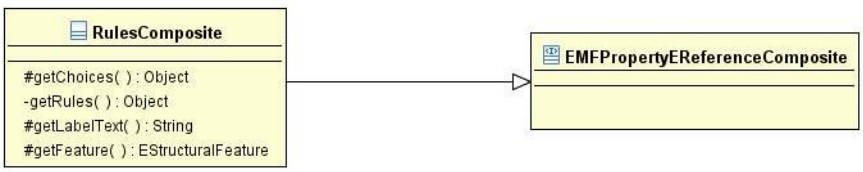
- **DefaultRuleComposite**

- Tipo: Composite
- Superclase: es.cv.gvcase.fefem.common.composites.EMFPropertyEReferenceComposite
- Carpeta: src
- Paquete: es.cv.gvcase.modelsync.configuration.formseditor.composites.fefem
- Modo de utilización: auxiliar.
- Métodos
  - `protected String getLabelText()`
    - Override: Sí.
    - Funcionalidad: Devuelve la etiqueta a mostrar para el elemento DefaultRule.
  - `protected EStructuralFeature getFeature()`
    - Override: Sí.
    - Funcionalidad: Devuelve la feature DefaultRule asociada al elemento ConfiguredPattern del modelo de configuración.
  - `protected Object[] getChoices()`
    - Override: Sí.
    - Funcionalidad: Devuelve una lista con los objetos que se mostrarán en el seleccionable.
  - `public boolean isEditable()`
    - Override: Sí.
    - Funcionalidad: Devuelve true o false dependiendo si el composite es editable o no. En este caso devuelve false.



- RulesComposite

- Tipo: Composite
- Superclase: es.cv.gvcase.fefem.common.composites.EMFPropertyEReferenceComposite
- Carpeta: src
- Paquete: es.cv.gvcase.modelsync.configuration.formseditor.composites.fefem
- Modo de utilización: auxiliar.
- Métodos
  - `protected String getLabelText()`
    - Override: Sí.
    - Funcionalidad: Devuelve la etiqueta a mostrar para la referencia rules del elemento ConfiguredPattern.
  - `protected EStructuralFeature getFeature()`
    - Override: Sí.
    - Funcionalidad: Devuelve la feature Rule asociada al elemento ConfiguredPattern del modelo de configuración.
  - `protected Object[] getChoices()`
    - Override: Sí.
    - Funcionalidad: Devuelve una lista con los objetos que se mostrarán en el seleccionable. Para ello utiliza el método getRules().
  - `private Object[] getRules()`
    - Override: No.
    - Funcionalidad: Devuelve la lista de reglas asociada al elemento ConfiguredPattern seleccionado en la tabla.

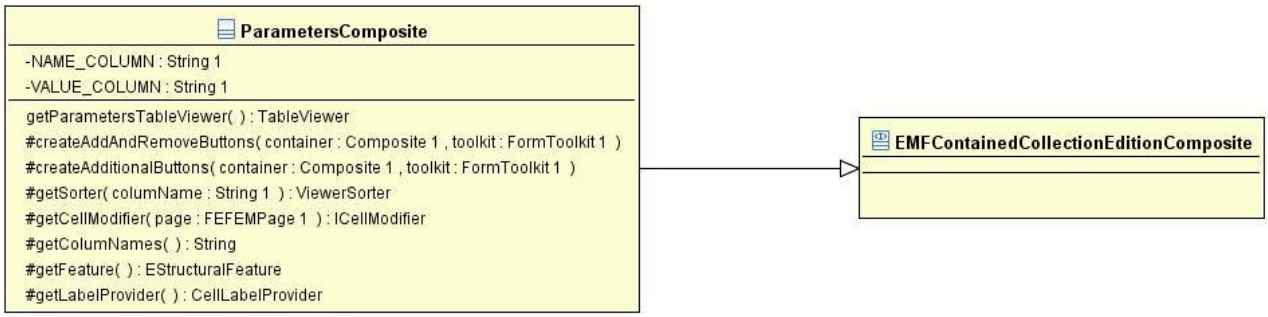


- **ParametersComposite**

- Tipo: Composite
- Superclase: es.cv.gvcase.fefem.common.composites.EMFContainedCollectionEditionComposite
- Carpeta: src
- Paquete: es.cv.gvcase.modelsync.configuration.formseditor.composites.fefem
- Modo de utilización: auxiliar.
- Campos
  - `private static final String NAME_COLUMN = "Name";`
  - `private static final String VALUE_COLUMN = "Value";`
- Métodos
  - `public TableViewer getParametersTableViewer()`
    - Override: No.
    - Funcionalidad: Devuelve el TableViewer asociado al composite.
  - `protected void createAddAndRemoveButtons(Composite container, FormToolkit toolkit)`
    - Override: Sí.
    - Funcionalidad: Crea los botones que añadirán y eliminarán elementos de la tabla.
  - `protected void createAdditionalButtons(Composite container, FormToolkit toolkit)`
    - Override: Sí.
    - Funcionalidad: Crea botones adicionales si hace falta.
  - `protected ViewerSorter getSorter(String columName)`
    - Override: Sí.
    - Funcionalidad: Devuelve el sorter adecuado para la columna que recibe como parámetro.
  - `protected ICellModifier getCellModifier(FEFEMPage page)`
    - Override: Sí.
    - Funcionalidad: Devuelve un objeto ParameterCellModifier para poder

modificar la celda 'value' de la tabla.

- `protected String[] getColumNames()`
  - Override: Sí.
  - Funcionalidad: Devuelve una lista con los nombres de las columnas.
- `protected EStructuralFeature getFeature()`
  - Override: Sí.
  - Funcionalidad: Devuelve la feature correspondiente para mostrar los elementos en la tabla.
- `protected CellLabelProvider getLabelProvider()`
  - Override: Sí.
  - Funcionalidad: Devuelve un objeto de tipo ParameterLabelProvider.

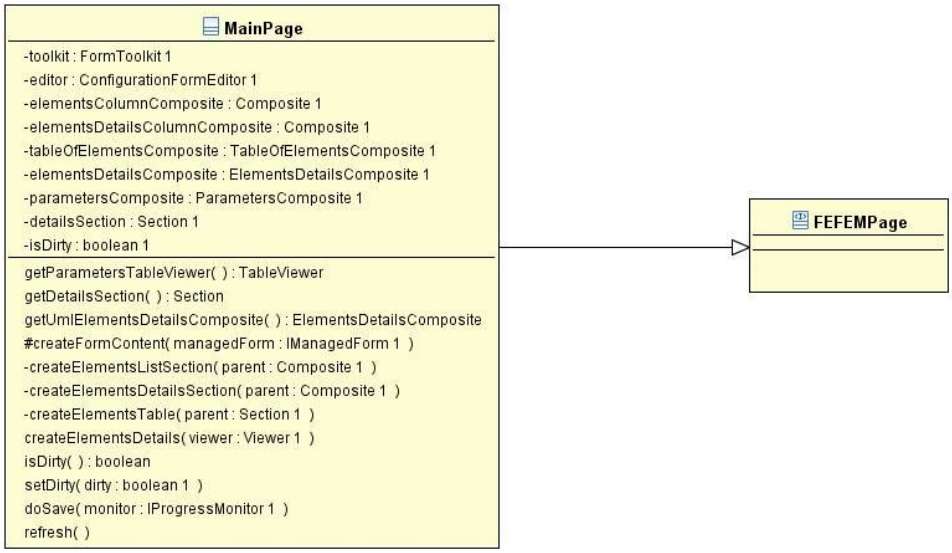


- **MainPage**
  - Tipo: Page
  - Superclase: es.cv.gvcase.fefem.common.FEFEMPage
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.pages
  - Modo de utilización: auxiliar.
  - Campos:
    - `private Composite elementsColumnComposite;`
    - `private Composite elementsDetailsColumnComposite;`

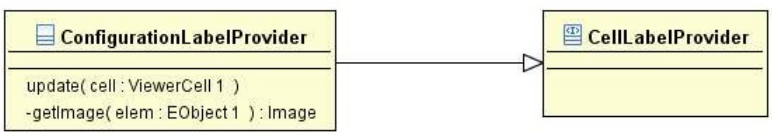
- `private TableOfElementsComposite tableOfElementsComposite;`
- `private ElementsDetailsComposite elementsDetailsComposite;`
- `private ParametersComposite parametersComposite;`
- `private Section detailsSection;`
- `private boolean isDirty;`

- Métodos:
  - `public TableViewer getParametersTableViewer()`
    - Override: No.
    - Funcionalidad: Devuelve el elemento TableViewer asociado al composite ParametersComposite.
  - `public Section getDetailsSection()`
    - Override: No.
    - Funcionalidad: Devuelve el campo detailsSection.
  - `public ElementsDetailsComposite getUmlElementsDetailsComposite()`
    - Override: No.
    - Funcionalidad: Devuelve el campo elementsDetailsComposite.
  - `protected void createFormContent(IManagedForm managedForm)`
    - Override: Sí.
    - Funcionalidad: crea el contenido de la página.
  - `private void createElementsListSection(Composite parent)`
    - Override: No.
    - Funcionalidad: crea la sección que contiene la tabla con los elementos UML configurados.
  - `private void createElementsDetailsSection(Composite parent)`
    - Override: No.
    - Funcionalidad: crea la sección que contiene los detalles del elemento UML seleccionado.

- `private void createElementsTable(Section parent)`
  - Override: No.
  - Funcionalidad: crea el composite y la table que contiene los elementos UML configurados.
- `public void createElementsDetails(Viewer viewer)`
  - Override: No.
  - Funcionalidad: crea el composite que contendrá los diferentes widgets para editar los detalles del elemento UML seleccionado.
- `public boolean isDirty()`
  - Override: Sí.
  - Funcionalidad: Devuelve si true o false dependiendo si el modelo ha sufrido cambios.
- `public void setDirty(boolean dirty)`
  - Override: Sí.
  - Funcionalidad: Modifica el estado de la página dependiendo del argumento pasado como parámetro.
- `public void doSave(IProgressMonitor monitor)`
  - Override: Sí.
  - Funcionalidad: Almacena los cambios en el modelo.
- `public void refresh()`
  - Override: Sí.
  - Funcionalidad: Refresca el TableViewer del composite ParametersComposite para que se actualicen los cambios.



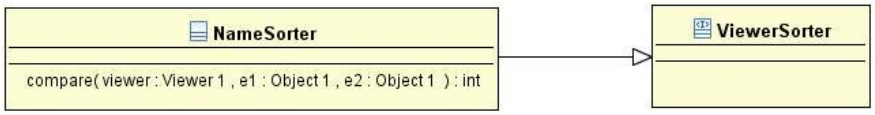
- **ConfigurationLabelProvider**
  - Tipo: LabelProvider
  - Superclase: org.eclipse.jface.viewers.CellLabelProvider
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.providers
  - Modo de utilización: auxiliar.
  - Métodos:
    - `public void update(ViewerCell cell)`
      - Override: Sí.
      - Funcionalidad: Actualiza la celda en base al valor de la celda actual.
    - `private Image getImage(EObject elem)`
      - Override: Sí.
      - Funcionalidad: Devuelve una imagen para mostrarla en la celda correspondiente.



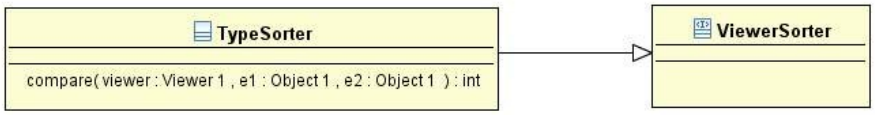
- **ParameterLabelProvider**

- Tipo: LabelProvider
- Superclase: org.eclipse.jface.viewers.CellLabelProvider
- Carpeta: src
- Paquete: es.cv.gvcase.modelsync.configuration.formseditor.providers
- Modo de utilización: auxiliar.
- Métodos:
  - `public void update(ViewerCell cell)`
    - Override: Sí.
    - Funcionalidad: Actualiza la celda en base al valor de la celda actual.

- **NameSorter**
  - Tipo: Sorter
  - Superclase: org.eclipse.jface.viewers.ViewerSorter
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.sorters
  - Modo de utilización: auxiliar.
  - Métodos:
    - `public int compare(Viewer viewer, Object e1, Object e2)`
      - Override: Sí.
      - Funcionalidad: Devuelve dados dos elementos un número positivo si el primer elemento es mayor que el segundo y negativo en caso contrario. Devuelve cero si son iguales. En este caso se comparan los nombres.



- **TypeSorter**
  - Tipo: Sorter
  - Superclase: org.eclipse.jface.viewers.ViewerSorter
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.sorters
  - Modo de utilización: auxiliar.
  - Métodos:
    - `public int compare(Viewer viewer, Object e1, Object e2)`
      - Override: Sí.
      - Funcionalidad: Devuelve dados dos elementos un número positivo si el primer elemento es mayor que el segundo y negativo en caso contrario. Devuelve cero si son iguales. En este caso se comparan los tipos de los elementos alfabéticamente.



- **ValueSorter**
  - Tipo: Sorter
  - Superclase: org.eclipse.jface.viewers.ViewerSorter
  - Carpeta: src
  - Paquete: es.cv.gvcase.modelsync.configuration.formseditor.sorters
  - Modo de utilización: auxiliar.
  - Métodos:
    - `public int compare(Viewer viewer, Object e1, Object e2)`
      - Override: Sí.
      - Funcionalidad: Devuelve dados dos elementos un número positivo si el primer elemento es mayor que el segundo y negativo en caso contrario. Devuelve cero si son iguales. En este caso se comparan los atributos 'value'

de los elementos alfabéticamente.