FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Transformaciones de modelos: yendo de UML a un DSL y viceversa. El caso de FlexAB.

AUTORES: Diego Mancino, Francisco Javier Andrade

DIRECTOR: Claudia Pons

CODIRECTOR: Alejandro Fernández

CARRERA: Licenciatura en Sistemas

Resumen

Generalmente UML es el lenguaje elegido para definir los modelos en MDD. UML es un estándar abierto y es el estándar de facto para el modelado de software. UML es un lenguaje de modelado de software de propósito general que podemos aplicar de varias formas. Las técnicas de modelado apropiadas para diseñar una aplicación de telefonía, embebida y de tiempo real y las técnicas de modelado para desarrollar una aplicación de comercio electrónico son bastante diferentes, sin embargo podemos usar UML en ambos casos.
La herramienta Flexible Application Builder® de AppliWare® permite generar aplicaciones para el manejo de información con orientación a objetos, de una manera sencilla, sin programación, y sólo configurando las reglas entre los objetos que intervienen dentro del proceso. FlexAB utiliza un lenguaje gráfico específico del dominio que permite definir los conceptos de la aplicación de manera abstracta e independiente de la implementación.
Dadas las ventajas y desventajas de UML con respecto a los DSLs en esta tesis se realizó una herramienta para la traducción automática entre modelos UML y modelos en un lenguaje DSL.
La herramienta desarrollada realiza la traducción en entre el modelo FlexAB y el modelo UML.

Líneas de Investigación

Técnicas de modelado.
Lenguajes de transformaciones ATL.
Eclipse.
Definición y modelado de FlexAB.
Transformaciones de modelos.

Conclusiones

Se ha desarrollado un software complejo que logra capturar la transformación bi-direccional (en parte) de ambos lenguajes (FlexAB y UML). El resultado es una herramienta interesante que permite el enfoque del modelador de acuerdo a sus preferencias y/o habilidades. También, las transformaciones permiten la migración de modelos escritos en FlexAB para poder presentárselos a modeladores que no conocen tanto este lenguaje y si poseen conocimientos de UML. En ocasiones, es beneficioso tener el modelo en UML porque muchas de las herramientas utilizadas para el modelado soportan el lenguaje UML.

Trabajos Realizados

En esta tesis se ha analizado el modelo de FlexAB y su correspondencia con el modelo UML. Se ha definido la arquitectura para la herramienta defnida y desarrollada en esta tesis. Se definió la transformación de FlexAB a UML y el caso contrario de UML a FlexAB.

Trabajos Futuros

Completar la transformaciones con las clases de los modelos que quedaron fuera del alcance de esta herramienta. Adaptar a las transformaciones una herramienta para la instanciación de los modelos.

Fecha de la presentación: Febrero de 2010

Agradecimientos

Quiero agradecer a toda mi familia por el esfuerzo que han realizado para que yo pueda llevar adelante mis estudios Universitarios. El apoyo de mis hermanos, mi abuela y principalmente mi mamá Alicia que ha sido la responsable de que pueda estudiar. También quiero agradecer a mi novia Mónica que ha estado conmigo estos últimos años dándome su apoyo incondicional.
Por último quiero agradecer a la Facultad de Informática por darnos esta posibilidad.

Francisco.

Agradecer enormemente a mi familia, papa Luis, mama Marta y mis hermanos Hernán y Laura, incondicionales siempre, y sin los cuales nunca hubiese podido llegar a este momento. A mi abuela Ñata y mi tía Ana por estar en todas. A mis amigos de siempre por aportar su apoyo cuando fue necesario.
A mi compañero de tesis Francisco por su esfuerzo desinteresado cuando me quedaban pocas energías.

Diego.

En nombres de los dos queremos agradecer a nuestra Directora de Tesis Claudia Pons y a Gabriela Pérez que nos han dedicado su tiempo para que podamos realizar este trabajo.

Índice

Lista de Figuras

1 Introducción

1.1 Motivación

El desarrollo de software dirigido por modelos (MDD)

El Desarrollo de Software Dirigido por Modelos MDD (por sus siglas en inglés: Model Driven software Development) se ha convertido en un nuevo paradigma de desarrollo software. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. El adjetivo "dirigido" (*driven)* en MDD, a diferencia de "basado" (*based)*, enfatiza que este paradigma asigna a los modelos un rol central y activo: son al menos tan importantes como el código fuente. Los modelos se van generando desde los más abstractos a los más concretos a través pasos de transformación y/o refinamientos, hasta llegar al código aplicando una última transformación. La transformación entre modelos constituye el motor del MDD.

Propuestas Concretas para MDD

Dos de las propuestas concretas más conocidas y utilizadas en el ámbito de MDD son, por un lado MDA desarrollada por el OMG y por otro lado el Modelado Específico del Domino (DSM) acompañado por los lenguajes específicos del dominio (DSLs). Ambas iniciativas guardan naturalmente una fuerte conexión con los conceptos básicos de MDD. Específicamente, MDA tiende a enfocarse en lenguajes de modelado basados en estándares del OMG, mientras que DSM utiliza otras notaciones no estandarizadas para definir sus modelos.

Arquitectura Dirigida por Modelos (MDA)

MDA, acrónimo de *Model Driven Architecture* (Arquitectura Dirigida por Modelos), es una de las iniciativas más conocida y extendida dentro del ámbito de MDD. MDA es un concepto promovido por el OMG *Object Management Group* en noviembre de 2000, con el objetivo de afrontar los desafíos de integración de las aplicaciones y los continuos cambios tecnológicos. MDA es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos, siguiendo el proceso MDD.
En MDA, la funcionalidad del sistema es definida en primer lugar como un modelo independiente de la plataforma (Platform-Independent Model o PIM) a través de un

lenguaje específico para el dominio del que se trate. En este punto aparece además un tipo de modelo existente en MDA, no mencionado por MDD: el modelo de definición de la plataforma (Platform Definition Model o PDM). Este especifica el metamodelo de la plataforma destino. Entonces, Dado un PDM correspondiente a CORBA, .NET, Web, etc., el modelo PIM puede traducirse a uno o más modelos específicos de la plataforma (Platform-Specific Models o PSMs) para la implementación correspondiente, usando diferentes lenguajes específicos del dominio, o lenguajes de propósito general como JAVA, C#, Python, etc. El proceso MDA completo se encuentra detallado en un documento que actualiza y mantiene el OMG denominado la Guía MDA.

MDA está relacionada con múltiples estándares, tales como el Unified Modeling Language (UML), el Meta-Object Facility (MOF), XML Metadata Interchange (XMI), Enterprise Distributed Object Computing (EDOC), el Software Process Engineering Metamodel (SPEM) y el Common Warehouse Metamodel (CWM).

Cumpliendo con las directivas del OMG, las dos principales motivaciones de MDA son la interoperabilidad (independencia de los fabricantes a través de estandarizaciones) y la portabilidad (independencia de la plataforma) de los sistemas de software; las mismas motivaciones que llevaron al desarrollo de CORBA. Además, OMG postula como objetivo de MDA separar el diseño del sistema tanto de la arquitectura como de las tecnologías de construcción, facilitando así que el diseño y la arquitectura puedan ser alterados independientemente. El diseño alberga los requisitos funcionales (casos de uso, por ejemplo) mientras que la arquitectura proporciona la infraestructura a través de la cual se hacen efectivos los requisitos no funcionales como la escalabilidad, fiabilidad o rendimiento. MDA se asegura que el modelo independiente de la plataforma (PIM), el cual representa un diseño conceptual que plasma los requisitos funcionales, sobreviva a los cambios que se produzcan en las tecnologías de fabricación y en las arquitecturas de software. Por supuesto, la noción de transformación de modelos en MDA es central. La traducción entre modelos se realiza normalmente utilizando herramientas automatizadas, es decir herramientas de transformación de modelos que soportan MDA. Algunas de ellas permiten al usuario definir sus propias transformaciones. Una iniciativa del OMG es la definición de un lenguaje de transformaciones estándar denominado QVT que aún no ha sido masivamente adoptado por las herramientas, por lo que la mayoría de ellas aun definen su propio lenguaje de transformación, y sólo algunos de éstos se basan en QVT. Actualmente existe un amplio conjunto de herramientas que brindan soporte para MDA.

Modelado Específico del Dominio (DSM y DSLs)

Por su parte, la iniciativa DSM (Domain-Specific Modeling) es principalmente conocida como la idea de crear modelos para un dominio, utilizando un lenguaje focalizado y especializado para dicho dominio. Estos lenguajes se denominan DSLs (por su nombre en inglés: Domain-Specific Language) y permiten especificar la solución usando directamente conceptos del dominio del problema. Los productos finales son luego generados desde estas especificaciones de alto nivel. Esta automatización es posible si ambos, el lenguaje y los generadores, se ajustan a los requisitos de un único dominio. Definimos como dominio a un área de interés para un esfuerzo de desarrollo en particular. En la práctica, cada solución DSM se enfoca en dominios pequeños porque el foco reductor habilita mejores posibilidades para su automatización y estos dominios pequeños son más fáciles de definir. Usualmente, las soluciones en DSM son usadas en relación a un producto particular, una línea de producción, un ambiente específico, o una plataforma.

El desafío de los desarrolladores y empresas se centra en la definición de los lenguajes de modelado, la creación de los generadores de código y la implementación de *framework* específicos del dominio, elementos claves de una solución DSM. Estos elementos no se encuentran demasiado distantes de los elementos de modelado de MDD. Actualmente puede observarse que las discrepancias entre DSM y MDD han comenzado a disminuir. Podemos comparar el uso de modelos así como la construcción de la infraestructura respectiva en DSM y en MDD. En general, DSM usa los conceptos dominio, modelo, metamodelo, meta-metamodelo como MDD, sin mayores cambios y propone la automatización en el ciclo de vida del software. Los DSLs son usados para construir modelos. Estos lenguajes son frecuentemente -pero no necesariamente- gráficos. Los DSLs no utilizan ningún estándar del OMG para su infraestructura, es decir no están basados en UML; los metamodelos no son instancias de MOF; a diferencia usan por ejemplo MDF, el framework de Metadatos para este propósito.

Desde la perspectiva del desarrollador de la aplicación, los modelos son artefactos de primera clase en el desarrollo de proyectos, y editores y transformaciones están integrados dentro de la IDE. Desde la perspectiva del desarrollador de la infraestructura, los metamodelos, definiciones de editores y transformaciones son artefactos de primera clase, y las herramientas para construirlos están también integradas dentro de la IDE.

Finalmente, existe una familia importante de herramientas para crear soluciones en DSM que ayudan en la labor de automatización. En tal sentido, surge el término "*Software Factories*", que ha sido acuñado por Microsoft. Su descripción en forma detallada puede encontrarse en el libro de Greenfields del mismo nombre. Una *Software Factory* es una línea de producción de software que configura herramientas de

desarrollo extensibles tales como Visual Studio Team System con contenido empaquetado como DSLs, patrones y *frameworks*, basados en recetas para construir tipos específicos de aplicaciones. Visual Studio provee herramientas para definir los metamodelos así como su sintaxis concreta y editores.

El rol de UML en MDD

Generalmente UML es el lenguaje elegido para definir los modelos en MDD. UML es un estándar abierto y es el estándar de facto para el modelado de software. UML es un lenguaje de modelado de software de propósito general que podemos aplicar de varias formas. Las técnicas de modelado apropiadas para diseñar una aplicación de telefonía, embebida y de tiempo real y las técnicas de modelado para desarrollar una aplicación de comercio electrónico son bastante diferentes, sin embargo podemos usar UML en ambos casos.

El uso de UML como lenguaje en MDD tiene los siguientes beneficios:

- El uso de perfiles de UML para MDD nos permite aprovechar la experiencia que se ha ganado en el desarrollo de este lenguaje. Esto significa que podemos proveer un ambiente de modelado personalizado sin afrontar el costo de diseñarlo e implementarlo desde cero;
- UML es un estándar abierto y el estándar de facto para el modelado de software en la industria;
- UML ha demostrado ser durable, su primera versión apareció en 1995;
- El éxito de UML ha propiciado la disponibilidad de muchos libros y cursos de muy buena calidad;
- La mayoría de los estudiantes de las carreras relacionadas con la ingeniería de software han aprendido algo de UML en la universidad;
- Existen muchas herramientas maduras que soportan UML;

Muchas organizaciones necesitan desarrollar varios tipos diferentes de software. Si podemos usar una solución de modelado en común, configurada de forma apropiada para describir cada uno de esos dominios de software, entonces será más fácil lidiar con su posterior integración.

Sin embargo, no todas son ventajas al usar UML como lenguaje de modelado. También podemos señalar las siguientes desventajas:

Algunas de las herramientas de modelado que soportan UML no permiten la definición de perfiles.

Las herramientas para UML que sí soportan perfiles sólo lo hacen superficialmente, no permitiendo aprovechar todo el potencial de este mecanismo de extensión.

UML por ser un lenguaje de propósito general es muy amplio. Incluye decenas de

elementos de modelado que además pueden combinarse de formas diversas. En cambio, los lenguajes específicos de un dominio suelen ser pequeños (es decir, contienen pocos conceptos). El modelador sólo necesita entender esos pocos conceptos, en cambio si usamos un lenguaje basado en UML necesitaremos como pre-requisito un conocimiento bastante completo sobre UML.

Un caso de estudio: el DSL de FlexAB

La herramienta Flexible Application Builder® de AppliWare® permite generar aplicaciones para le manejo de información con orientación a objetos, de una manera sencilla, sin programación, y sólo configurando las reglas entre los objetos que intervienen dentro del proceso. FlexAB utiliza un lenguaje gráfico específico del dominio que permite definir los conceptos de la aplicación de manera abstracta e independiente de la implementación.

Planteo del problema

Dadas las ventajas y desventajas de UML con respecto a los DSLs seria deseable contar con la posibilidad de realizar una traducción automática entre modelos UML y modelos en un lenguaje DSL. Esta traducción no puede definirse en forma general sino que debe crearse para cada DSL en particular.

1.2 Objetivos

El objetivo de esta tesis es analizar la correspondencia entre UML y un DSL - en particular el lenguaje de modelado de FlexAB - e implementar una herramienta que automatice las transformaciones entre los correspondientes elementos de modelado de ambos lenguajes.

1.3 Organización de la Tesis

A continuación se hace una breve descripción de cómo está organizada la tesis. En el primer capítulo se hace una introducción al tema que se va tratar explicando los objetivos de esta tesis. En el capítulo 2 se explicará que es un metamodelo, distintas técnicas para definirlos y algunos ejemplos de metamodelos. En el capítulo 3 se hace una introducción de la herramienta FlexAB explicando su funcionamiento y sus objetivos de utilización, a continuación se define el metamodelo de FlexAB detallando los paquetes con sus respectivas clases. En el capítulo 4 se describe los conceptos de

transformaciones.

Los capítulos 5, 6 y 7 están muy relacionados. En el capítulo 5 se detallan las tecnologías utilizadas, en el capítulo 6 se detalla el diseño de la solución planteada y en el capítulo 7 se describe la implementación de la solución.

En el capítulo 8 se hace un caso de estudio. En el capítulo 9 se describen las conclusiones y en el capítulo 10 se proponen los trabajos futuros. En el último capítulo de este trabajo de grado, el 11, se encuentra la bibliografía consultada.

2 Definición Formal de lenguajes de modelado

En este parte explicaremos los mecanismos para definir la sintaxis de los lenguajes con los cuales se crean los modelos de un sistema. En particular, discutiremos la técnica de metamodelado, la arquitectura de cuatro capas de modelado definida por el OMG y por qué el metamodelado es tan importante en el contexto del desarrollo de software dirigido por modelos (MDD).

2.1 Mecanismos para definir la sintaxis de un lenguaje de modelado

Hace algunos años, la sintaxis de los lenguajes se definía casi exclusivamente usando Backus Naur Form (BNF). Este formalismo es una meta sintaxis usada para expresar gramáticas libres de contexto, es decir, una manera formal de describir cuales son las palabras básicas del lenguaje y cuales secuencias de palabras forman una expresión correcta dentro del lenguaje. Una especificación en BNF es un sistema de reglas de la derivación, escrito como:
<símbolo> ::= <expresión con símbolos> donde <símbolo> es un no-terminal, y la expresión consiste en secuencias de símbolos separadas por la barra vertical, '|', indicando una opción, cada una de las cuales es una posible substitución para el símbolo a la izquierda. Los símbolos que nunca aparecen en un lado izquierdo son terminales.
El BNF se utiliza extensamente como notación para definir la sintaxis (o gramática) de los lenguajes de programación. Por ejemplo, las siguientes expresiones BNF definen la sintaxis de un lenguaje de programación simple:
<Programa> ::= "**Begin**" <Comando> "**end**"
<Comando> ::= <Asignación>|<Loop>|<Decisión>|<Comando>";"<Comando>
<Asignación> ::= variableName "**:=**" <Expresión>
<Loop> ::= "**while**" <Expresión> "**do**" <Comando> "**end**"
<Decisión> ::= "**if**"<Expresión>"**then**"<Comando>"**else**"<Comando>"**endif**"
<Expresión> ::= ...
El siguiente código es una instancia válida de esta definición:

```
Begin
   if y=0 then
         result:=0
   else
         result:=x;
         while (y>1) do result:=result+x; y:=y-1 end
   endif
end
```

Este método es útil para lenguajes textuales, pero dado que los lenguajes de modelado en general no están basados en texto, sino en gráficos, es conveniente recurrir a un mecanismo diferente para definirlos. Las principales diferencias entre los lenguajes basados en texto y los lenguajes basados en gráficos son las siguientes:

- **contenedor vs. referencia**: En los lenguajes textuales una expresión puede formar parte de otra expresión, que a su vez puede estar contenida en otra expresión mayor. Esta relación de contenedor da origen a un árbol de expresiones.

  En el caso de los lenguajes gráficos, en lugar de un árbol se origina un grafo de expresiones ya que una sub-expresión puede ser referenciada desde dos o más expresiones diferentes.
- **sintaxis concreta vs. sintaxis abstracta**. En los lenguajes textuales la sintaxis concreta coincide (casi) exactamente con la sintaxis abstracta mientras que en los lenguajes gráficos se presenta una marcada diferencia entre ambas.
- **ausencia de una jerarquía clara en la estructura del lenguaje**. Los lenguajes textuales en general se aprehenden leyéndolos de arriba hacia abajo y de izquierda a derecha, en cambio los lenguajes gráficos suelen asimilarse de manera diferente dependiendo de su sintaxis concreta (por ejemplo, comenzamos prestando atención al diagrama más grande y/o colorido). Esto influye en la jerarquía de la sintaxis abstracta, ocasionando que no siempre exista un orden entre las categorías sintácticas.

Por lo tanto, en los últimos años se desarrolló una técnica específica para facilitar la definición de los lenguajes gráficos, llamada “metamodelado”. Veamos en que consiste esta nueva técnica: usando un lenguaje de modelado, podemos crear modelos; un modelo especifica que elementos pueden existir en un sistema. Si se define la clase Persona en un modelo, se pueden tener instancias de Persona como Juan, Pedro, etc.

Por otro lado, la definición de un lenguaje de modelado establece que elementos pueden existir en un modelo. Por ejemplo, el lenguaje UML establece que dentro de un modelo se pueden usar los conceptos Clase, Atributo, Asociación, Paquete, etc. Debido a esta similitud, se puede describir un lenguaje por medio de un modelo, usualmente llamado “metamodelo”. El metamodelo de un lenguaje describe que elementos pueden ser usados en el lenguaje y como pueden ser conectados.

Como un metamodelo es también un modelo, el metamodelo en sí mismo debe estar escrito en un lenguaje bien definido. Este lenguaje se llama metalenguaje. Desde este punto de vista, BNF es un metalenguaje. En la siguiente figura se muestra gráficamente esta relación.

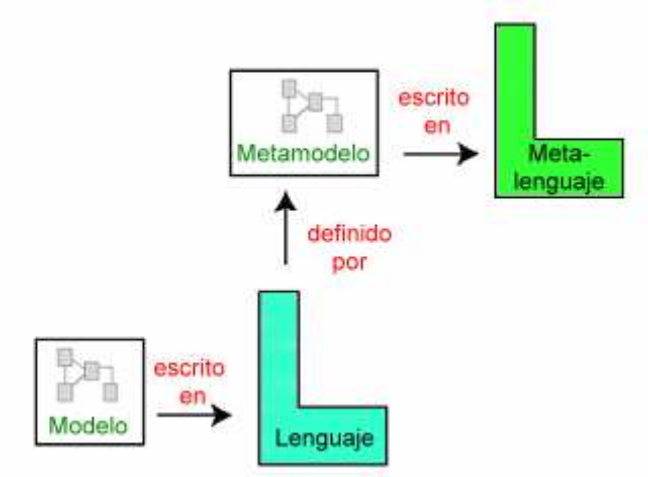

Figura 1 - Metalenguaje

El metamodelo describe la sintaxis abstracta del lenguaje. Esta sintaxis es la base para el procesamiento automatizado (basado en herramientas) de los modelos. Por otra parte, la sintaxis concreta es definida mediante otros mecanismos y no es relevante para las herramientas de transformación de modelos. La sintaxis concreta es la interfaz para el modelador e influye fuertemente en el grado de legibilidad de los modelos.
Como consecuencia de este desacoplamiento, el metamodelo y la sintaxis concreta de un lenguaje pueden mantener una relación 1:n, es decir que la misma sintaxis abstracta (definida por el metamodelo) puede ser visualizada a través de diferentes sintaxis concretas. Inclusive un mismo lenguaje puede tener una sintaxis concreta gráfica y otra textual.

2.1.1 La arquitectura de 4 capas de modelado de OMG

El metamodelado es entonces un mecanismo que permite definir formalmente lenguajes de modelado, como por ejemplo UML. La Arquitectura de cuatro capas de modelado es la propuesta del OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.
Los cuatro niveles definidos en esta arquitectura se denominan: M3, M2, M1, M0 y los describimos a continuación. Para entender mejor la relación entre los elementos en las distintas capas, presentamos un ejemplo utilizando el lenguaje UML. En este ejemplo modelamos un sistema de venta de libros por Internet, que maneja información acerca de los clientes y de los libros de los cuales se dispone.

Nivel M0: Instancias

En el nivel M0 se encuentran todas las instancias "reales" del sistema, es decir, los objetos de la aplicación. Aquí no se habla de clases, ni atributos, sino de entidades físicas que existen en el sistema.
En la figura siguiente se muestra un diagrama de objetos UML donde pueden verse las distintas entidades que almacenan los datos necesarios para nuestro sistema. Se tiene una librería virtual, llamada 'El Ateneo'. Esta librería tiene un cliente, Juan García, del cual queremos guardar su nombre de usuario, su palabra clave, su nombre real, su dirección postal y su dirección de e-mail. Además, la librería tiene una categoría de libros con nombre 'Novela'. Por último, la librería tiene un libro con título 'Cien Años de Soledad' de la cual se conoce su autor.
Todas estas entidades son instancias pertenecientes a la capa M0.

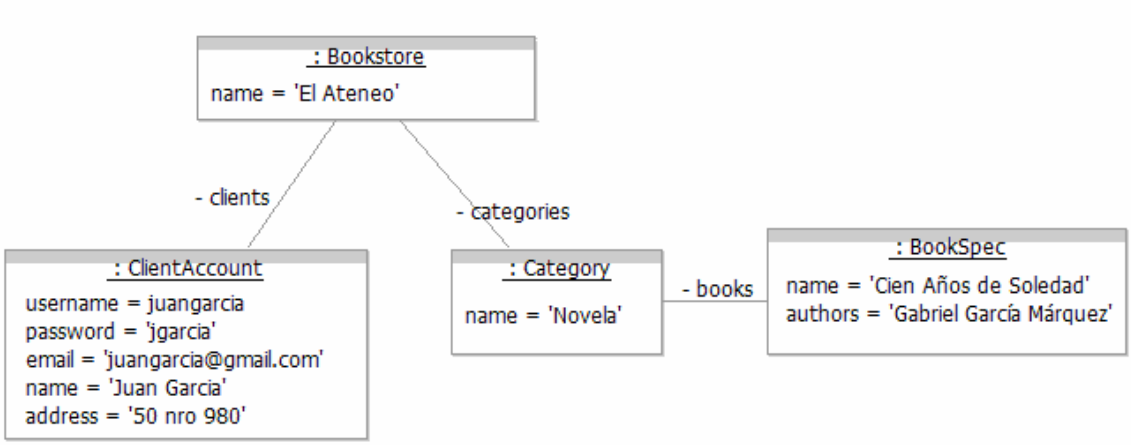

Figura 2 - Nivel M0

Nivel M1: Modelo del sistema

Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es una instancia de un elemento de M1. Sus elementos son modelos de los datos. En el nivel M1 aparece entonces la entidad Librería, la cual representa las librerías del sistema, tales como 'El Ateneo', con los atributos nombre y dirección. Lo mismo ocurre con las entidades Cliente y Libro. En la siguiente figura se muestra un modelo de clases UML para este ejemplo.

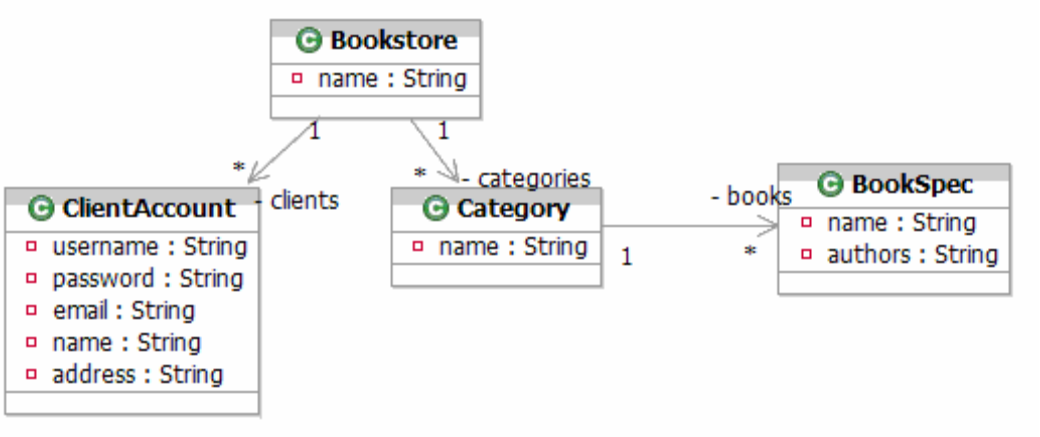

Figura 3 - Nivel M1

El objeto con nombre 'El Ateneo' puede verse ahora como una instancia de Librería. De la misma manera, el cliente de nombre Juan García puede verse como una instancia de la entidad Cliente y 'Cien Años de Soledad' como una instancia de Libro. En la figura siguiente se muestra la relación entre el nivel M0 y el nivel M1.

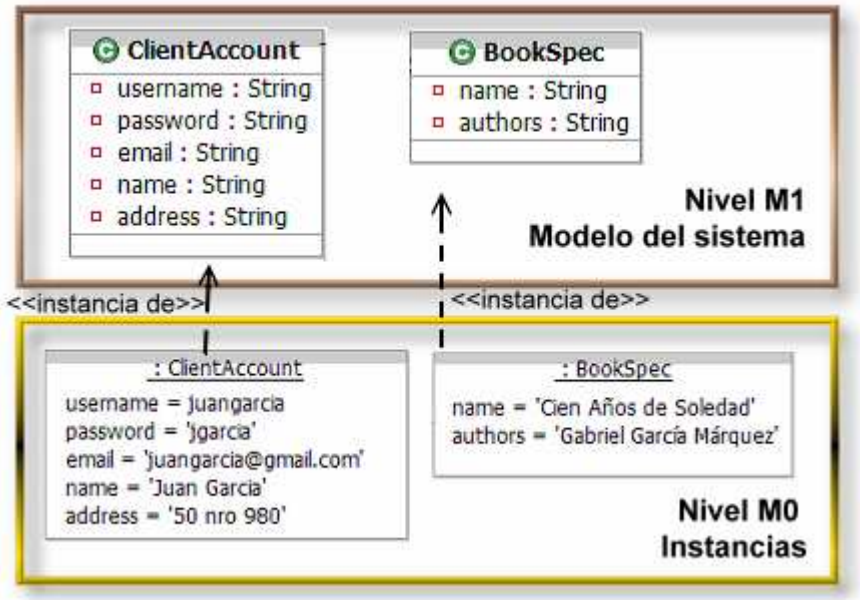

Figura 4 - Relación entre nivel M0 y M1

Nivel M2: Metamodelo

Análogamente a lo que ocurre con las capas M0 y M1, los elementos del nivel M1 son a su vez instancias del nivel M2. Esta capa recibe el nombre de metamodelo. La figura siguiente muestra una parte del metamodelo UML. En este nivel aparecen conceptos tales como Clase (Class), Atributo (Attribute) y Operación (Operation).

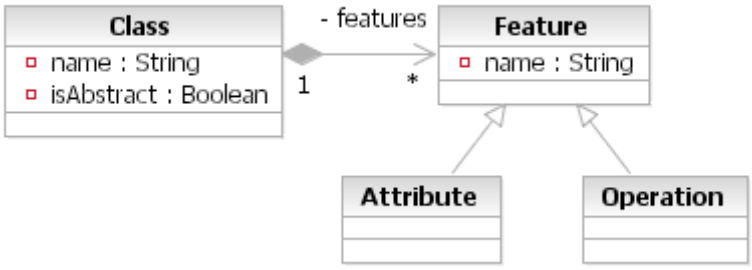

Figura 5 - Nivel M2

Siguiendo el ejemplo, la entidad ClientAccount será una instancia de la metaclase Class del metamodelo UML. Esta instancia tiene cinco objetos relacionados a través de la meta asociación feature, por ejemplo una instancia de Attribute con name=’username’ y type=’String’ y otra instancia de Attribute con name=’password’ y type=’String’.
La figura siguiente muestra la relación entre los elementos del nivel M1 con los elementos del nivel M2.

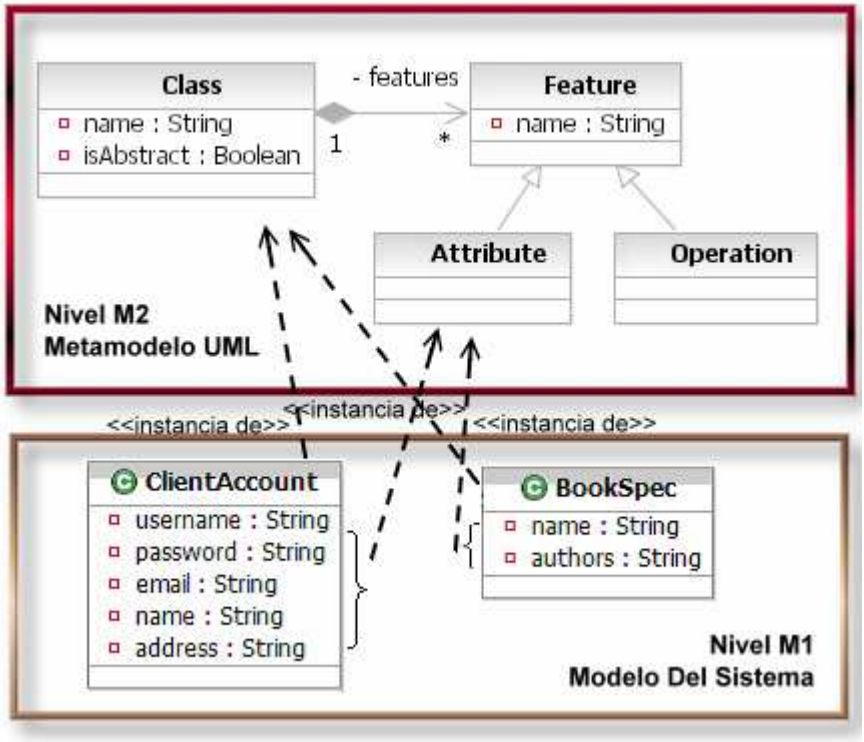

Figura 6 - Relación M1 y M2

Nivel M3: Meta-metamodelo

De la misma manera podemos ver los elementos de M2 como instancias de otra capa, la capa M3 o capa de meta-metamodelo. Un meta-metamodelo es un modelo que define el lenguaje para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y un modelo.
M3 es el nivel más abstracto, que permite definir metamodelos concretos. Dentro del OMG, MOF [6] es el lenguaje estándar de la capa M3. Esto supone que todos los metamodelos de la capa M2, son instancias de MOF.

La figura siguiente muestra la relación entre los elementos del metamodelo UML (Nivel M2) con los elementos de MOF (Nivel M3). Puede verse que las entidades de la capa M2 son instancias de las metaclases MOF de la capa M3.

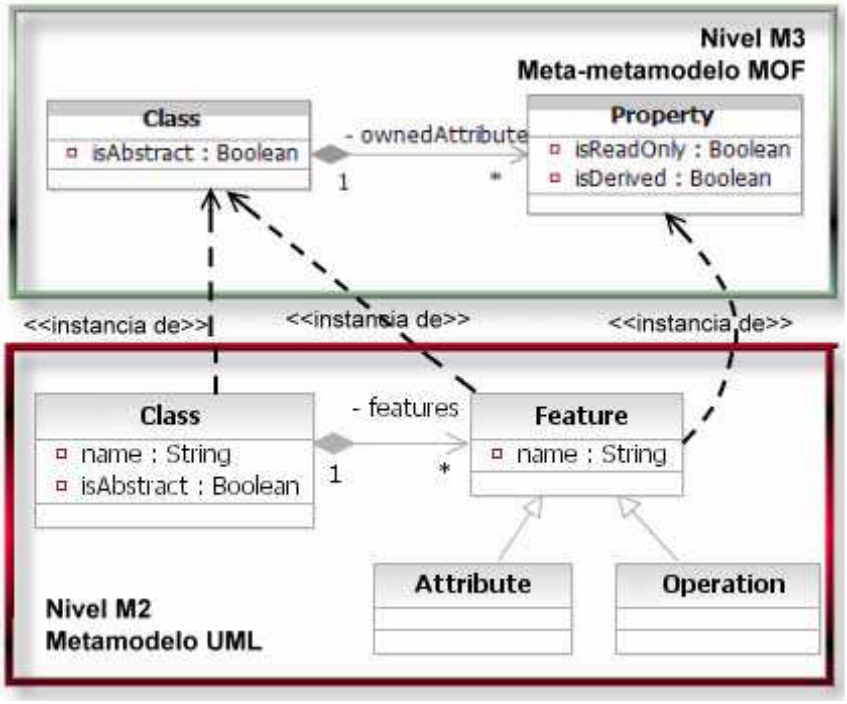

Figura 7 - Nivel M3

No existe otro meta nivel por encima de MOF. Básicamente, el MOF se define a sí mismo.

Por último, como vista general, la figura siguiente muestra las cuatro capas de la arquitectura de modelado, indicando las relaciones entre los elementos en las diferentes capas.

Puede verse que en la capa M3 se encuentra el meta-metamodelo MOF, a partir del cual se pueden definir distintos metamodelos en el nivel M2, como UML, JAVA, OCL, etc.

Instancias de estos metamodelos serán los elementos del nivel M1, como modelos UML, o modelos JAVA. A su vez, instancias de los elementos M1 serán los objetos que cobran vida en las corridas del sistema.

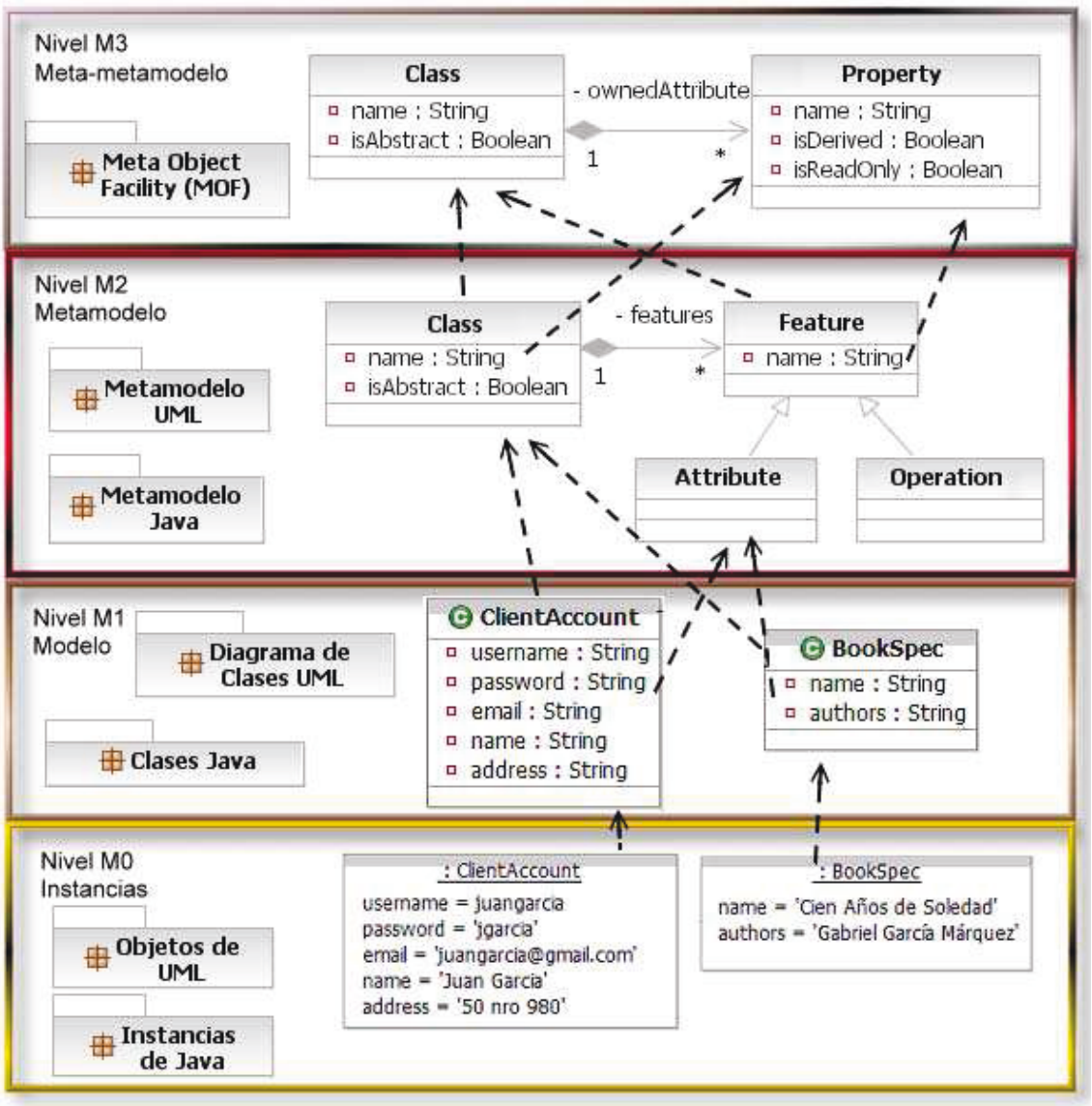

Figura 8 - Niveles M0, M1, M2 y M3

2.1.2 El uso del metamodelado en MDD

Las razones por las que el metamodelado es tan importante en MDD son:

- En primer lugar, necesitamos contar con un mecanismo para definir lenguajes de modelado sin ambigüedades y permitir que una herramienta de transformación pueda leer, escribir y entender los modelos;
- Luego, las reglas de transformación que constituyen una definición de una transformación describen como un modelo en un lenguaje fuente puede ser transformado a un modelo en un lenguaje destino. Estas reglas usan los metamodelos de los lenguajes fuente y destino para definir la transformación;
- Y finalmente, la sintaxis de los lenguajes en los cuales se expresan las reglas de transformación también debe estar formalmente definida para permitir su automatización. En este caso también se utilizará la técnica de metamodelado para especificar su sintaxis.

La figura siguiente muestra como se completa MDD con la capa de metamodelado. La parte baja de la figura es con lo que la mayoría de los desarrolladores trabaja habitualmente.
En el centro se introduce el metalenguaje para definir nuevos lenguajes. Un pequeño grupo de desarrolladores, usualmente con más experiencia, necesitarán definir lenguajes y las transformaciones entre estos lenguajes. Para este grupo, entender el metanivel será esencial a la hora de definir transformaciones.

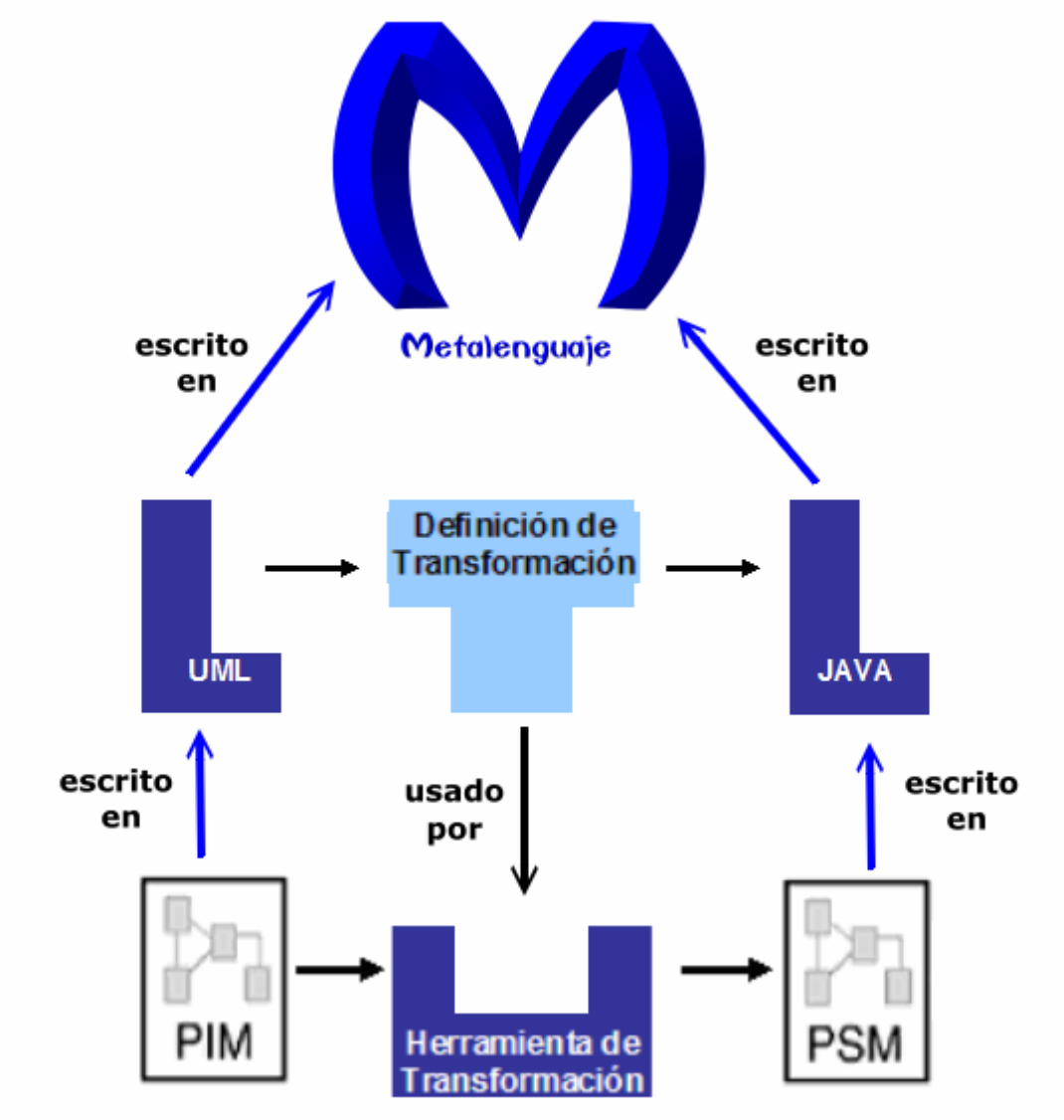

Figura 9 - Metalenguaje

2.2 El lenguaje de modelado más abstracto: MOF

El lenguaje MOF [6], acrónimo de Meta-Object Facility, es un estándar del OMG para la ingeniería conducida por modelos. Como se vio anteriormente, MOF se encuentra en la capa superior de la arquitectura de 4 capas. Provee un meta-meta lenguaje que permite definir metamodelos en la capa M2. El ejemplo más popular de un elemento en la capa M2 es el metamodelo UML, que describe al lenguaje UML.
Esta es una arquitectura de metamodelado cerrada y estricta. Es cerrada porque el metamodelo de MOF se define en términos de sí mismo. Y es estricta porque cada

elemento de un modelo en cualquiera de las capas tiene una correspondencia estricta con un elemento del modelo de la capa superior.

Tal como su nombre lo indica, MOF se basa en el paradigma de Orientación a Objetos. Por este motivo usa los mismos conceptos y la misma sintaxis concreta que los diagramas de clases de UML.
Actualmente, la definición de MOF está separada en dos partes fundamentales, EMOF (Essential MOF) y CMOF (Complete MOF), y se espera que en el futuro se agregue SMOF (Semantic MOF). La figura siguiente muestra la relación entre EMOF y CMOF.

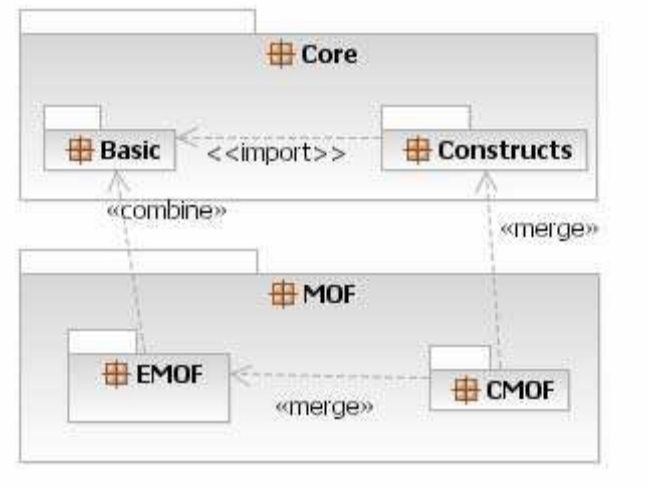

Figura 10 - Relación EMOF y CMOF

Ambos paquetes importan los elementos de un paquete en común, del cual utilizan los constructores básicos y lo extienden con los elementos necesarios para definir metamodelos simples, en el caso de EMOF y metamodelos más sofisticados, en el caso de CMOF. La figura siguiente presenta los principales elementos contenidos en el paquete EMOF.

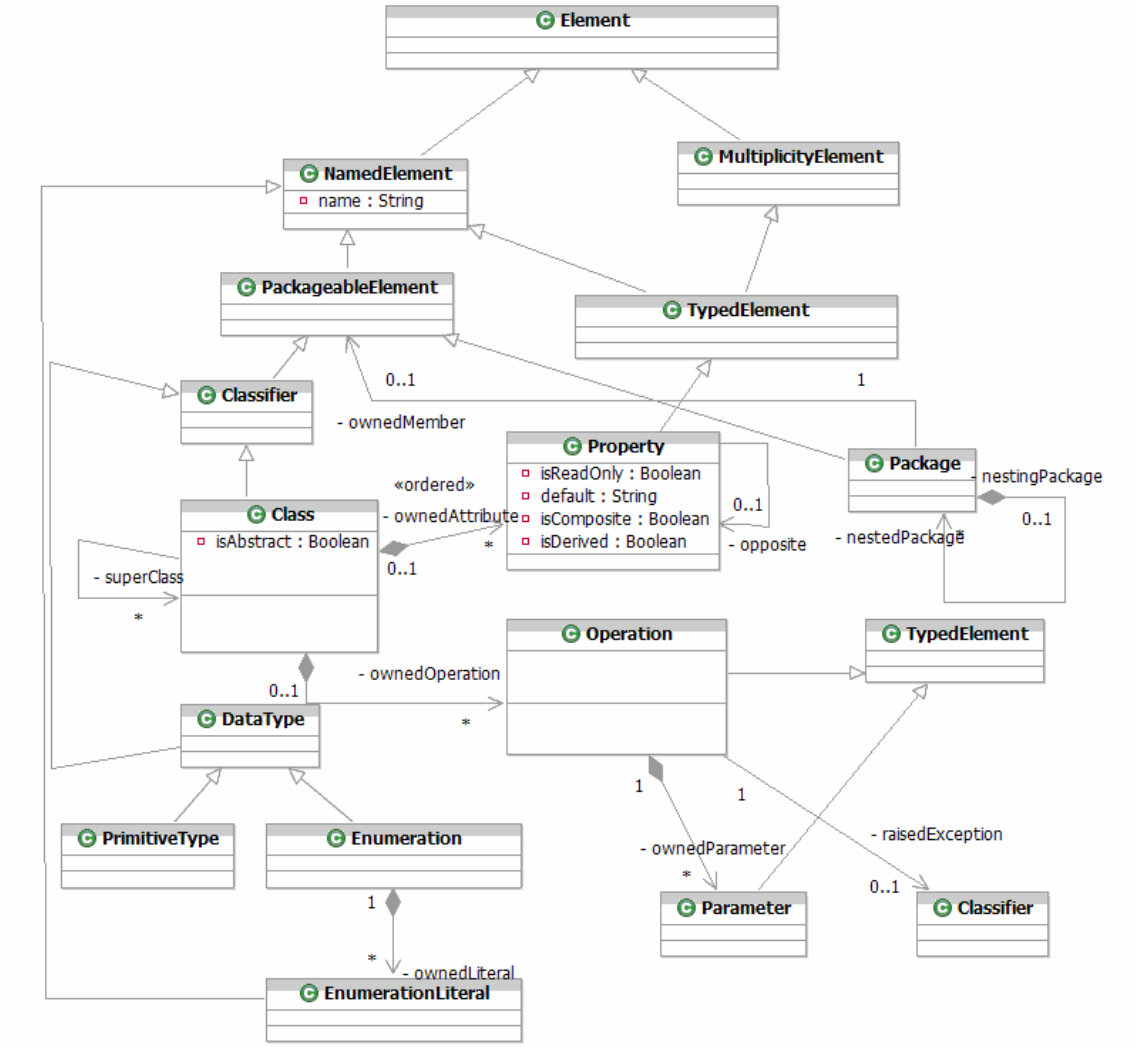

Figura 11 - Elementos EMOF

2.2.1 MOF vs. BNF

Si bien la meta sintaxis BNF y el meta lenguaje MOF son formalismos creados con el objetivo de definir lenguajes textuales y lenguajes gráficos respectivamente, se ha demostrado que ambos formalismos poseen igual poder expresivo, siendo posible la transformación bi-direccional de sentencias cuya sintaxis fue expresada en BNF y sus correspondientes modelos cuya sintaxis fue expresada en MOF. Por ejemplo, ya hemos visto como expresar la sintaxis abstracta de UML usando MOF; el siguiente código escrito en BNF define al mismo extracto de UML mostrado anteriormente:

<Class> ::= “name” String “feature” <Feature>*
<Feature> ::= <Attribute> | <Operation>
<Attribute> ::= “name” String “type” String
<Operation> ::= “name” String “type” String

2.2.2 MOF vs. UML

Actualmente la sintaxis de UML está definida usando MOF, sin embargo UML fue creado antes que MOF. Inicialmente UML no estaba formalmente definido, su sintaxis sólo estaba descrita informalmente a través de ejemplos. MOF surgió posteriormente, con el objetivo de proveer un marco formal para la definición de UML y otros lenguajes gráficos.
La sintaxis concreta de MOF coincide con la de UML, lo cual resulta algo confuso para los principiantes en el uso de estos lenguajes. Además MOF contiene algunos elementos también presentes en UML, por ejemplo, ambos lenguajes tienen un elemento llamado Class. A pesar de que los elementos tienen el mismo nombre y superficialmente describen al mismo concepto, no son idénticos y no deben ser confundidos.

2.2.3 Implementación de MOF – Ecore

El metamodelo MOF está implementado mediante un plugin para Eclipse [7] llamado Ecore [8]. Este plugin respeta las metaclases definidas por MOF. Todas las metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra "E", indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase Eclass implementa a la metaclase Class de MOF.
La primera implementación de Ecore fue terminada en Junio del 2002. Esta primera versión usaba como meta-metamodelo la definición estándar de MOF (v1.4). Gracias a las sucesivas implementaciones de Ecore y basado en la experiencia ganada con el uso de esta implementación en varias herramientas, el metalenguaje evolucionó. Como conclusión, se logró una implementación de Ecore realizada en JAVA, más eficiente y con sólo un subconjunto de MOF, y no con todos los elementos como manipulaban las implementaciones hechas hasta ese momento. A partir de este conocimiento, el grupo de desarrollo de Ecore colaboró más activamente con la nueva definición de MOF, y se estableció un subconjunto de elementos como esenciales, llegando a la definición de EMOF, que fue incluida como parte del MOF (v2.0).
MOF y Ecore son conceptualmente muy similares, ambos basados en el concepto de clases con atributos tipados y operaciones con parámetros y excepciones. Además ambos soportan herencia múltiple y utilizan paquetes como mecanismo de agrupamiento.

2.3 Ejemplos de metamodelos

Es claro que UML no es el único lenguaje de modelado, sino que otros lenguajes específicos de dominio y lenguajes de modelado estandarizados pueden ser definidos sobre la base de MOF. MOF puede ser usado para definir metamodelos de lenguajes orientados a objetos, como es el caso de UML, y también para otros lenguajes no orientados a objetos, como es el caso de las redes de Petri o los lenguajes para servicios web.
En esta sección mostramos algunos ejemplos de metamodelos, en particular la figura siguiente muestra el metamodelo simplificado del lenguaje UML donde se especifica que un modelo UML está formado por paquetes (Package), donde cada paquete está integrado por clases (Clase) que poseen atributos (Attribute). Los atributos tienen un nombre (name) y un tipo (type), que puede ser un tipo de dato primitivo (PrimitiveDataType) o una clase.

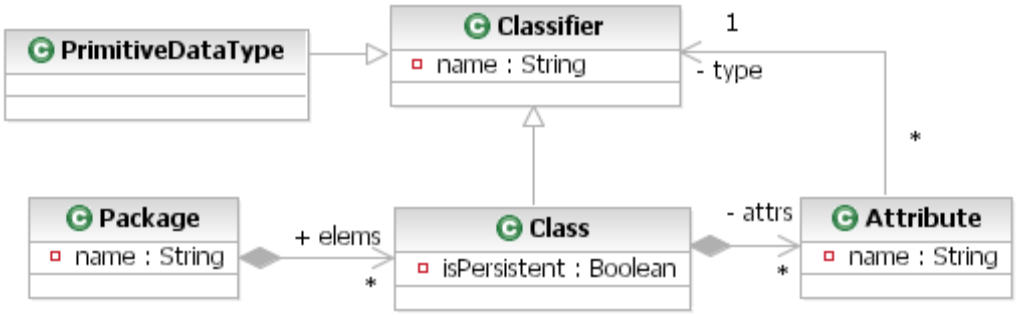

Figura 12 - Ejemplo metamodelo

El metamodelo del lenguaje RDBMS que mostramos en la figura siguiente indica que un modelo Relacional consta de esquemas (Schema), donde cada esquema está compuesto por tablas (Table) que tienen columnas (Column). Las columnas tienen un nombre (name) y un tipo de dato (type). La tabla tiene una clave primaria (pkey) y cero o más claves foráneas (fkeys). Cada clave foránea (Kkey) hace referencia a columnas en otras tablas.

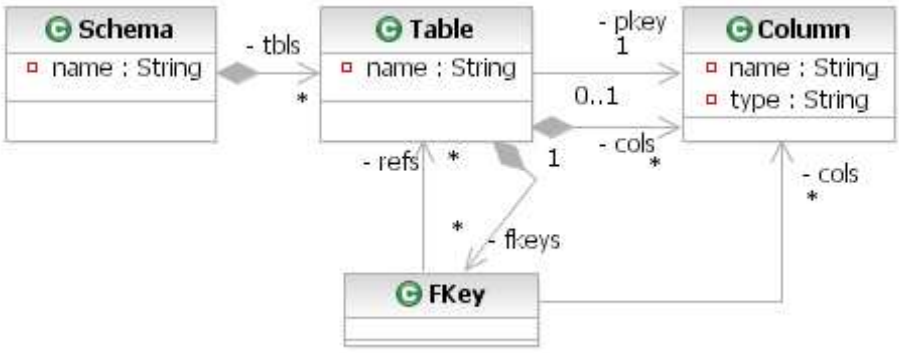

Figura 13 - Metamodelo RDBMS

2.4 El rol de OCL en el metamodelado

Los lenguajes gráficos de modelado han sido ampliamente aceptados en la industria. Sin embargo su falta de precisión ha originado la necesidad de utilizar otros lenguajes de especificación, como OCL, para definir restricciones adicionales. Con el uso de OCL se ofrece al diseñador la posibilidad de crear modelos precisos y completos del sistema en etapas tempranas del desarrollo.
Cualquier modelo puede ser enriquecido mediante información adicional o restricciones sobre los elementos del modelo. Estas restricciones pueden ser escritas en lenguaje natural, pero en ese caso pueden ser mal interpretadas y además no es posible chequear su validez en el modelo. Otra alternativa es utilizar lenguajes formales, sin embargo estos resultan difíciles de usar y no son bien recibidos por los modeladores de sistemas. OCL fue desarrollado para solucionar este problema. Es un lenguaje formal que por su naturaleza orientada a objetos resulta fácil de entender para las personas con conocimiento de lenguajes OO tales como JAVA o Smalltalk.
OCL es un lenguaje puro de especificación, el cual garantiza estar libre de efectos laterales.

Cuando se evalúa una expresión OCL, simplemente retorna un valor, sin hacer modificaciones en el modelo.
OCL permite especificar diferentes tipos de restricciones con las cuales podemos definir reglas de buena formación (conocidas como wfr por su nombre en ingles: well-formedness rules) para el modelo.

2.5 Resumen

En este capitulo hemos considerado los mecanismos para definir la sintaxis de los lenguajes de modelado. En particular hemos comenzado mencionando a la técnica clásica consistente en utilizar Backus Naur Form (BNF). Este método es útil para lenguajes textuales, pero dado que los lenguajes de modelado actuales en general no están basados en texto, sino en gráficos, es conveniente recurrir a un mecanismo diferente. En los últimos años se desarrolló una técnica específica para facilitar la definición de los lenguajes gráficos, llamada “metamodelado”. El metamodelo describe la sintaxis abstracta del lenguaje y constituye la base para el procesamiento automatizado de los modelos.
Esta técnica se basa en la idea de que un lenguaje de modelado también puede ser definido mediante un modelo conocido como metamodelo. Esta definición recursiva da lugar a una arquitectura de capas, en la cual cada nivel se instancia a partir del nivel inmediato superior. La arquitectura de cuatro capas de modelado es la propuesta del OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos. Los cuatro niveles definidos en esta arquitectura se denominan: M3, M2, M1, M0.

3 El lenguaje de modelado FlexAB

La herramienta FlexAB de Appliware es un software que le permite crear sus propias aplicaciones de sistemas de información de manera sencilla, ágil, rápida, y segura sin necesidad de programación. FlexAB tiene embebidas funcionalidades para la gestión de archivos en forma transparente.

FlexAB es una herramienta para la creación de aplicaciones sin programación, con la flexibilidad de orientada a objetos, y utilizando robustas bases de datos relacionales. El desarrollador interactúa solo con objetos, los datos y su estructura relacional son transparentes para el configurador, no perdiendo tiempo en relaciones, índices, tablas, etc. Esto hace que la creación de aplicaciones tenga una reducción en su tiempo de desarrollo. La utilización de Objetos posibilita la reutilización de los mismos en otras aplicaciones, así como su vinculación a otras aplicaciones, compartiendo la información en toda la familia de sistemas desarrollados con FlexAB de manera directa, sin interfaces o procesos intermedios.

FlexAB se apoya en una estructura especial, normalizada, fija, y optimizada de tablas sobre bases de datos relacionales que hace que se puedan almacenar objetos de manera transparente. Esta estructura de tablas y sus funciones asociadas para el manejo de la información es el OrDBC® (Object Relational Data Base Core) de Appliware.

FlexAB está desarrollado desde sus bases orientado a Objetos y a la aplicación de los objetos a los procesos de informatización. Debido a esto es una herramienta poderosa a la hora de informatizar cualquier proceso.

FlexAB es una combinación de una herramienta que brinda el marco para desarrollar una aplicación totalmente a medida, pero sobre una plataforma que no requiere programación y por lo tanto estandariza la creación de software. Que significa esto: que se eliminan las restricciones del software a medida, ya que las aplicaciones construidas con FlexAB no dependen de un programador, son abiertas, el know how de la aplicación queda en el cliente final, son flexibles, y totalmente ampliables sin sobrecostos.

¿Cómo Funciona FlexAB?

El enfoque de para la creación de aplicaciones con FlexAB es totalmente diferente, la idea central es orientar todo, no solo la programación y sus rutinas, a objetos. Es decir definir cuáles son los objetos relacionados con los procesos que necesitamos informatizar, y como interactúan estos objetos con su entorno (otros objetos). FlexAB nos permite crear objetos y manejarlos de una manera sencilla y sin necesidad de preocuparnos en como se almacenan los datos (en que tablas, y con que registros). El creador de la aplicación solo se preocupa sobre que tienen que hacer los objetos, y como se deben relacionar entre sí. Otro punto importante es que no existe programación. Es decir no es necesaria la generación de código para el manejo de objetos. La pregunta sería: ¿Si no hay código como genero los procesos que de los datos extraen información? La respuesta es que cada objeto tiene la capacidad de definir como interactúan su propios datos, y como interactúa él con los demás objetos. Esta funcionalidad se traduce en funciones FQL, es decir funciones SQL orientadas al manejo de información de objetos. Toda la lógica de la aplicación se hace por medio de

FQLs. Además se pueden definir operaciones que toman datos de los resultados de FQL y lo operan según se le indique en la configuración de las clases. También de manera similar se cargan las validaciones de integridad de cada objeto. Por otro lado, parte de la configuración permite el manejo de mensajes y la ejecución de programas externos los cuáles pueden interactuar con objetos FlexAB mediante la utilización de las funciones COM.

Después de haber explicado el significado y para que se usa la herramienta FlexAB a continuación se detalla el metamodelo de FlexAB.

La complejidad del metamodelo para el lenguaje flexAB se dividió organizando los conceptos en paquetes lógicos. Cada uno de estos paquetes agrupa metaclases con una alta cohesión entre ellas y un bajo acoplamiento con las metaclases de otros paquetes.
La siguiente figura muestra los paquetes que conforman la definición del metamodelo.

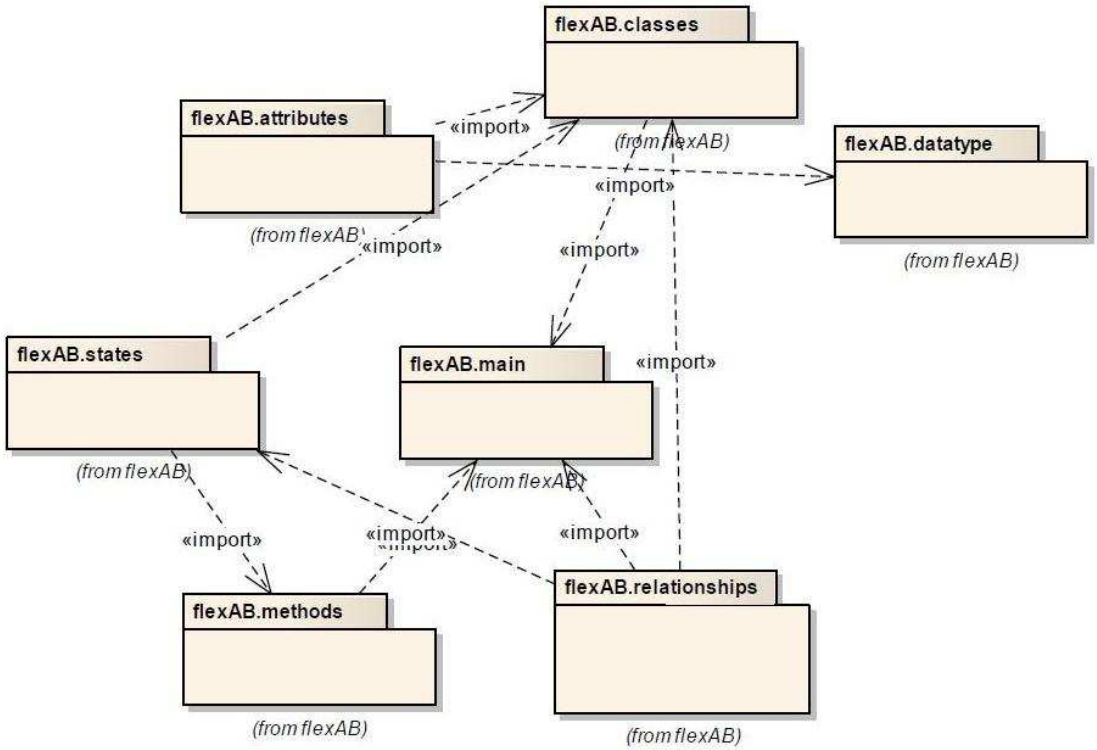

Figura 14 – Paquetes del modelo FlexAB

Main: Contiene las clases principales que definen un componente en flexAB, tales como Universo, Sistema, Aplicación.
Classes: Contiene la definición de las distintas clases definidas, como clase real, virtual, modulo
Relationships: Contiene las clases que definen las relaciones
Attributes: Contiene todas las clases y abstracciones que componen la definición de atributos para las clases flexAB

Methods: Contiene todas las clases y abstracciones que constituyen la definición de métodos para las clases flexAB
States: Contiene las clases necesarias para la definición de estados y comportamiento de los objetos de las clases.
Datatype: Contiene los tipos de datos definidos en flexAB.

3.1 Paquete Main

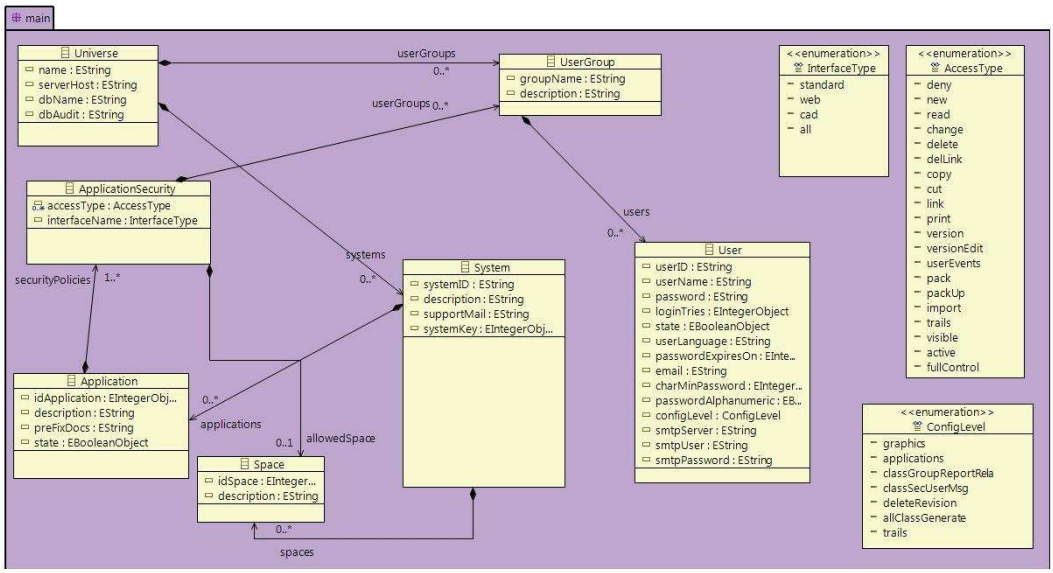

Figura 15 - Paquete Main

3.1.1 Application

Descripción:
Una aplicación es una instancia de un sistema, es decir que agrupa un conjunto de objetos que son instancias de clases que pertenecen a dicho sistema. Además, la definición de aplicaciones permite especificar los permisos de los grupos de usuario para acceder a los objetos (en término de los espacios del universo).

Atributos:
- idApplication: Integer Indica el identificador interno de la aplicación, el cual debe ser numérico.
- description: String Indica el nombre de la nueva aplicación.
- preFixDocs: String Indica un prefijo que se incorporará en el nombre de cada documento de la aplicación.
- state: Boolean Indica el estado en el que se encuentra la aplicación. Si el valor es true,

el estado es habilitado, en caso contrario es deshabilitado.

Asociaciones:
- applicationSecurity: ApplicationSecurity [1..*] Indica quienes podrán acceder a la aplicación y que operaciones tendrán disponibles. Para que una aplicación pueda ser accedida debe tener al menos una política de seguridad definida.
- system: System [1] Indica el sistema dentro del cual esta definida la presente aplicación.

3.1.2 ApplicationSecurity

Descripción:
FlexAB permite definir políticas de seguridad sobre una aplicación para restringir el acceso a determinado grupo de usuarios.

Atributos:
- accessType: AccessType [0..*] Indica el tipo de acceso a la aplicación tendrá el grupo de usuarios seleccionado.
- interfaceName: InterfaceType Indica la interface de trabajo con el sistema. (Todas- Estándar- Web – Cad).

Asociaciones:
- userGroups: UserGroup [1..*] Indica los grupos de usuarios que tendrá acceso a la aplicación.
- application: Application [1..*] Indica la aplicación que usa la política de seguridad.
- allowedSpace: Space Indica los espacios a los cuales tendrá acceso el grupo de usuarios definido. Si no tiene valor, significa que el grupo de usuarios tendrá acceso a todos los espacios definidos en el sistema.

Restricciones:
[1] Los tipos de acceso deny y full no se pueden combinar con otros tipos de acceso.
self.accessType -> select (a | a = #deny or a = #full) > 0 implies self.accessType -> size () = 1.

3.1.3 Space

Descripción:
Un espacio posee un conjunto de definiciones de clases y de relaciones entre éstas. Las relaciones entre clases son relativas al espacio, es decir que las definiciones de clases pueden especificar relaciones con otras clases de manera diferente en distintos espacios. Los espacios se crean dentro de los sistemas. Las aplicaciones creadas dentro de un mismo sistema, poseen los mismos espacios.

Atributos:
- idSpace: Integer Indica el identificador interno del espacio, el cual debe ser numérico.
- description: String Indica el nombre del espacio.

Asociaciones:
- system: System [1] Indica el sistema dentro del cual está definido el espacio**.**

3.1.4 System

Descripción:
Un sistema define un conjunto de clases y relaciones organizadas en espacios. La separación en distintos sistemas permite dividir la complejidad del dominio en partes que puedan ser tratadas con cierta autonomía. Esto quiere decir que un sistema tiene un conjunto de definiciones propias y comparte solo algunas con otros sistemas.

Atributos:
- system ID: String Indica el identificador interno del sistema. Ej: COM (Modulo Comercial).
- description: String Indica el nombre del nuevo sistema.
- supportMail: String Indica la cuenta de e-mail del soporte del sistema.
- systemKey: Integer Indica el número de llave del sistema. El sistema y las aplicaciones correspondientes, solo serán visibles si se encuentra presente una llave que contenga el System Key indicado.

Asociaciones:
- applications: Application [0..*] Indica las aplicaciones definidas en el sistema.
- spaces: Space [0..*] Indica los espacios definidos en el sistema.
- universe: Universe [1] Indica el universo donde se encuentra definido el sistema.

Restricciones:
[1] No pueden definirse aplicaciones con el mismo applicationNumber dentro de un sistema self.applications -> forAll (a1, a2 | a1.applicationNumber = a2.applicationNumber implies a1 = a2)
[2] No pueden definirse espacios con el mismo idSpace dentro de un sistema
self.spaces -> forAll (s1, s2 | s1.idSpace = s2.idSpace implies s1 = s2).

3.1.4 Universe

Descripción:
Un universo es una configuración de trabajo que especifica un Fileserver (para el manejo de los archivos asociados a sus objetos), un Servidor de bases de datos, y el nombre de dos bases de datos existentes en dicho servidor. Una de estas bases de datos es la que almacenará los objetos de las aplicaciones y la otra almacenará información de auditoria sobre dichos objetos.
Dentro de un determinado universo es posible configurar sistemas, espacios, aplicaciones y grupos de usuarios.

Atributos:
- name: String Indica el identificador interno del universo. Ej: FWK_FUM - serverHost: String Indica el servidor de bases de datos.
- dbName: String Indica el nombre de una base de datos que almacenará los objetos de las aplicaciones. Esta base de datos debe existir en el servidor indicado por serverHost.
- dbAudit: String Indica el nombre de una base de datos que almacenará información de auditoria de los objetos de las aplicaciones. Esta base de datos debe existir en el servidor indicado por serverHost.

Asociaciones:
- systems: System [0..*] Indica los sistemas definidos en el universo.
- userGroups: UserGroup [0..*] Indica los grupos de usuarios definidos en el universo.

Restricciones:
[1] No pueden definirse sistemas con el mismo SystemID dentro de un universo
self.systems -> forAll (s1, s2 | s1.systemID = s2.systemID implies s1 = s2)

[2] No pueden definirse grupos de usuarios con el mismo groupName dentro de un universo self.userGroups -> forAll (ug1, ug2 | ug1.groupName = ug2.groupName implies ug1= ug2)

3.1.5 User

Descripción:
FlexAB permite definir usuarios para restringir los permisos de operación sobre los objetos desde cualquier interfaz.

Atributos:
- userID: String Indica el identificador de usuario con el cual se ingresará al sistema. Por ej: iniciales del usuario, alopez.
- userName: String Indica el nombre del usuario.
- password: String Indica la clave con la que el usuario ingresará al sistema.
- loginTries: Integer Este campo indica la cantidad de intentos de acceso al sistema fallidos, luego de pasada dicha cantidad, el usuario queda deshabilitado.
- state: Boolean Indica si el usuario esta Habilitado o Deshabilitado. Los usuarios deshabilitados no pueden ingresar al sistema.
- userLanguage: String Indica el idioma que utilizará el usuario para visualizar mensajes, labels, etc.
- passwordExpiresOn: Integer Específica la cantidad de días a los que vence el password. El valor 0 (cero) significa que no tiene vencimiento.
- email: String A partir de la dirección de correo electrónico ingresada se puede enviar un e-mail al soporte del sistema, informándole el error ocurrido en la aplicación. Es la dirección de correo que le aparece a la persona que da soporte, cuando recibe el informe de error.
- charMinPassword: Integer Indica la cantidad mínima de caracteres que contendrá el password del usuario. Si es cero, no tiene límite.
- passwordAlphanumeric: Boolean Si el password es Alphanumeric, significa que debe contener obligatoriamente números y letras. En caso contrario puede contener cualquier carácter
- configLevel: ConfigLevel Indica el alcance que tendrá el usuario para configurar opciones dentro del configurador.
- smtpServer: String Indica el nombre de servidor de correo saliente
- smtpUser: String Indica el nombre de usuario para ingresar al servidor SMTP
- smtpPassword: String Indica la password para ingresar al servidor SMTP

Asociaciones:
- userGroup: UserGroups [0..*] Indica los grupos de usuarios a los cuales pertenece.

3.1.6 UserGroup

Descripción:
FlexAB permite grupos de usuarios para restringir los permisos de operación sobre los objetos desde cualquier interfaz de usuario. Las operaciones permitidas para el grupo se configuran de manera independiente en cada aplicación.

Atributos:
- groupName: String Indica el nombre de grupo de usuarios
- description: String Indica una descripción del grupo de usuarios definido.

Asociaciones:
- users: User [0..*] Indica el conjunto de usuarios pertenecientes al grupo.
- universe: Universe Indica el universo en el cual esta definido el grupo de usuarios.
- securityPolicy: SecurityPolicy [0..1] Indica las políticas de seguridad que hace referencia al grupo de usuarios.

3.1.7 Tipos de datos enumerativos del paquete flexAB.main

AccessType

Descripción:
AccessType es un tipo de dato enumerativo, cuyos valores indican que tipo de acceso a la aplicación tendrá el grupo de usuarios seleccionado. Define los siguientes valores literales:
- deny: Negado. Sin acceso.
- new: Permite crear nuevos objetos.
- read: Solo lectura.
- change: Solo cambios, no nuevos.
- delete: Eliminar objetos.
- delLink: Eliminar vínculos de objetos.
- copy: Copiar objetos.
- cut: Cortar objetos.
- link: Vincular objetos.
- print: Permiso de impresión.
- version: Revisionar objetos.
- versionEdit: Editar revisiones de objetos.

- userEvents: Ejecutar eventos de usuario.
- pack: Empaquetar objetos.
- packUp: Subir paquetes al sistema.
- import: Importar objetos.
- trails: Herramienta de caminos.
- visible: Define si el grupo de usuarios puede visualizar los atributos desde la edición.
- active: Define si el grupo de usuarios puede modificar los atributos desde la edición.
- fullControl: Acceso con control total.

Los permisos Deny y Full Control no se pueden combinar con otros permisos.

ConfigLevel

Descripción:

ConfigLevel es un tipo de dato enumerativo cuyos valores indican el alcance que tendrá el usuario para configurar opciones dentro del FlexAB-SCI (configurador). Define los siguientes valores literales:

- graphics: permite configurar los gráficos.
- applications: permite configurar sistemas y aplicaciones.
- classGroup – Report – Rela: permite configurar los grupos, reportes y relaciones estructurales de una clase, desde las respectivas solapas de la clase.
- classSec–User–Msg: permite configurar la seguridad, administrar usuarios y administrar las tablas de traducción de mensajes.
- deleteRevision: permite eliminar revisiones.
- all Class–Generate: permite generar clases.
- maintenance: permite utilizar el menú Maintenance del configurador, está destinado a aquellos usuarios que realicen solo el mantenimiento del sistema.
- trails: permite configurar Trails.

InterfaceType

Descripción:

InterfaceType es un tipo de dato enumerativo cuyos valores indican el tipo de interface de trabajo con el sistema. Define los siguientes valores literales:

- all: se trabajará con todas las interfaces.
- standard: se trabajará con una interface standard
- web: se trabajará con una interface web.
- cad: se trabajará con una interface cad.

3.2 Paquete Classes

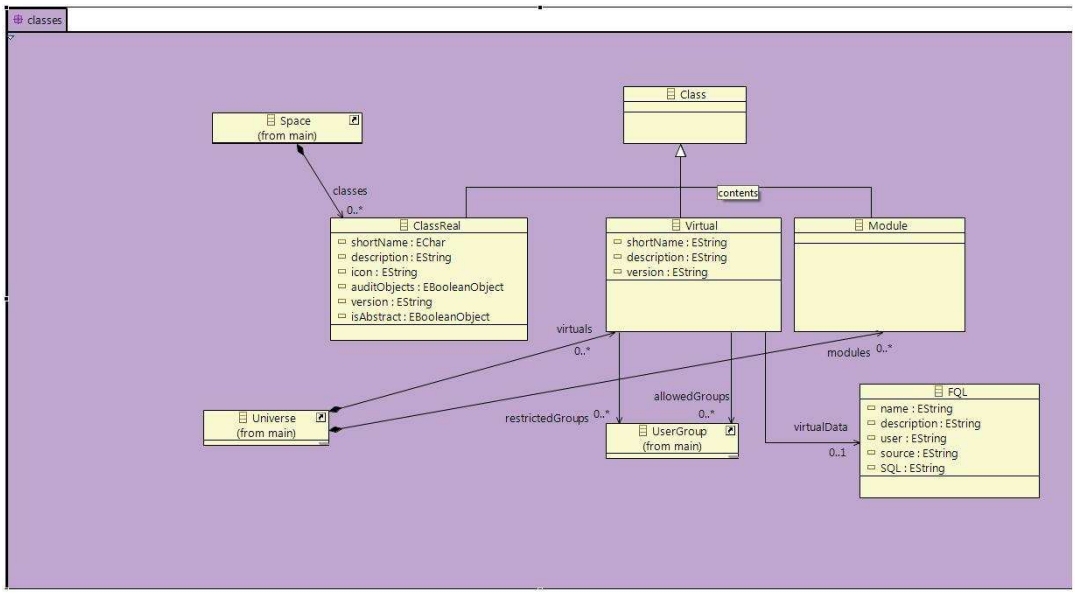

Figura 16 - Paquete Classes

3.2.1 Class

Descripción:

Una clase encapsula las abstracciones de datos y métodos que se requieren para describir el contenido y comportamiento de alguna entidad del mundo real. La clase es un modelo o prototipo que define los atributos y métodos comunes a todos los objetos de cierta clase. Una class es una clase abstracta raíz de la jerarquía de clases existentes en FlexAB.

Atributos:

- className: String Indica el identificador único de la clase. Ej: CODCOMERCIAL.

3.2.1 Module

Descripción:

Una clase de este tipo, contiene Métodos que son globales y pueden ser utilizados por las demás clases. No pueden verse en el árbol estructural.

En la versión actual, el Módulo SHAREDMETHODS contiene los métodos globales que existen en el sistema. No es posible crear otra clase de tipo Módulo.

Asociaciones:
- universe: Universe [1] Indica el universo donde esta definida la clase modulo.

3.2.2 Real

Descripción:
Una clase real permite generar una instancia de un objeto real en una aplicación.

Atributos:
- shortName: Char [5] Indica un identificador único por aplicación, compuesto por un conjunto de caracteres que permiten que el usuario reconozca las clases de cada aplicación. Ej: (TAR). Son cinco caracteres obligatoriamente. Por convención se utilizan tres letras entre paréntesis.
- description: String Indica un comentario descriptivo de la clase.
- icon: String Contiene el icono que se le asigna a la clase.
- auditObjects: Boolean Indica si se realizará la auditoría de los objetos de la clase.

Asociaciones:
space: Space [1] Indica el espacio donde esta definida la clase.

Space (from flexAB.main)
Asociaciones:
- classes: Real [0..*] Indica el conjunto de clases definidas en el espacio.

Universe (from flexAB.main)
Asociaciones:
- virtuals: Virtual [0..*] Indica el conjunto de clases virtuales definidas en el universo.
- modules: Module [0..*] Indica el conjunto de clases modulo definidas en el universo. En la versión actual, el Módulo SHAREDMETHODS contiene los métodos globales que existen en el sistema. No es posible crear otra clase de tipo Módulo.

3.2.3 Virtual

Descripción:
Estas clases se definen como vistas sobre las clases reales. La estructura de objeto que define una Clase Virtual se compone de información proveniente de objetos de otras clases. Por este motivo, estas clases no pueden ser instanciadas. Su finalidad es ser utilizada como medio para obtener ciertos datos.

Las clases virtuales se definen a nivel del universo (es decir, valen para todos los sistemas), y no a nivel de cada sistema-espacio como en el caso de las clases reales.

Atributos:

- shortName: Char [5] Indica un identificador compuesto por un conjunto de cinco caracteres obligatoriamente. Por convención se utilizan tres letras entre paréntesis.
- description: String Indica un comentario descriptivo de la clase.
- virtualData: FQL [1] Indica la consulta FQL que conformará los datos de la clase virtual.

Asociaciones:

- restrictedGroups: UserGroup [0..*] Indica el conjunto de grupos de usuarios que tienen el acceso restringido a la clase virtual.
- allowedGroups: UserGroup [0..*] Indica el conjunto de grupos de usuarios que tienen el acceso permitido a la clase virtual.

3.3 Paquete Relationships

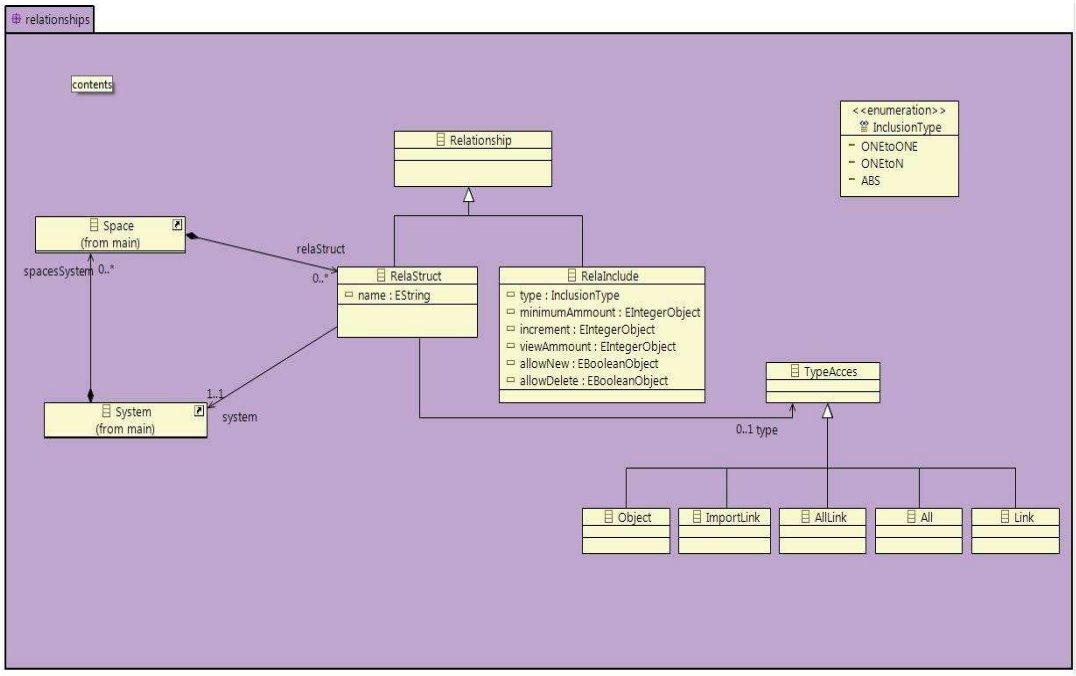

Figura 17 - Paquete Relationships

3.3.1 Link

Descripción:

Es una relación de conocimiento entre dos objetos. A diferencia de la relación Object, el

conocimiento no implica creación. Un objeto puede estar referenciado por múltiples objetos (múltiples padres en sentido estructural de tipo Link). Representa un vínculo (un acceso directo al objeto original).

3.3.2 Import Link

Descripción:
Es una relación de conocimiento entre dos objetos que permite acceso directo pero desde otra aplicación, es decir, el objeto original se encuentra en otra aplicación.

3.3.3 All Link

Descripción:
Es una relación de conocimiento entre dos objetos. Permite los tipos de relaciones Link o Import Link.

3.2.4 All

Descripción:
Es una relación de conocimiento entre dos objetos. Permite cualquier tipo de relación.

Real (from flexAB.classes)
Asociaciones:
- relationships: Relationships [*] Especifica el conjunto de relaciones que define la clase.

3.2.5 Object

Descripción:
La relación Object es la relación de un objeto con su padre estructural, es decir, con el objeto a partir del cual fue creado. Un objeto puede tener un solo padre en sentido de este tipo de relación.
En un mismo espacio, una clase no puede tener más de un tipo de relación con su clase padre. Existe solo una relación Object de la clase con su clase padre en todas las aplicaciones.
Una clase REAL puede participar en más de una Relación Estructural (relation type = Object), o sea que puede tener más de una clase padre. Por ejemplo la clase “Cable” puede ser hija de la clase “Tablero” y también de la clase “Consola”.

Existe solo una relación Object por aplicación, del objeto con su objeto padre.
El resto de las relaciones que pudieran tener ambos son de tipo Link y van en espacios diferentes.

3.3.3 RelaInclude

Descripción:
La relación de inclusión es aquella que verifica que sus componentes no pueden existir fuera del objeto compuesto (objeto padre). Por ello la clase incluída, no tiene existencia por sí misma, sino que es accesoria a su clase contenedora. A una clase incluída, no se la puede acceder directamente, esto solamente lo puede hacer su clase padre. Su principal característica es que cualquier objeto componente está completamente encapsulado, en el sentido de que su estado sólo puede ser alterado por eventos locales, adecuadamente coordinados con los eventos proporcionados por la interfaz de la agregación. En consecuencia, si tenemos una agregación inclusiva, los objetos componentes sólo podrán ser accedidos a través del objeto compuesto.
La relación de inclusión es una relación entre dos clases reales: una clase contenedora y una clase componente, teniendo la clase contenedora una relación estructural.
El tipo de relación puede ser: 1-1 (uno a uno) o 1-N (uno a muchos), especifica cuántos objetos componentes pueden relacionarse con un único objeto contenedor. Un objeto contenedor hace que se trate de forma atómica o encapsulada a los objetos componentes. Además, una clase real puede tener relaciones de inclusión con más de una clase.

Atributos:
- type: InclusionType Indica el tipo de relación, puede ser 1-1 o 1-N. En la relación 1-1, la clase padre solo podrá contener una clase incluída. En la 1-N la clase padre puede tener varias. También aparece como opción, “ABS”, que es para una clase abstracta.
- minimumAmmount: Integer
- increment: Integer
- viewAmmount: Boolean
- allowNew: Boolean
- allowDelete: Boolean

Nota: una clase incluída, puede configurarse como clase padre en una relación estructural sin que haya restricciones desde la configuración. Esto sería un error conceptual, dado que las clases estructurales hijas de la incluída no serían visibles y no podrían operarse.

Restricciones:
[1] Una clase incluída puede contener otras clases incluídas, pero no puede contenerse a si misma como incluída.
self.class <> self.classFather
[2] En la relación 1-1, la clase padre solo podrá contener una clase incluída. En la 1-N la clase padre puede tener varias.
self.type = #1-1 implies

3.3.4 RelaStruct

Descripción:
La Relación Estructural difiere de la inclusiva por el hecho de que existe una relación entre los objetos de las clases participantes, sin tener una inclusión completa de los objetos componentes en el objeto compuesto. Los objetos componentes no están encapsulados en el objeto compuesto (objeto padre) y su comportamiento es independiente del comportamiento del padre.
Es la relación entre clases reales dentro de un sistema. Una clase real puede participar en más de una Relación estructural, es decir que puede tener uno ó más padres estructurales.
En la relación estructural se distinguen dos roles: una clase padre y una clase hijo, donde los objetos relacionados tienen comportamiento independiente el uno del otro (diferencia con respecto a la relación de inclusión).

Atributos:
- name: String Indica el nombre de la relación, es un string para el análisis del sistema. Por ej: ES UN, HIJO DE, ESTA EN, etc.

Asociaciones:
- system: Indica el sistema al que pertenece.
- space: Indica el espacio al que pertenece la relación estructural
- group Father: Indica el grupo al que debe pertenecer el objeto padre para poder generar la relación. En caso de que hubiera objetos dentro de un padre con un determinado grupo válido para la relación, y luego el grupo del mismo cambiara a un grupo no válido para la relación estructural, los objetos hijos creados quedarán, pero no se podrán crear nuevos, debido justamente al filtro establecido.
- fatherState: Indica el estado al que debe pertenecer el objeto padre para poder generar la relación. En caso de que hubiera objetos dentro de un padre con un determinado estado válido para la relación, y luego el estado del mismo cambiara a un estado no

válido para la relación estructural, los objetos hijos creados quedarán, pero no se podrán crear nuevos, debido al filtro establecido.

3.3.5 Relationship

Descripción:
Relationship es una clase abstracta raíz de las posibles relaciones existentes en flexAB.

Asociaciones:
- class: Real [1] Indica la clase poseedora de la relación.
- classFather: Real [1] Indica la clase padre en la que se desea incluir a la clase actual.

Space (from flexAB.main)
Asociaciones:
- relaStruct: RelaStruct [*] Especifica el conjunto de relaciones estructurales declaradas dentro del espacio.

Restricciones:
[1] Una clase no puede tener mas de un tipo de relación con su clase padre
en un mismo espacio
self.relationships -> select (r | r.isOclKindOf(RelaStruct)) -> forAll (r1, r2 | r1.classFather
= r2.classFather implies r1.space <> r2.space)
En un mismo espacio, una clase no puede tener más de un tipo de relación con su clase padre.

3.3.6 Type

Descripción:
Type es una clase abstracta raíz de los posibles tipos de relaciones estructurales entre dos clases reales.

3.3.7 Tipos enumerativos del paquete flexAB.relationships

InclusionType

InclusionType es un tipo de dato enumerativo cuyos valores indican el tipo de una relación de inclusión. Define los siguientes valores literales:
- 1-1
- 1-n
- ABS

3.4 Paquete Attributes

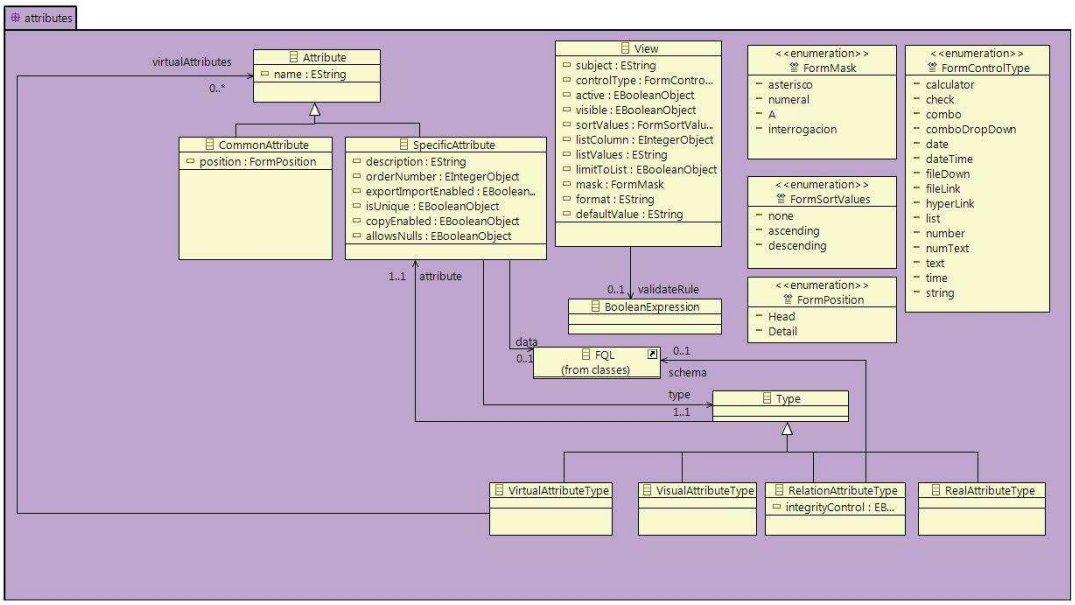

Figura 18 - Paquete Attributes

3.4.1 Attribute

Descripción:

Los atributos son las características o propiedades de las clases de objetos, es la información de interés que las describen. Todo objeto puede tener cierto número de propiedades, cada una de las cuales tendrá, a su vez, uno o varios valores.

En Orientación a Objetos, las propiedades corresponden a las clásicas "variables" de la programación estructurada. Son, por lo tanto, datos encapsulados dentro del objeto, junto con los métodos y las relaciones (punteros a otros objetos).

Las propiedades de un objeto pueden tener un valor único o pueden contener un conjunto de valores más o menos estructurados. Attribute es la clase abstracta raíz de todos los atributos.

Atributos:
- name: String Indica el identificador único del atributo en la clase. Ej.: descripción. El nombre no debe contener espacios, ni caracteres especiales, este identificador conformará el id del atributo en la tabla.

3.4.2 BooleanExpression

Regla de validación en término de operaciones de comparación, conjunción, disyunción y operadores sobre String (Begins, Ends, Contains).

3.4.3 CommonAttribute

Descripción:
Los atributos comunes están siempre presentes para cualquier clase que se configure. A estos atributos no se los puede remover, ni pueden agregarse nuevos.

Atributos:
- position: Position Indica en qué sección del formulario se muestra el atributo. Puede tomar los valores HEAD o DETAIL.

Real (from flexAB.classes)
Restricciones
[1] Una clase real debe tener cuatro atributos comunes llamados idGroup (es uno de los grupos definidos para la clase), idOBJECT (es un identificador único dentro de la aplicación. Se genera de acuerdo a los tipos definidos en su clase), idState (es el estado actual del objeto, un estado entre los estados definidos en la clase) y Tag
(self.commonAttributes->select (a| a.name = idGroup) -> size = 1)
&&
(self.commonAttributes->select (a| a.name = idObject)-> size = 1)
&&
(self.commonAttributes->select (a| a.name = idState) -> size = 1)
&&
(self.commonAttributes->select (a| a.name = tag) -> size = 1)

3.4.3 RealAttributeType

Descripción:
Se define este tipo de atributo cuando estamos dando de alta un atributo común, que puede llevar cualquier tipo de dato (integer, double, boolean, long, text, etc.). El valor de Data Type puede ser cualquiera de los definidos.

3.4.4 RelationAttributeType

Descripción:
Se define este tipo de atributo cuando el mismo necesita traer un dato de otro atributo que se encuentra en otra clase, es decir, cuando necesitamos recuperar información que se encuentra en atributos de otras clases.
Estos atributos tienen la característica de tener asociadas dos consultas SQL que nos permiten traer información desde otros atributos en otras clases.
En la tabla correspondiente a la clase que contiene a los atributos de tipo relation, por cada uno de estos atributos se crea un campo. Este campo contiene el IdInternal del objeto al que apunta el atributo Relation.

Atributos:
- integrityControl: Boolean Integridad referencial con los objetos del universo. Si se utiliza una base de datos externa, no se debe tildar la opción, porque no se realiza el control.

Asociaciones:
- schema: flexAB.classes::FQL Genera la vista y no puede contener XQL

Restricciones:
[1] Un atributo de tipo relationAttribute debe llevar como tipo de Data Type un char (n) y como controlType un Combo o Lista.
self.attribute.datatype.isOclKindOf(Char(n)) &&(self.attribute.view.controlType = #Combo or
self.attribute.view.controlType = #List)

3.4.5 SpecificAttribute

Descripción:
Un SpecificAttribute representa los atributos específicos de la clase, los que define el usuario configurador a la hora de la realización de un sistema

Atributos:
- description: String Indica el nombre descriptivo del atributo. Ej.: Tarea del equipo de trabajo.
- orderNumber: Integer Indica el número de orden que tendrá el atributo en el formulario.
- exportImportEnabled: Boolean Si tiene el valor true, significa que el valor de este atributo se puede importar y exportar. De lo contrario no puede realizar ninguna de las dos operaciones.
- isUnique: Boolean Si tiene el valor true, el valor del atributo no se puede repetir en ningún objeto de la clase.
- copyEnabled: Boolean Si tiene el valor true significa que cuando copiemos un objeto (a uno nuevo), el atributo tendrá el mismo valor que en el original. Si CopyEnabled tiene el valor false, al copiar el objeto, el valor del atributo estará en blanco.
- allowsNulls: Boolean Si tiene el valor false, el atributo no acepta valor nulo.

Asociaciones:
- dataType: String Indica tipo de dato del atributo Ej.: boolean, integer, long, autonumber, single, double, char, text, date, time, datetime, numtext (traduce un número a su escritura en texto), etc. Su determinante es el Type.
- type: Type [1] Indica el tipo de atributo, puede ser real, virtual, relation o visual.
- data: flexAB.classes::FQL Hay determinados tipos de atributos, por ejemplo los atributos de tipo RELATION que pueden poseer un FQL asociado. Se puede crear un FQL o tomar uno existente. Puede contener XQL, y es la que realiza la selección los datos.

3.4.6 Type

Descripción:
Type es la superclase abstracta raíz de los tipos de atributos existentes en flexAB.

3.4.7 View

Descripción:
Esta clase define la vista estándar de los objetos de una determinada clase.

Atributos:
- subject: String Indica el label que tendrá el atributo cuando se muestre en el formulario.

- controlType: ControlType Indica el tipo de control que tendrá el atributo cuando se muestre en el formulario.
- active: Boolean Indica si el atributo va a estar activo en el formulario.
- visible: Boolean Indica si el atributo va a estar visible en el formulario.
- sortValues: FormSortValues
- listColumn: Indica el número de orden que aparece el atributo, en la sección Objeto de la Interfaz Estándar FlexAB-SUI. Los atributos que tienen un valor distinto de cero para este campo se muestran en la lista de objetos de la ventana principal de la Interfaz Estándar FlexAB-SUI. Se pueden mostrar hasta 14 atributos. Este valor debe ser distinto para cada atributo (a excepción del cero). Los primeros cuatro atributos con ListColumn distinto de cero, se muestran en la Barra de Objeto de la Interfaz de Usuario, al seleccionar en el árbol un objeto de la clase.
- listValues:String [*] Indica la lista de valores posibles, en caso que la edición del valor esté restringida a un conjunto de valores.
- limitToList: esta opción limita los valores del combo, es decir, los únicos valores que podrá contener, son los correspondientes a la lista listValues.
- mask: Mask Indica la máscara que tendrá el atributo en el formulario, solo si es necesario. Máscara de entrada de datos (cantidad de caracteres, alfabéticos o numéricos)
- validateRule: BooleanExpression Indica la regla de validación que tendrá el atributo en el formulario, solo si es necesario. Es el campo en el cual se puede validar un atributo de una clase. Regla de validación en término de operaciones de comparación, conjunción, disyunción y operadores sobre String (Begins, Ends, Contains).
- format: String Indica un formato que para dar a nuestro atributo. El formato depende fuertemente del tipo de Dato.
- defaultValue: String Indica el valor por defecto.

3.4.8 VirtualAttributeType

Descripción:

Este tipo de atributo cambia de valor dependiendo de la información de otras Clases en el Universo, es la forma en que los Objetos se nutren de lo que pasa en el Universo, de la información que brindan las demás Clases. Todos los objetos pueden acceder a la información de los demás objetos, solo deben suscribir a la información de las Clases que le interesa. Este tipo de atributo tiene asociado una lista de clases, cualquier objeto que modifique alguna de las clases de esta lista, va a generar la actualización del valor del atributo virtual. Queda guardado en la base de datos, pero es el resultado de un cálculo. Cuando un objeto de estas clases asociadas se graba, se produce el cambio en el atributo virtual.

Si en la columna “Virtual Attribute” se coloca un “*” (asterisco), se actualizarán todos los objetos de la clase que contenga el atributo virtual.
Si en la columna “Virtual Attribute” se indica un determinado atributo (debe ser de tipo Relation), ese atributo posee internamente la información del objeto cuyo atributo virtual se debe actualizar. Esto es así, porque el atributo de la clase asociada, por ser de tipo “Relation”, tiene esta cualidad.
El valor del atributo virtual queda guardado en la base de datos, pero es el resultado de un cálculo.

Restricciones:
[1] Un atributo de tipo virtualAttribute puede llevar cualquier tipo de Data Type menos FILELINK – FILEDOWN – NUMTEXT.
Not (self.attribute.datatype.isOclKindOf(FILELINK) or
self.attribute.datatype.isOclKindOf(FILEDOWN) or
self.attribute.datatype.isOclKindOf(NUMTEXT))
[2] El FQL del atributo virtual, debe contener obligatoriamente los campos IdInternal y Value, y no puede contener expresiones XQL.
[3] Los atributos virtuales no disparan eventos, por lo que no podrán utilizarse por ejemplo en un evento FieldChange para utilizar métodos.

3.4.9 VisualAttribute

Descripción:
Se define este tipo de atributo cuando es necesario que el usuario de la aplicación observe alguna información de otras clases, por ejemplo la descripción del campo. Los atributos visuales, obtienen datos de otro lado como por ejemplo de un cálculo; no se almacenan en la base de datos.
Los atributos visuales, se cargan desde un método Operation; muestran por ejemplo el resultado de un cálculo.
Estos atributos se manejan a través de métodos. Por lo tanto, los valores se le asignan por métodos. Un método Operation asociado a un evento cualquiera, puede cargarle algún valor al atributo.
Es muy común en la práctica, mostrar al usuario un atributo visual, y antes de la grabación del objeto, en el evento BeforeSave de la clase, por medio de un método asignar el valor de dicho atributo a un atributo común y de esa forma grabarlo en la base de datos.

Atributos

Restricciones:

[1] Un atributo de tipo visualAttribute puede llevar cualquier tipo de Data Type menos FILELINK FILEDOWN – NUMTEXT.

Not (self.attribute.datatype.isOclKindOf(FILELINK) or self.attribute.datatype.isOclKindOf(FILEDOWN) or self.attribute.datatype.isOclKindOf(NUMTEXT))

3.4.10 Tipos de datos enumerativos

FormControlType

Control Type: se selecciona el tipo de control que tendrá el atributo en el formulario.

- calculator
- check
- combo
- comboDropDown
- date
- dateTime
- fileDown: se utiliza cuando es necesario enviar hacia otro sector de la empresa, o algún otro lugar, los archivos de información que tiene nuestro sistema. Debido a que los mismos se encuentran resguardados en nuestro sistema por el FileServer, no podemos acceder irectamente a ellos. La manera de obtenerlos es a través de un atributo FileDown. Dicho atributo, trabaja por medio de una consulta FQL, que obtiene los archivos que se le indiquen. Luego el usuario, ve presentados en una lista dichos archivos y selecciona los que necesita.

El List Column de un atributo Filedown debe ser 0.

Cada vez que se grabe el objeto que contiene al atributo FileDown y se hayan seleccionado archivos dentro del atributo, se creará un archivo ZIP (comprimido) conteniendo los archivos elegidos.

- fileLink: Un atributo de tipo FileLink, permite incorporar un archivo al objeto. Este archivo, se guardará junto con el objeto, y el usuario podrá acceder a dicho archivo, solamente desde el sistema a través del objeto.

El archivo será removido de su ubicación original, y se enviará al FileServer.

A este tipo de atributo, se le puede indicar que tipos de archivo puede aceptar.

Por ejemplo si se trata de un atributo destinado a contener informes, se le puede indicar que solamente acepte archivos que posean la extensión ".doc".

En caso que se seleccione fileLink, se debe indicar las extensiones de los archivos que

aceptará el filelink. Para aceptar cualquier tipo de archivo, se debe indicar "*.*"
- hyperLink
- list
- number
- numText
- text
- time
- string

FormMask

Representa la máscara que tendrá el atributo en el formulario, solo si es necesario.
Todas las mascaras tienen una longitud fija de un solo caracter.
- * : Permite un carácter de cualquier tipo.
- # : Permite un carácter de tipo numérico
- A : Permite un carácter de tipo letra.
- ? : Permite un carácter símbolo

FormPosition

Representa la ubicación que tendra el elemento dentro del formulario.
- Head: el elemento se ubica en el encabezado del formulario
- Detail : el elemento se ubica en el cuerpo del formulario

FormSortValues

Representa el orden que tendrá una lista de atributos dentro del formulario
- None:
- ascending
- descending

3.5 Paquete Methods

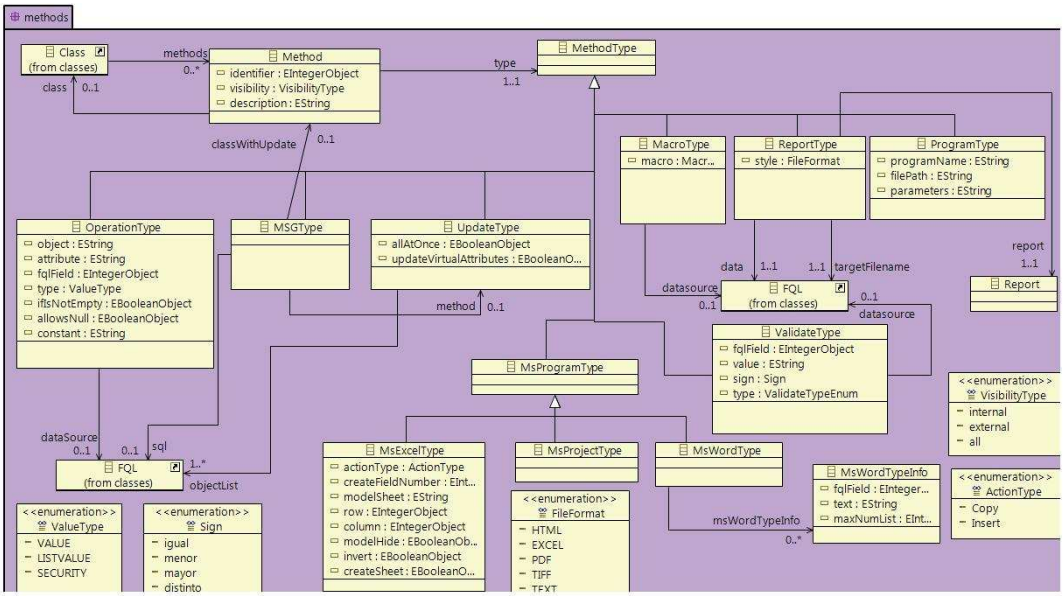

Figura 19 - Paquete Methods

3.5.1 MSGType

Descripción:

Este método invoca métodos de otras clases, ejecutando eventos de objetos de otras clases. Por medio de un mensaje un objeto puede disparar un evento de otro objeto.

En la versión actual, la única restricción, es que los métodos invocados sean de tipo UPDATE.

Este método se debe definir siguiendo los siguientes pasos:

- Se define un método de tipo MSG en la clase A, asociado a un evento e1.
- El FQL asociado debe recuperar un conjunto de IdInternal+IdFather
- Se define una clase B y un evento e2, para indicar que será disparado dicho evento en los objetos de dicha clase.
- En la clase B se define un método (de cualquier tipo) asociado al evento e2.

Asociaciones:

- classWithUpdate: Class Indica la clase que tiene el método Update a invocar
- method: Update Indica el método Update invocado
- Sql: FQL Indica el FQL que devuelve una lista de objetos para los cuales se invocará el método. Se debe indicar el IdInternal y el IdClass

3.5.2 MacroType

Descripción:
Los métodos MACRO permiten ejecutar una secuencia de funciones COM de FlexAB sobre el resultado de una consulta FQL. Estos métodos permiten llevar a cabo operaciones tales como modificar las relaciones con otros objetos, bloqueo, versionado y empaquetado de objetos, entre otras.
Estas operaciones permiten modificar atributos del XML de un objeto que no pueden modificarse desde un método operación
- con las macros puedo "modificar el objeto" (incluídos, relaciones, etc…)
- con los métodos operación puedo "modificar atributos del objeto" (cambiar un valor de atributo).
Para esto, el método MACRO mantiene un XML de trabajo (XML auxiliar) sobre el cual se llevan a cabo modificaciones intermedias.
Todas las funciones de la MACRO se ejecutan sobre un XML auxiliar. Por lo tanto, si se desea hacer una operación con otro objeto (por ejemplo crear otro objeto en otro espacio), es necesario poner en el XML auxiliar el otro objeto para ejecutarle funciones.
El esquema resultante de la consulta FQL debe incluir los campos requeridos en la primera función que se utilice. Es decir que si la función f tiene una lista de parámetros (nom1,…, nomk) entonces el esquema de la consulta FQL debe incluir al menos los campos nom1 … nomk.

Atributos:
- macro: MacroFunction Indica las funciones que ejecuta la Macro.

Asociaciones:
- dataSource: FQL Indica la consulta FQL, que se utiliza para obtener los parámetros requeridos en las funciones que utiliza la Macro.

3.5.3 Method

Descripción:
Los métodos son aquellas funciones que permite efectuar el objeto, y que nos rinden algún tipo de servicio durante el transcurso del programa. Determinan a su vez como va a responder el objeto cuando recibe un mensaje.
Un método es un programa asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena mediante un "mensaje". De esta forma el objeto se activa y responde al evento según lo determinado en el método.

Existen métodos que son GLOBALES y métodos que son propios de cada clase. Los métodos GLOBALES pueden ser utilizados por las clases, pero no pueden ser modificados desde las mismas. Deben modificarse directamente desde el Editor de Métodos. Estos métodos GLOBALES se encuentran contenidos en el módulo SHAREDMETHODS.

Atributos:
- identifier: String Indica el identificador del método
- visibility: VisibilityType Corresponde al tipo de ejecución del método. Puede ser Internal, External o All.
- descripción: String Indica la denominación del método.

Asociaciones:
- class: Class [1] Indica la clase a la que pertenece el método. Puede pertenecer a una clase en particular, o a una clase módulo.
- type: MethodType Indica el tipo de método.

3.5.4 MSExcelType

Descripción:
Método que permite abrir un archivo Excel y cargarle el contenido con los datos que se le indiquen.

Atributos:
- action Type: ActionType Define si en la planilla Excel, los datos van a ser insertados o copiados.
- createFieldNumber: Integer Se utiliza conjuntamente con Create Sheet. Es un valor numérico que representa el número de columna devuelto en el FQL del Data Source.
- modelSheet: String Indica el nombre de la hoja de la planilla en la que trabajará.
- row: Integer Indica el número de fila en la cual comenzarán a insertarse o copiarse los datos.
- column: Integer Indica el número de columna en la cual comenzarán a insertarse o copiarse los datos.
- modelHide: Boolean
- invert: Boolean
- createSheet: Boolean Se utiliza conjuntamente con Create Field Number. Crea una hoja en la planilla por cada valor devuelto en la columna elegida en Create Field Number. En cada hoja, pone el valor correspondiente a cada registro devuelto, y nombra

a la hoja con el valor que existe en el campo correspondiente al Create Field Number elegido.

3.5.5 MsProgramType

MsProgramType es una clase abstracta raíz de los programas de MS.

Asociaciones:
- sourceFilename: FQL Determina el nombre del archivo origen, con qué nombre se abrirá el archivo para trabajar.
- targetFilename:FQL Determina el nombre del archivo destino, con qué nombre será guardado el archivo.
- dataSource: FQL [*] Indica las "n" consultas FQL asociadas. Data Source es el nombre de las consultas.

3.5.6 MsProjectType

Descripción:
Permite abrir un archivo Project y cargarle el contenido indicado en la consulta FQL.

3.5.7 MsWordType

Permite abrir un archivo Word y cargarle el contenido con los datos que se le indiquen.

Asociaciones:
- msWordTypeInfo: MsWordTypeInfo [*] Indica una lista ordenada de información necesaria para la edición del archivo word.

3.5.8 MsWordTypeInfo

Esta clase tiene la información para cambiar el contenido del archivo word.

Atributos:
- fqlField: Integer La consulta SQL puede devolver "n"' número de columnas. En este campo se especifica cual de las columnas del FQL es la que va a copiar en la marca incorporada en el Word.
- text: String Marca o etiqueta incorporada en el archivo de Word, en donde se copiará el valor btenido de la consulta SQL. Esta marca, es la que será reemplazada con el texto

que devuelva la consulta en la columna indicada en Fql Field.
- maxNumList: Integer Número máximo de registros que utilizará del resultado de la consulta.

3.5.9 OperationType

Descripción:
Una operación en un método que permite manipular el estado interno, es decir, los atributos de un objeto. Proveen un mecanismo para hacer obtener resultados provenientes de cualquier objeto y modificar algún atributo del objeto.
Este método genera un cambio de valor del XML activo. Lo que hace es, mediante una consulta SQL, traer las 'n' columnas del resultado, y la columna correspondiente (indicada por Fql Field) la copia en el atributo deseado.

Atributos:
- object: String referencia dentro del XML al objeto en sí (OBJECT), a los objetos incluídos (INCLUDE), al padre del objeto (FATHER), a la clase abstracta que contenga el objeto ABSTRACT) , y al primer valor que encuentre en el XML del objeto (ABSOLUTE VALUE)
- attribute: String Indica el nombre del atributo que tomará el resultado que nos devuelve la consulta FQL. Este atributo es en relación valor elegido en campo Object.
- fqlField: Intenger como el resultado del FQL tiene 'n' columnas, se especifica cual es la columna que se desea copiar como valor correspondiente al campo de la columna Attribute.
- type: ValueType Indica si la consulta FQL, devolverá un valor (VALUE), devolverá una lista de valores (LISTVALUE) o se aplicará sobre el nivel de seguridad del atributo (SECURITY).
- ifIsNotEmpty: Boolean Significa que solo se ejecutará cuando el atributo no contenga un valor, o sea siempre que esté en blanco.
- allowsNull: Boolean indica si acepta valores nulos
- constant : String Si existe un valor en esta columna, se tomará el mismo en vez del valor devuelto por el FQL. En este caso no es necesario ingresar una consulta FQL. Si la columna Constant está en blanco, se exigirá una consulta
FQL. El valor de la columna es cualquiera que el usuario defina.

Asociaciones:
- dataSource: FQL [0..1] Los métodos deben poseer un FQL asociado. Se puede crear un FQL o levantar uno existente.

3.5.10 UpdateType

Descripción:
Este método es el único que modifica directamente los datos en la Base de Datos. No se utiliza el XML de los objetos. Obligatoriamente cambia de estado al objeto.

Atributos:
- objectList: FQL Lista de Objetos de la clase, sobre los que se realizará el Update (esto depende si el evento que lo dispara es OnMessage o UserEvent). La sentencia SQL debe contener obligatoriamente IdInternal. No puede contener XQL, ya que no se utiliza el XML de los objetos.
- AllAtOnce: Boolean Si el valor es verdadero, los objetos serán actualizados en conjunto, de una sola vez. De lo contrario, se realizarán actualizaciones individuales por cada objeto.
- UpdateVirtualAttributes: Boolean indica que se actualicen los atributos virtuales que dependan de la grabación del objeto sobre el cual corre el método. Al actualizar un objeto que estuviera referenciado en otro de otra clase (por medio de un atributo relation) y esta última clase tuviese un atributo virtual que se modifica con la grabación del objeto relacionado, solo se realizará esto si se tilda la opción.

Asociaciones:
- nextState: State Estado al que pasará el objeto. Este estado solamente puede contener métodos de tipo Update, ningún otro. Esto se debe a que no tiene el XML del objeto para ejecutar cualquier otro método. En Initial tampoco puede haber métodos que no sean Update.
- Attribute [*] Indica el atributo que será modificado.
- FQLField [*] Indica el campo de la sentencia SQL que modificará el campo.

Restricciones
- Cuando la clase tiene métodos update, en el estado INITIAL no pueden existir métodos.
- En el estado al que pasará el objeto, solo puede haber métodos Update.
- No se puede definir en SHAREDMETHODS; siempre este método debe pertenecer a una clase.
- SharedMethods no puede definir métodos update.
- Es un método público, por lo cual está “visible” para otras clases.
- Este método solamente puede ser invocado por eventos onMessage y UserEvent
- Cuando es por UserEvent, se ejecuta solo el método update que corresponde al objeto

que dispara el UserEvent. Este método "conoce" el IdInternal del objeto que lo está disparando Cuando es por OnMessage, puede ser para una lista de objetos. Por lo tanto se ejecutan los métodos Update de cada uno de los objetos de la lista
- No se puede modificar el IdObject de los objetos.

3.5.11 ProgramType

Descripción:
Este tipo de método permite la ejecución de un programa externo.

Atributos:
- programName: String Indica el nombre del programa que se desea ejecutar.
- filePath: String Indica la ruta donde se encuentra el programa.
- parameters: String Indican los parámetros que se le pasan al programa.

3.5.12 ReportType

Descripción:
Método que permite la creación de un reporte.

Atributos:
- editFileAfterFinish: Boolean Si el valor es verdadero, se abre el archivo después de la ejecución del método.
- STYLE: FileFormat Indica el formato de salida que tendrá el reporte.

Asociaciones:
- Data: FQL Indica la consulta FQL que será la que obtenga los datos para volcar al reporte.
- Target Filename: FQL Indica la consulta FQL que contiene ruta y nombre del archivo destino del reporte
- Report: Report Indica el reporte original, que se crea desde el Editor de Reportes.

3.5.13 ValidateType

Descripción:
Este método permite realizar validaciones sobre los atributos del objeto al que se aplica así como los atributos de cualquier otro objeto en el universo.
Una validación verifica que el resultado de una consulta FQL cumpla una cierta

condición y en función de este resultado puede configurarse la lógica del objeto. Para esto, la configuración de un método validación requiere:

Atributos:
- fqlField: Integer Indica el numero de columna de la que tomara el valor como resultado de la consulta
- type: ValidateType El Type determina si lo que va a validar es, cantidad de Registros (REC) , comparación del resultado con un valor numérico (NUM) o comparación del resultado con un valor de texto (TXT) .
- sign: Sign Indica el signo de comparación: =, <, >, <>, <=, >= utilizado en la validación
- value: String Es el valor por el cual se compara para la validación.

Asociaciones:
- dataSource: FQL Indica la consulta FQL asociada al método que nos devolverá el valor que necesitamos para realizar la validación.

Virtual (from flexAB.classes)
Restricciones:
[1] Una clase virtual no puede contener métodos self.methods -> size = 0

3.5.14 Tipos de datos enumerativos de flexAB.methods

VisibilityType

VisibilityType es un tipo de dato enumerativo que indica tipo de ejecución de un método.
Define los siguientes literales:
- Internal: cuando el método sólo puede ser ejecutado desde la clase (privado).
- External: cuando el método puede ser ejecutado desde afuera (público).
- All: puede ser ejecutado de modo interno y externo.

ValueType

ValueType es un tipo de dato enumerativo que indica un tipo de valor. Define los siguientes literales
- VALUE: representa un solo valor
- LISTVALUE: representa una lista de valores

- SECURITY: representa el nivel de seguridad

Sign

Sign es un tipo de dato enumerativo que indica las el signo de comparación utilizado en una validación. Define los siguientes literales:
=, <, >, <>, <=, >=

ValidateType

ValidateType es un tipo de dato enumerativo que indica que es lo que se va a validar. Define los siguientes literales:
- REC: determina que va a validar la cantidad de Registros
- NUM determina que va comparar el resultado con un valor numérico
- TXT: determina que va comparar el resultado con un valor de texto

MacroFunction

MacroFunction es un tipo de dato enumerativo que indica las funciones que ejecuta la Macro. Define los siguientes literales:
- actualOBJECTToXML: Copia el XML del objeto actual, al XML auxiliar. No necesita parámetros.
- actualXMLtoOBJECT: Copia el XML auxiliar, al XML del objeto actual
- OBJECTlock: Bloquea un objeto cualquiera. Necesita idInternal como parámetro.
- OBJECTunlock: Desbloquea un objeto. Necesita idInternal como parámetro.
- OBJECTerase : Elimina un objeto. Necesita idInternal, idFather como parámetros.
- OBJECTrevision : Crea una revisión de un objeto. Necesita idInternal, idFather, Version, Description, datemake, ExecutedBy, CheckedBy y ApprovedBy como parámetros.
- OBJECTpacking: Crea un paquete. Necesita idInternal como parámetro.
- OBJECTpackingUp: Restaura el paquete indicado con ruta completa Ej: C:\paquete.zip. Necesita Filename como parámetro.
- OBJECTtoXMLatt: Trae el XML de un objeto existente al XML auxiliar. Necesita idInternal, idFather como parámetros.
- OBJECTtoXMLnew: Traer el XML de un objeto nuevo al XML auxiliar. Necesita idClass, idFather como parámetros.
- OBJECTtoXMLcopy: Trae una copia de un objeto existente al XML auxiliar. Necesita idInternal, idFather como parámetros.

- SpaceChange: Cambia el espacio actual. Necesita NewIdSpace como parámetro.
- XMLtoOBJECT: Guarda el XML auxiliar como un objeto. Si existe el objeto lo pisa, sinó lo crea. No necesita parámetros.
- XMLtoXMLadd: Agrega un objeto incluido al XML auxiliar. Necesita idCLASS, idFather como parámetros.
- XMLtoXMLdel: Elimina objeto incluido del XML auxiliar. Necesita idInternal como parámetro.
- XMLtoXMLchgFather: Cambia un padre de un objeto. El cambio lo hace sobre el XML auxiliar. Necesita idOldFather, idNewFather como parámetros.
- XMLtoXMLnewLink: Crea un nuevo Link a un objeto. Lo hace sobre el XML auxiliar. Necesita idNewFather, idnewSpace como parámetros.

Action Type

ActionType es un tipo de dato enumerativo que indica si los datos se insertan o se copian en un archivo dado. Define los siguientes literales:
- Copy copia desde la fila indicada, sobrescribiendo las restantes.
- Insert inserta las filas a partir de donde se le indique, pero desplaza las que hubiera más abajo, en caso de que hubiera.

FileFormat

FileFormat es un tipo de dato enumerativo que indica el formato de salida que tendrá un archivo. Define los siguientes literales:
- HTML
- EXCEL
- PDF
- TIFF
- TEXT
- RTF
- Printer

3.6 Paquete States

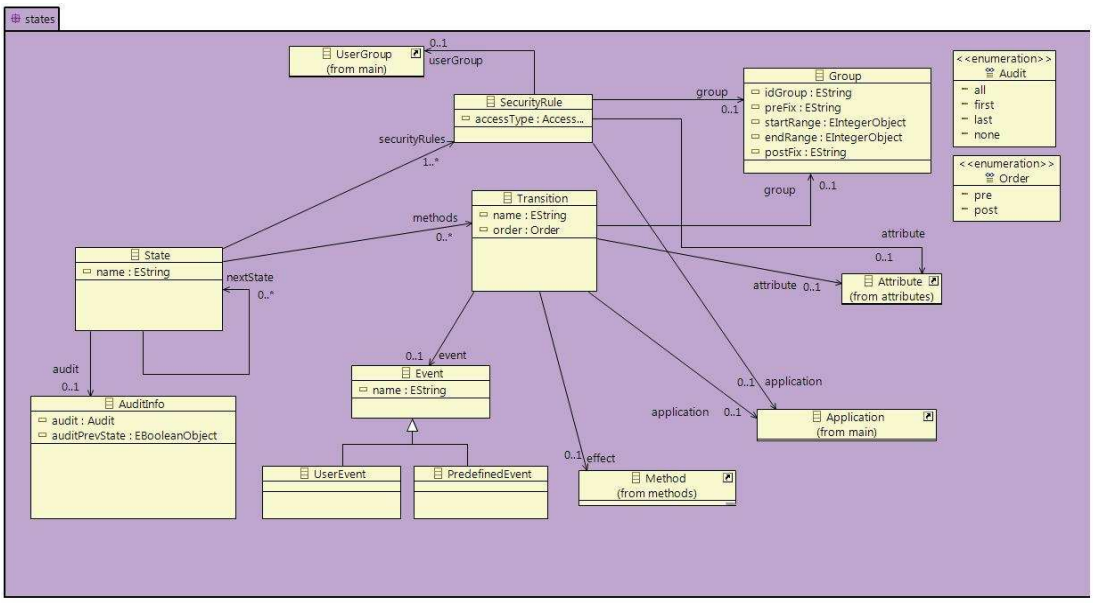

Figura 20 - Paquete States

Estados en FlexAB

FlexAB permite definir un conjunto de estados por los que puede pasar un objeto durante su ciclo de vida. Dicho conjunto de estados y la relación entre ellos conforman un grafo.

- Definición del grafo de Estados: El grafo de estados tiene una estructura visual de árbol, por lo que deben repetirse los nodos para representar un ciclo de grafo. En esta estructura de árbol no se especifica cuando ni como se realizan las transiciones.

Para cada estado es posible definir un conjunto de métodos que se ejecuten en respuesta a un determinado evento solo cuando el objeto se encuentre en un dicho estado.

3.6.1 AuditInfo

Descripción:

Mantiene la información de la auditoria de los estados.

Atributos:

- audit: Audit Indica en que momento se guardará información para la auditoria.
- auditPrevState: Boolean Si el valor es verdadero se audita también el estado previo que tenía el objeto.

3.6.2 Event

Descripción:
Representa el evento por medio del cual se ejecutará el método de la clase. Es una clase abstracta.

Atributos:
- name: String Indica el nombre del evento.

Group
Descripción:
Representa los grupos asociados que tiene la clase. Un grupo puede tener un límite, un rango de inicio y final, por ejemplo al ingresar Start Range igual 1 y End Range igual 10, en este caso solo podríamos agregar diez objetos de la clase por ese grupo. Además este rango es utilizado para manejar la forma de enumerar los objetos que se crean, por ejemplo si estoy dando de alta un objeto chequera, limitar que el número de la chequera sea del 1 al 100, del 20 al 50, etc.

Atributos:
- idGroup: String Indica el identificador del Grupo.
- preFix: String Indica un prefijo para que preceda al nombre. Este prefijo, será parte del Object ID del objeto
- startRange: Integer Indica el número inicial del rango del TAG.
- endRange: Integer Indica el número final del rango del TAG.
- mask: a partir de la opción de configurar grupos también podemos desde un grupo considerar ciertos aspectos del Tag del objeto. Para tal motivo utilizaremos las máscaras. Es decir cuando definimos un grupo le colocamos una máscara, entonces el objeto que pertenezca a ese grupo tendrá un Tag acorde con el formato de la máscara que hemos ingresado.
- postFix: String Indica un posfijo para que proceda al nombre

Asociaciones
- class: Real [1] Indica la clase donde esta definido el grupo.

3.6.3 PredefinedEvent

Descripción:
Representa el evento por medio del cual se ejecutará el método de la clase. Estos

eventos ya se encuentran predefinidos en el sistema.

Atributos:
- type: EventType [1] Indica el tipo de evento

Real (from flexAB.classes)
Descripción:
Asociaciones:
- states: State [*] Representa al conjunto de estados definidos en la clase.
- initialState: State Indica el estado inicial que tendrán los objetos de la clase
- gruops: Group [*] Indica los grupos definidos en la clase.

3.6.4 SecurityRule

Descripción:
Define el tipo de acceso. El tipo de acceso va a depender de los filtros configurados, dependiendo del estado en que se encuentre la clase, de la Clase Padre (Class Father), del atributo (Attribute), del grupo a que pertenece la clase (Group), de acuerdo a la Application definida, y al User Group definido. Por ejemplo podemos definir que para el objeto que pertenece a esta clase, cuando se encuentre en el estado "INITIAL", solo tengan control total sobre el objeto, los Administradores del Sistema.
De acuerdo a lo definido en estos campos de la grilla, va a depender la seguridad de la clase.

Atributos:
- accessType: AccessType [*] Indica el tipo de acceso al objeto que tendrá el grupo de usuarios seleccionado.

Asociaciones:
- classFather: Real [1] Indica la clase a la cual se agrega la regla de seguridad.
- userGroup: UserGroup [0..1] Indica el grupo de usuarios que regula la regla de seguridad.
- attribute: Attribute [1] Indica el atributo del que se quiere regular el acceso.
- application: Application [1] Indica la aplicación a la que pertenece la clase.
- group: Group [1] Indica el grupo al que pertenece la clase.

3.6.4 State

Descripción:
Cuando se crea el objeto, el estado del mismo por defecto es INITIAL.
Las Reglas de Seguridad y los Métodos definidos para el estado INITIAL, se aplican para todos los estados. INITIAL funciona como un estado "comodín". Esto significa que todo lo definido en INITIAL se ejecutará en cada estado existente.
El estado INITIAL no puede ser eliminado.
- Este estado no puede tener hermanos. Cuando se muestran las opciones para elegir el próximo estado del objeto, se muestran hijos del estado actual, por lo que un estado a su nivel, no sería nunca visible.
- Este estado no puede tener una transición a sí mismo. No es posible que uno de los próximos estados del objeto, sea éste.

Atributos:
- name: String Indica el nombre del estado

Asociaciones:
- real: Real Indica la clase a la cual pertenece el estado
- nextStates:State [*] Representa el conjunto con los estados siguientes al estado actual
- securityRules
- events:

Restricciones:
[1] El estado INITIAL no puede tener hermanos.
[2] Este estado no puede tener una transición a sí mismo. No es posible que uno de los próximos estados del objeto, sea éste.

3.6.5 Transition

Atributos:
- name: String Indica el nombre de la transición.
- order: Order Se utiliza para agrupar los métodos antes y después de algún proceso importante del evento.

Asociaciones:
- **attribute: Attribute [0..1]** Si nuestro método tiene el evento
FieldChange, es necesario en este punto asociar el atributo que deberá cambiar para que

se ejecute el método.
- **Application: Application [0..1]** Indica la aplicación para la cual funcionará el método. Mediante este campo determinamos cual será la aplicación en la que se ejecutará el método, también se puede definir el valor 0, lo cual indica que éste método será utilizado por todas las aplicaciones
- **classFather: Real [0..1]** Indica la clase padre.
- **system: System** Indica el sistema para el cual funcionará el método. Mediante este campo determinamos cual será el sistema en el que se ejecutará el método, también se puede definir el valor * (asterisco), lo cual indica que éste método será utilizado en todos los sistemas.
- group: Group [1]

3.6.6 UserEvent

Descripción:
Representa un evento que define el usuario. Dicho evento, será lanzado desde la FlexAB-SUI (Interfaz de Usuario de FlexAB). Seleccionando el objeto y haciendo clic con el botón derecho, se despliega un menú contextual con la lista de eventos de usuarios definidos. Para lanzar el evento, hacer clic sobre el mismo.

3.6.7 Tipos de datos enumerativos del paquete flexAB.states

Audit

Los estados tienen la posibilidad de seleccionar una de las siguientes auditorías. La Auditoría se realiza al momento de GUARDAR, no al momento del FieldChange del Estado.
- ALL: Se realiza la auditoría cada vez que el objeto pasa por el estado. Se crea un registro diferente cada vez en la tabla de auditoría (staNombreClase).
- FIRST: Se realiza la auditoría únicamente la primera vez que se entra al estado.
- LAST: Se realiza la auditoría de la última vez que se entra al estado. Se va actualizando el mismo registro, de manera que quede solamente el dato de la última vez.
- NONE: No se realiza auditoría. Si un estado tenía configurado que realice auditoría, y se modifica para que ya no lo haga, la auditoría hecha no se elimina.

EventType

Descripción:

Representa el evento por medio del cual se ejecutará el método de la clase.

Atributos:

- New: el método se ejecuta cuando se está dando de alta un objeto
- BeforeSave: el método se ejecuta antes de grabar el objeto.
- AfterSave: el método se ejecuta después de grabar el objeto.
- Edit: el método se ejecutará cuando se edita el objeto.
- Delete: el método se ejecutará cuando se elimina el objeto.
- DeleteLink: el método se ejecutará cuando se elimine un link.
- CopyPaste: el método se ejecutará cuando se copia y pega el objeto.
- Cut-Paste: el método se ejecutará cuando se corta y pega el objeto.
- Link-Paste: el método se ejecutará cuando se pega un link.
- Print: el método se ejecutará cuando se envíe a imprimir ó en caso de que el objeto se utiliza en un reporte.
- FieldChange: el método se ejecutará cuando cambia el valor contenido en el campo.
- Revisión: el método se ejecutará cuando se ejecute una revisión.
- Lock: el método se ejecutará cuando se bloquea un objeto.
- UNLock: el método se ejecutará cuando se desbloquea un objeto.
- Packing: el método se ejecutará cuando se empaqueta un objeto.
- PackingRestore: el método se ejecutará cuando se desempaqueta un objeto.
- BeforeOpenFile: el método se ejecutará antes de abrir un archivo.
- AfterOpenFile: el método se ejecutará después de abrir un archivo.
- StateChange: el método se ejecutará al cambiar un estado.

3.7 Paquete DataType

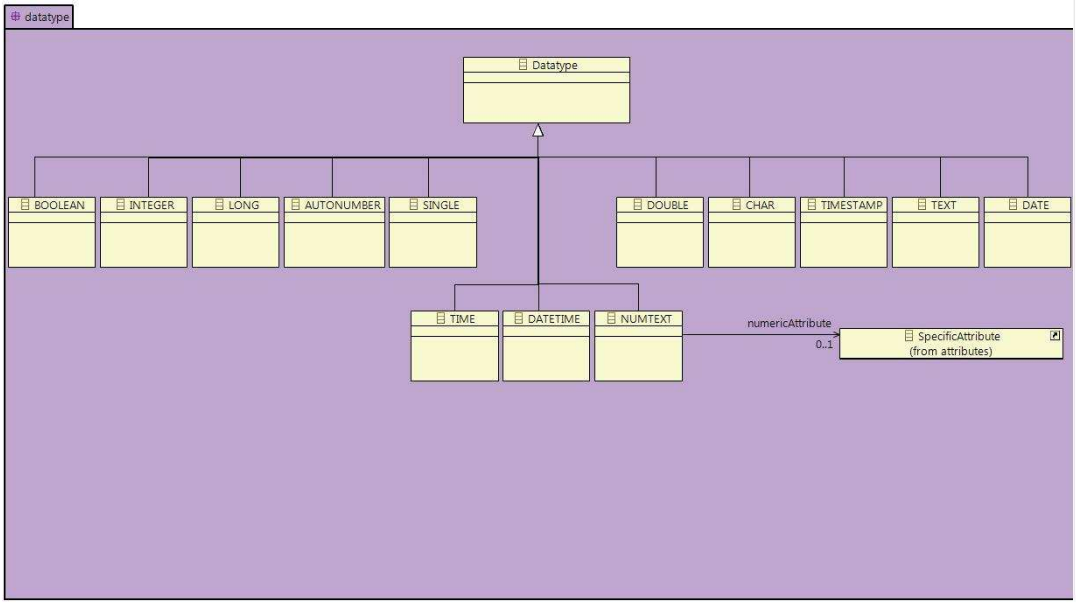

Figura 21 - Paquete Data Type

3.7.1 NumText

Descripción:

Un atributo de este tipo, tiene la característica de traducir un número a su escritura en texto.

Asociaciones:

- numericAttribute: SpecificAttribute Indica un atributo de tipo numérico que tiene definidos la clase. Debemos seleccionar el atributo que deseamos sea traducido a letras.

4 Transformaciones de modelos

4.1 ¿Qué es una transformación?

El proceso MDD, descrito anteriormente, muestra el rol de varios modelos, PIM, PSM y código dentro del framework MDD. Una herramienta que soporte MDD, toma un PIM como entrada y lo transforma en un PSM. La misma herramienta u otra, tomará el PSM y lo transformará a código. Estas transformaciones son esenciales en el proceso de desarrollo de MDD. En la figura siguiente se muestra la herramienta de transformación como una caja negra, que toma un modelo de entrada y produce otro modelo como salida.

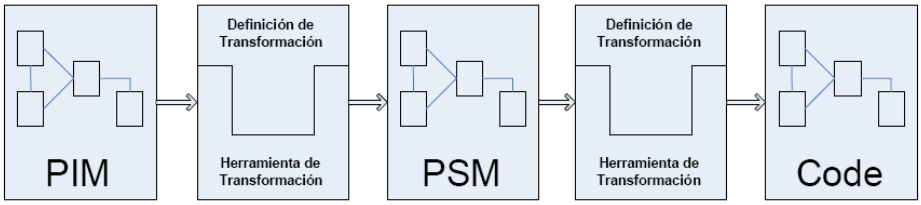

Figura 22 - PIM a PSM

Si abriéramos la herramienta de transformación y mirásemos dentro, podríamos ver qué elementos están involucrados en la ejecución de la transformación. En algún lugar dentro de la herramienta hay una definición que describe como se debe transformar el modelo fuente para producir el modelo destino. Esta es la definición de la transformación. La figura 2-6 muestra la estructura de la herramienta de transformación. Notemos que hay una diferencia entre la transformación misma, que es el proceso de generar un nuevo modelo a partir de otro modelo, y la definición de la transformación.

Para especificar la transformación, (que será aplicada muchas veces, independientemente del modelo fuente al que será aplicada) se relacionan construcciones de un lenguaje fuente en construcciones de un lenguaje destino. Se podría, por ejemplo, definir una transformación que relaciona elementos de UML a elementos Java, la cual describiría como los elementos Java pueden ser generados a partir de los elementos UML. Esta situación se muestra en la figura siguiente.

En general, se puede decir que una definición de transformación consiste en una colección de reglas, las cuales son especificaciones no ambiguas de las formas en que un modelo (o parte de él) puede ser usado para crear otro modelo (o parte de él).

Figura 23 - Uml a Java

Las siguientes definiciones fueron extraídas del libro de Anneke Kepple [KWB 03]:

- *Una transformación es la generación automática de un modelo destino desde un modelo fuente, de acuerdo a una definición de transformación.*

- *Una definición de transformación es un conjunto de reglas de transformación que juntas describen como un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino.*

- *Una regla de transformación es una descripción de como una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.*

4.2 ¿Cómo se define una transformación?

Una transformación entre modelos puede verse como un programa de computadora que toma un modelo como entrada y produce un modelo como salida. Por lo tanto las transformaciones podrían describirse (es decir, implementarse) utilizando cualquier lenguaje de programación, por ejemplo Java. Sin embargo, para simplificar la tarea de codificar transformaciones se han desarrollado lenguajes de más alto nivel (o específicos del dominio de las transformaciones) para tal fin, tales como ATL [ATL] y QVT [QVT].

4.3 Un ejemplo de transformación

Exhibiremos un ejemplo de una transformación de un modelo PIM escrito en UML a un modelo de implementación escrito en Java. Transformaremos el diagrama de clases de un sistema de venta de libros (Bookstore) en las clases de Java correspondientes a ese modelo. La figura siguiente muestra gráficamente la transformación que se intenta realizar, la cual consta de 2 pasos: Transformaremos el PIM en un PSM y luego

transformaremos el PSM resultante a código Java.

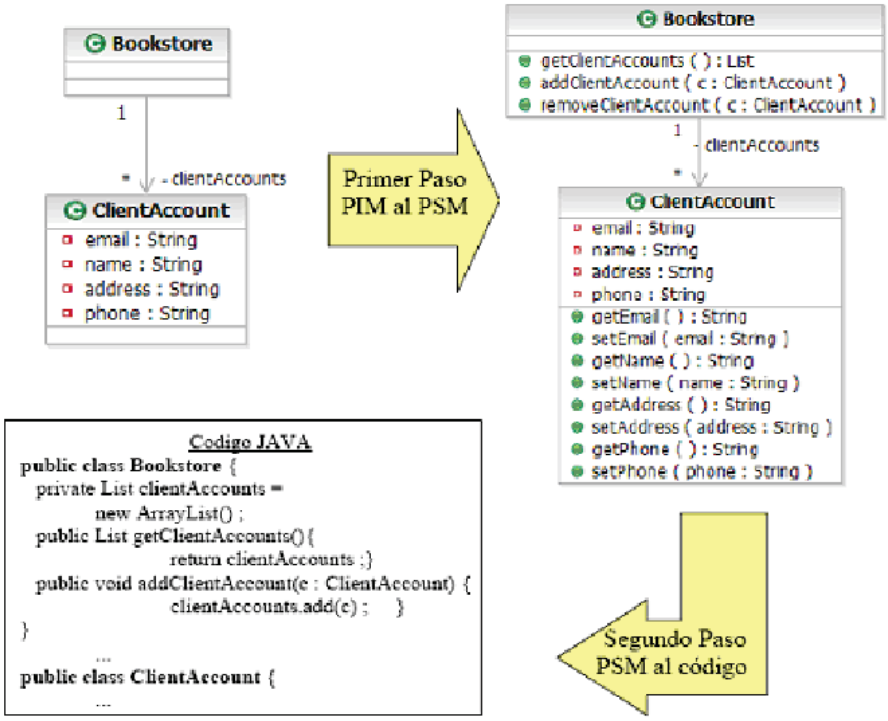

Figura 24 - Ejemplo de transformación

Tanto el PIM como el PSM son modelos útiles ya que proveen el nivel de información adecuado para diferentes tipos de desarrolladores y otras personas involucradas en el proceso de desarrollo de software. Existe una clara relación entre estos modelos. La definición de la transformación que debe aplicarse para obtener el PSM a partir del PIM consiste en un conjunto de reglas. Estas reglas son:

- Para cada clase en el PIM se genera una clase con el mismo nombre en el PSM.
- Para cada relación de composición entre una clase llamada classA y otra clase llamada classB, con multiplicidad 1 a n, se genera un atributo en la clase classA con nombre classB de tipo Collection.
- Para cada atributo público definido como attributeName:Type en el PIM los siguientes atributos y operaciones se generan como parte de la clase destino :
    - Un atributo privado con el mismo nombre: attributeName : Type
    - Una operación pública cuyo nombre consta del nombre del atributo precedido con "get" y el tipo del atributo como tipo de retorno: getAttributeName() : Type
    - Una operación pública cuyo nombre consta del nombre del atributo precedido con "set" y con el atributo como parámetro y sin valor de retorno: setAttributeName(att : Type).

El siguiente paso consistirá en escribir una transformación que tome como entrada el PSM y lo transforme a código Java. Combinando y automatizando ambas transformaciones podremos generar código Java a partir del PIM.

5 Descripción de tecnologías

5.1 Eclipse

5.1.1 Plataforma

Eclipse es una plataforma de software de código abierto independiente de una plataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos integrados de desarrollo (del inglés IDE).
La plataforma eclipse, está construida sobre un mecanismo para descubrir, integrar y correr módulos llamados plugins. Un proveedor de herramientas escribe una herramienta como un plugin separado, que opera sobre archivos en el workspace y su interfaz de usuario específica trabaja sobre la superficie del workbench. Cuando la plataforma es cargada, el usuario es presentado con un ambiente de desarrollo integrado (IDE) compuesto de un conjunto de plugins habilitados.
La Plataforma Eclipse está diseñada y construida para cumplir con los siguientes requerimientos:

- Soportar la construcción de una variedad de herramientas para el desarrollo de aplicaciones.
- Soportar un irrestricto conjunto de proveedores de herramientas, incluyendo vendedores de software independientes.
- Soportar herramientas para manipular tipos de contenido arbitrario (por ej. HTML, Java, C, JSP, EJB, XML, UML, GIF, ETC).
- Permitir una fácil integración de las herramientas entre sí, a través de los diferentes tipos de contenidos y sus proveedores.
- Soportar el desarrollo de aplicaciones basadas y no, en interfaz gráfica de usuario (en inglés Graphical User Interface, GUI y non-GUI-based).
- Correr en una gran cantidad de sistemas operativos.

El principal objetivo es el de proveer facilidades a los proveedores de herramientas para desarrollar las mismas con mecanismos de uso y reglas para seguir. Estos mecanismos son provistos por una API (Application Programming Interface - Interfaz de Programación de Aplicaciones) de interfaces bien definida, clases y métodos.

5.1.2 La Arquitectura Eclipse

Eclipse está formado por el núcleo, el entorno de trabajo (Workspace), el área de desarrollo (Workbench), la ayuda al equipo (Team support) y la ayuda o documentación (Help).

- Núcleo: su tarea es determinar cuáles son los plugins disponibles en el directorio de plugins de Eclipse. Cada plugin tiene un fichero XML manifest que lista los elementos que necesita de otros plugins así como los puntos de extensión que ofrece. Como la cantidad de plugins puede ser muy grande, sólo se cargan los necesarios en el momento de ser utilizados con el objeto de minimizar el tiempo de arranque de Eclipse y recursos.
- Entorno de trabajo: maneja los recursos del usuario, organizados en uno o más proyectos.
  Cada proyecto corresponde a un directorio en el directorio de trabajo de Eclipse, y contienen archivos y carpetas.
- Interfaz de usuario: muestra los menús y herramientas, y se organiza en perspectivas que configuran los editores de código y las vistas. A diferencia de muchas aplicaciones escritas en Java, Eclipse tiene el aspecto y se comporta como una aplicación nativa. No está programada en Swing, sino en SWT (Standard Widget Toolkit) y Jface (juego de herramientas construida sobre SWT), que emula los gráficos nativos de cada sistema operativo. Este ha sido un aspecto discutido sobre Eclipse, porque SWT debe ser portada a cada sistema operativo para interactuar con el sistema gráfico. En los proyectos de Java puede usarse AWT y Swing salvo cuando se desarrolle un plugin para Eclipse.
- Ayuda al grupo: este plugin facilita el uso de un sistema de control de versiones para manejar los recursos en un proyecto del usuario y define el proceso necesario para guardar y recuperar de un repositorio. Eclipse incluye un cliente para CVS.
- Documentación: al igual que el propio Eclipse, el componente de ayuda es un sistema de documentación extensible. Los proveedores de herramientas pueden añadir documentación en formato HTML y, usando XML, definir una estructura de navegación.

5.2 EMF (Eclipse Modeling Framework)

El proyecto EMF es un Framework de modelado y generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurados. A partir de una especificación de modelo descripta en XMI, EMF provee herramientas y soporte

Runtime para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y edición vía comandos del modelo, y un editor básico.

Los modelos pueden ser especificados usando anotación Java, documentos XML, o herramientas de modelado como Rational Rose, y después ser exportados a EMF. Lo más importante de todo, EMF suministra las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.

El EMF incluye XML Schema Infoset Model (XSD), un nuevo componente del proyecto Model

Development Tools (MDT), y una implementación basada en EMF de Service Data Objects (SDO). XSD provee un modelo y una API para manipular componentes de un esquema XML, con acceso a representación DOM del documento del esquema.

EMF consiste de tres piezas fundamentales:

- **EMF-Core**: El core del Framework EMF incluye un meta modelo (ECore) para describir modelos y soporte runtime para los modelos incluyendo notificaciones de cambio, soporte de persistencia con serialización XMI (XML Metadata Interchange) por defecto, y una muy eficiente API para manipular objetos EMF genéricamente.
- **EMF-Edit**: El Framework EMF.Edit incluye clases reutilizables genéricas para construir editores para modelos EMF. Esto provee:

  Clases proveedoras de contenido y etiquetas, y otras clases que permiten a los modelos EMF ser mostrados usando visualizadores de escritorio estándar.

  Un Framework de comandos, incluyendo un conjunto de clases de implementación de comandos genéricos para editores de construcción que soporten completamente el “rehacer” y “deshacer” automáticos.
- **EMF-Codegen**: La generación de código EMF es capaz de generar todo lo necesitario para construir un editor completo para un modelo EMF. Esto incluye un GUI desde el cual pueden ser especificadas las opciones de generación, y los generadores a ser invocados. La generación se basa en JDT (Java Development Tooling), un componente de Eclipse. Hay tres niveles de generación de código que son soportadas:

  Modelo: Proporciona clases Java de implementación e interfaz para todas las clases del modelo, además una clase de implementación de factory y package (meta data).

  Adaptadores: Genera clases de implementación (llamadas ItemsProviders) que adaptan las clases del modelo para ser editadas y mostradas.

  Editor: Produce un editor estructurado adecuadamente que se ajusta al

estilo recomendado por los editores de modelos EMF Eclipse y sirve como punto de partida al comienzo de la personalización.

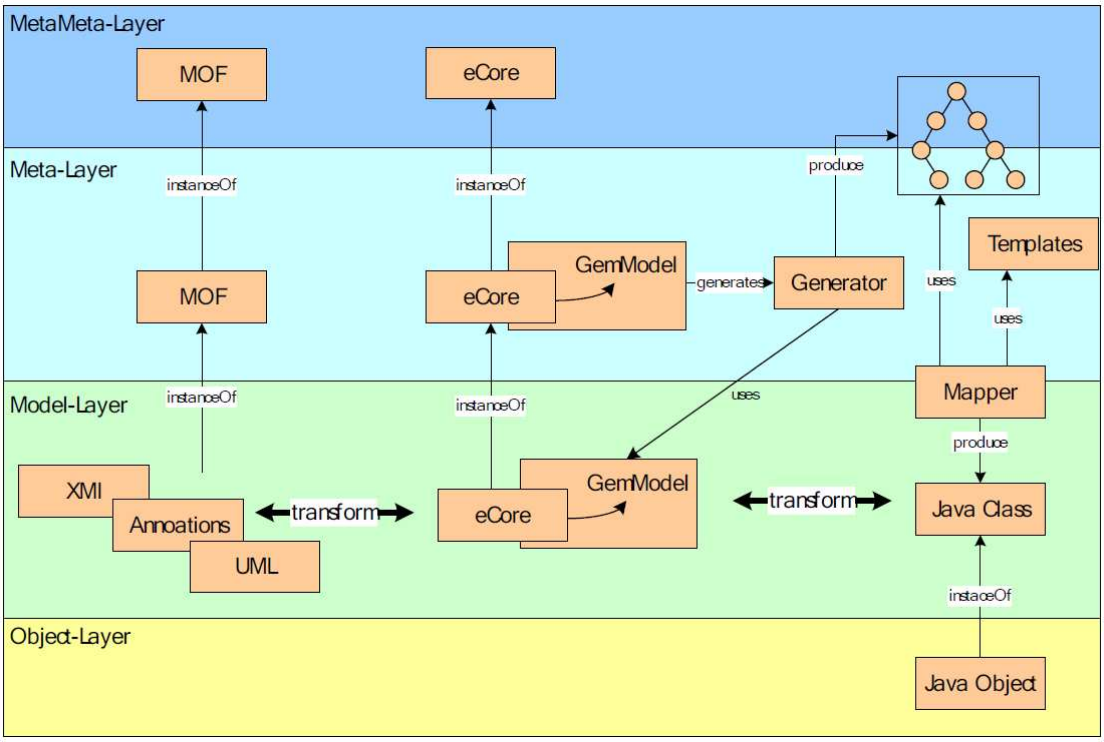

Figura 25 - EMF

5.3 Transformaciones ATL

ATL (Atlas Transformation Language) [ATL] consiste en un lenguaje de transformación de modelos y un *toolkit* creados por ATLAS Group (INRIA y LINA) que forma parte del proyecto GMT de Eclipse. En él se presenta un lenguaje híbrido de transformaciones declarativas y operacionales de modelos que si bien es del estilo de QVT no se ajusta a las especificaciones. Aunque la sintaxis de ATL es muy similar a la de QVT, no es interoperable con este último. Tiene herramientas que realizan una compilación del código fuente a *bytecodes* para una máquina virtual que implementa comandos específicos para la ejecución de transformaciones. ATL forma parte del framework de gestión de modelos AMMA que se encuentra integrado en Eclipse y EMF. ATL posee un algoritmo de ejecución preciso y determinista. El toolkit es de código abierto.

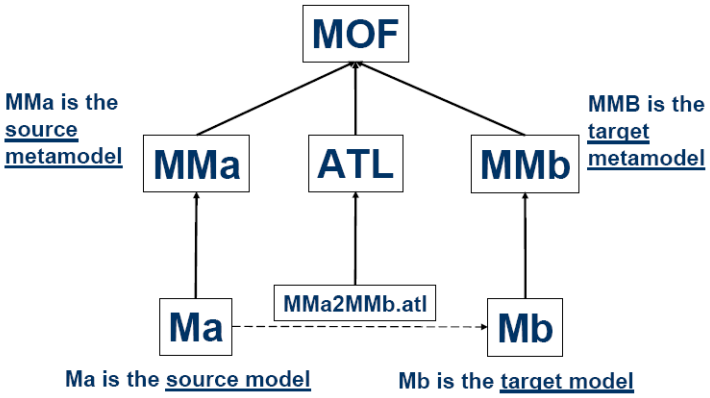

Figura 26 - ATL

6 Diseño de la solución

En este capítulo describiremos el diseño de una solución que nos permitirá hacer dos tipos de transformaciones. En primer lugar se explica conceptualmente la solución propuesta, por último se explica la arquitectura elegida para dar soporte a la solución propuesta.

6.1 Solución propuesta

Para poder realizar un buen trabajo y poder remarcar las ventajas de la solución hemos restringido las clases del metamodelo FlexAB que participarán en las transformaciones. El motivo para esta decisión es que no permite hacer hincapié en las clases a transformar que representa una ventaja al estar representadas en el metamodelo UML.
Otro punto importante es que se decidió implementar un metamodelo UML simplificado, esta decisión fue tomada para tener una mejor apreciación del modelo de UML. Esto último no quiere decir que se haya definido un metamodelo de UML nuevo, sino que en base al metamodelo original de UML se desarrolló un metamodelo con una cantidad de clases más acotadas y necesarias para la solución planteada.
La solución propuesta se divide en dos etapas.

6.1.1 Transformación de FlexAB a UML

En esta transformación se tomará como entrada el modelo de FlexAB y se obtendrá como salida el modelo de UML. La entrada de la transformación no es el modelo FlexAB propiamente dicho sino un conjunto de instancias de las clases pertenecientes a dicho modelo y se obtendrá como salida las instancias correspondientes, según la transformación, pertenecientes a las clases del modelo UML.

6.1.2 Transformación de UML a FlexAB

Esta transformación representa lo contrario a la transformación anteriormente explicada, es decir que como entrada tendrá un conjunto de instancias pertenecientes al modelo UML y como salida se obtendrá las instancias correspondientes del modelo FlexAB.

6.2 Arquitectura propuesta para la solución

La arquitectura no es un punto menor en el desarrollo, y es por eso que a continuación se hace una descripción de la arquitectura planteada para dar soporte de los problemas

planteados.

La arquitectura consta de dos proyectos. Un proyecto java que contendrá todo lo relacionados a los modelos, diagramas, instancias de los modelos y código generado a partir de los modelos. Por otro lado habrá un proyecto ATL que contendrá las transformaciones detalladas anteriormente.

El proyecto ATL consumirá datos del proyecto java y los datos producidos por las transformaciones serán volcados en el proyecto java.

La jdk (Java Development Kit) utilizada para está arquitectura es la versión 1.5.

7 Implementación de la solución

En este capítulo se detalla la implementación concreta de los módulos que componen la arquitectura de la solución planteada. En una primera parte se explicará la capa que utilizada para la creación de modelos y en una segunda parte se explicará todo lo referido a la transformaciones. La ultima sección de este capitulo explica la correspondencia entre las clases FlexAB y las clases de UML.

7.1 Implementación de capa contenedora de modelos a transformar

Lo primero que debemos hacer es crear un proyecto, para esto desde eclipse vamos a la opción file/ New y seleccionamos la opción Java Project como se indica a continuación.

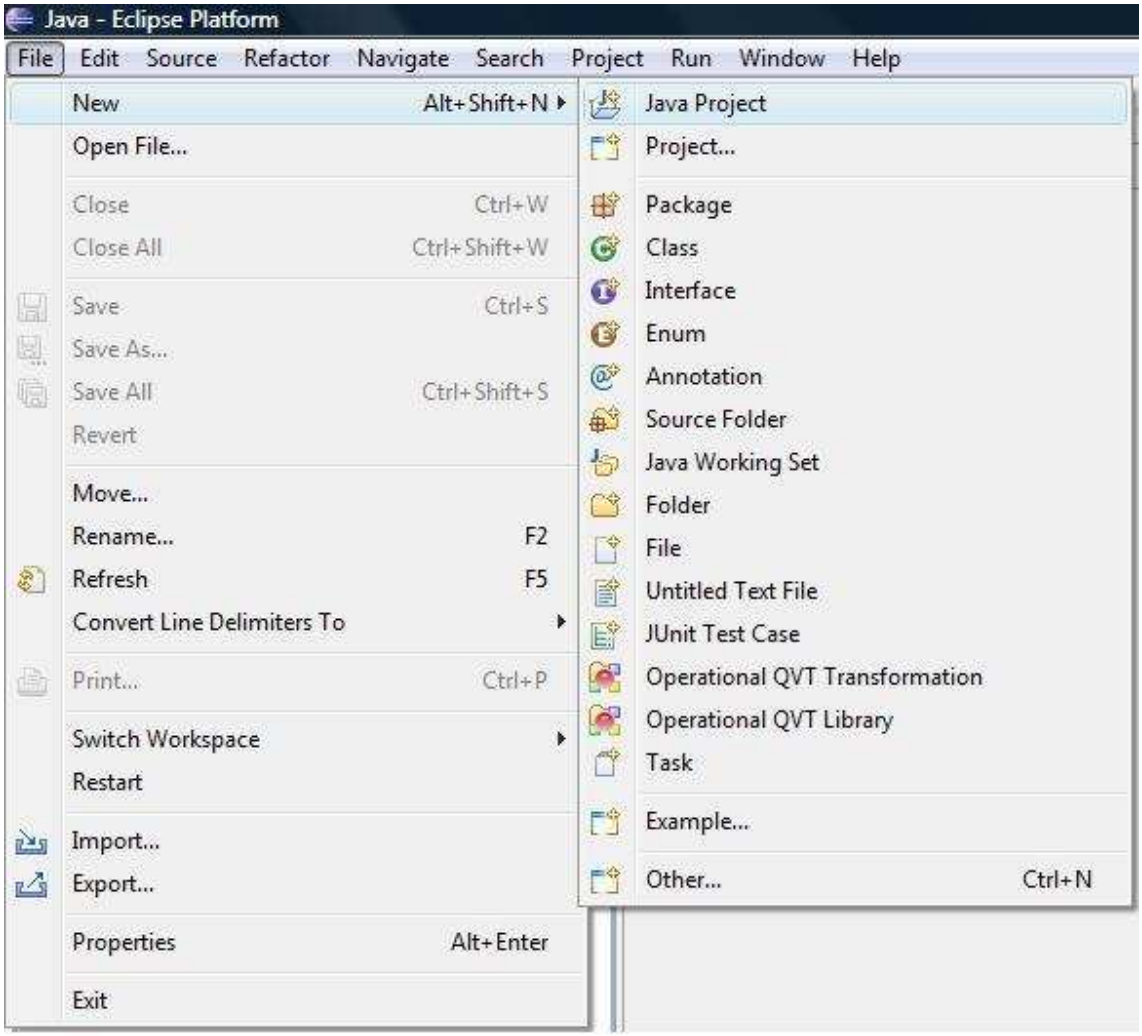

Figura 27 - Proyecto Java

A continuación debemos completar los datos como se muestran en la siguiente figura y presionar por último el botón *finish.*

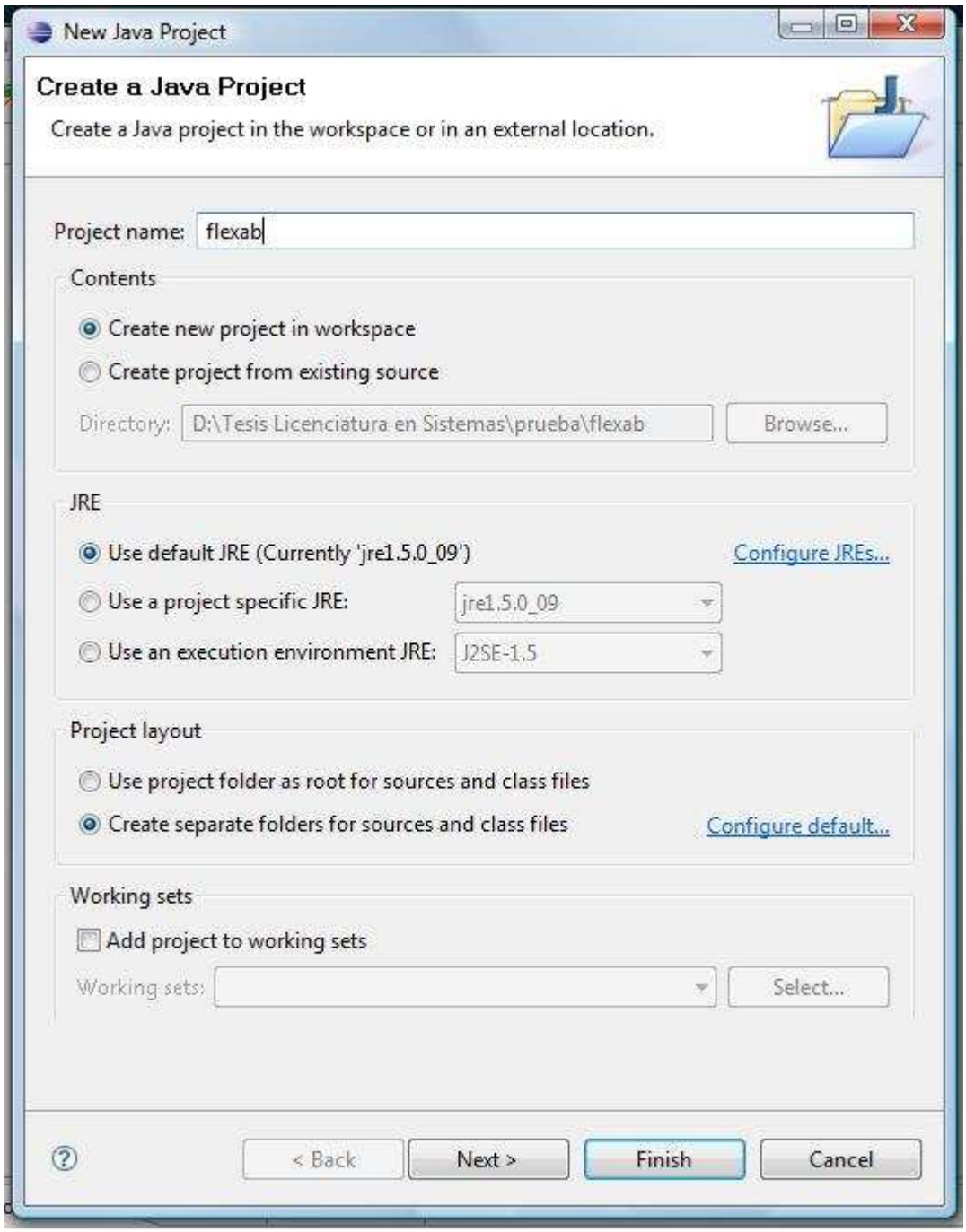

Figura 28 - Crear Proyecto Java

Con los dos pasos anteriores tendremos creado el proyecto java, a continuación se muestra la estructura que tendrá dicho proyecto y el contenido de cada una de las carpetas.

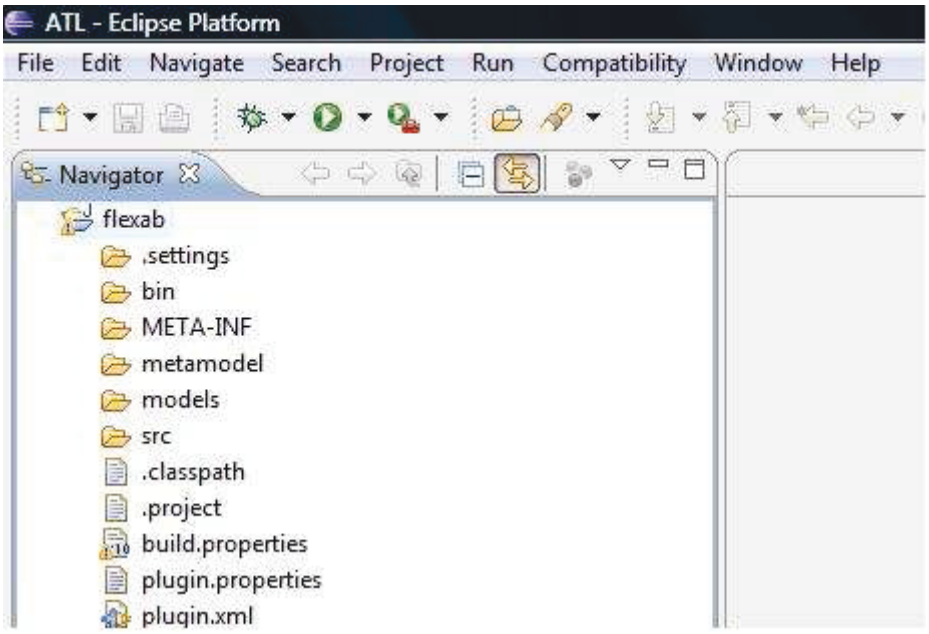

Figura 29 - Estructura Proyecto Java

La anterior nos muestra la estructura del proyecto:

- La carpeta metamodel contendrá los metamodelos que participarán en la transformación.
- La carpeta models contendrá los archivos de entradas y salidas utilizados en las transformaciones.
- La carpeta src contiene el código fuente generado a través de los modelos contenidos en metamodel.
- El resto de las carpetas son creadas a partir de la creación del proyecto flexab.

Ahora nos queda crear el metamodelo de FlexAB, la figura a continuación nos muestra la forma de hacer esto.

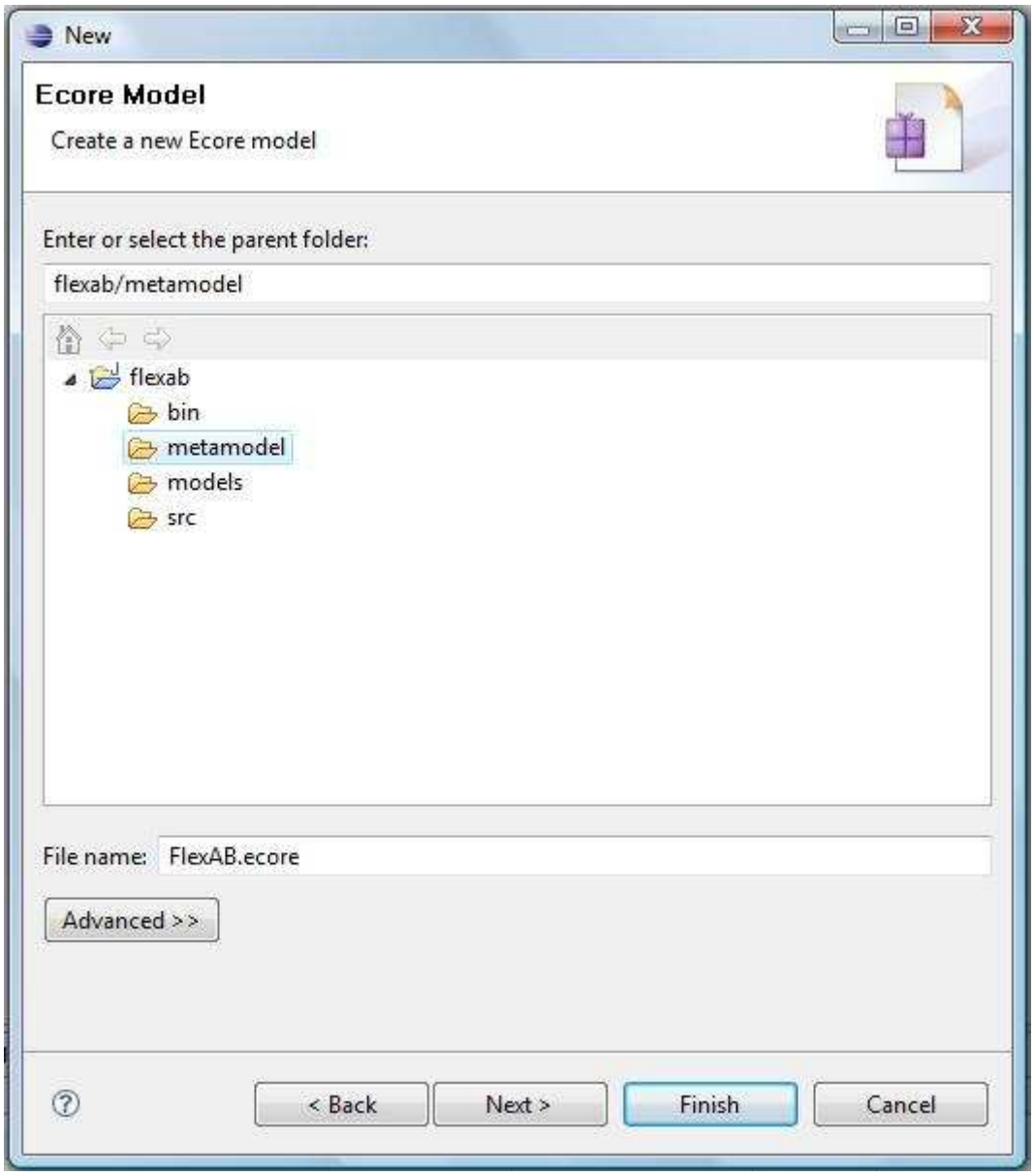

Figura 30 - Carpeta metamodel

Una vez aceptado el cuadro de diálogo anterior nos quedará creado el archivo FlexAB.ecore. A partir de este archivo se crearán las clases.

Para completar las clases del metamodelo FlexAB debemos pararnos sobre este archivo, presionamos botón derecho y nos aparecerá la opción como se muestra en la siguiente figura.

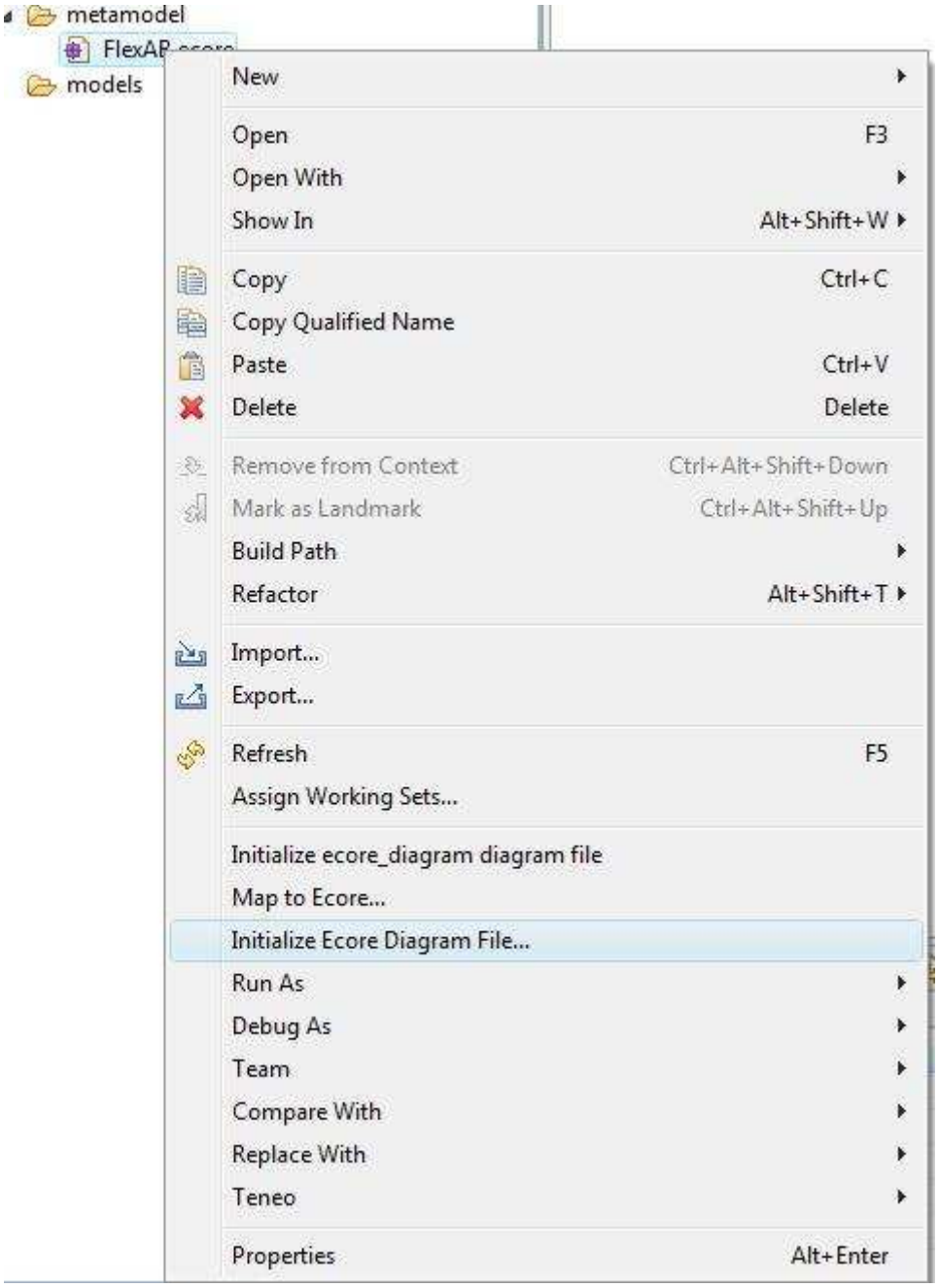

Figura 31 - Crear FlexAB.ecore

Al seleccionar la opción anterior nos abrirá un cuadro de diálogo en el que debemos completar algunos datos como se muestra a continuación.

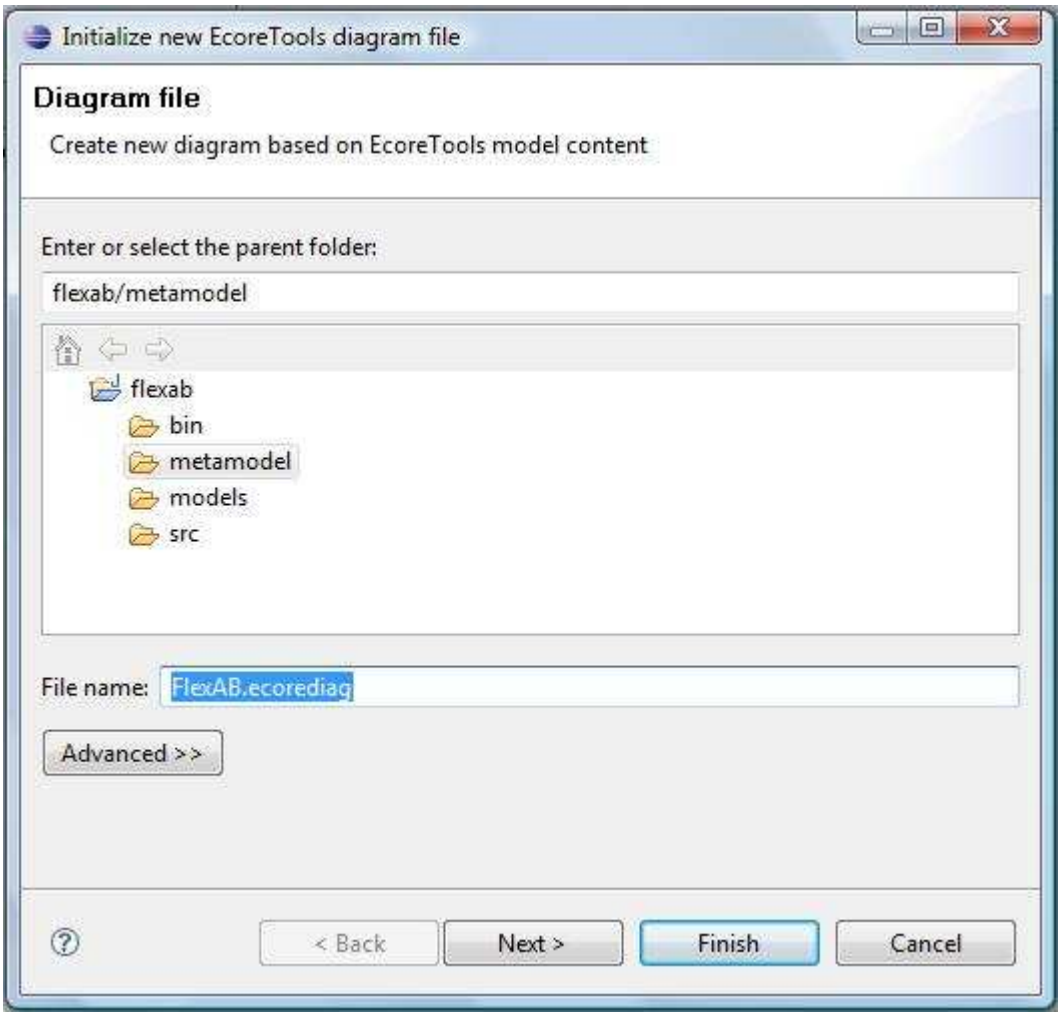

Figura 32 - Crear diagrama de FlexAB

A continuación se abrirá una herramienta perteneciente a EMF que nos permitirá completar nuestros modelos. La figura a continuación nos muestra dicha herramienta.

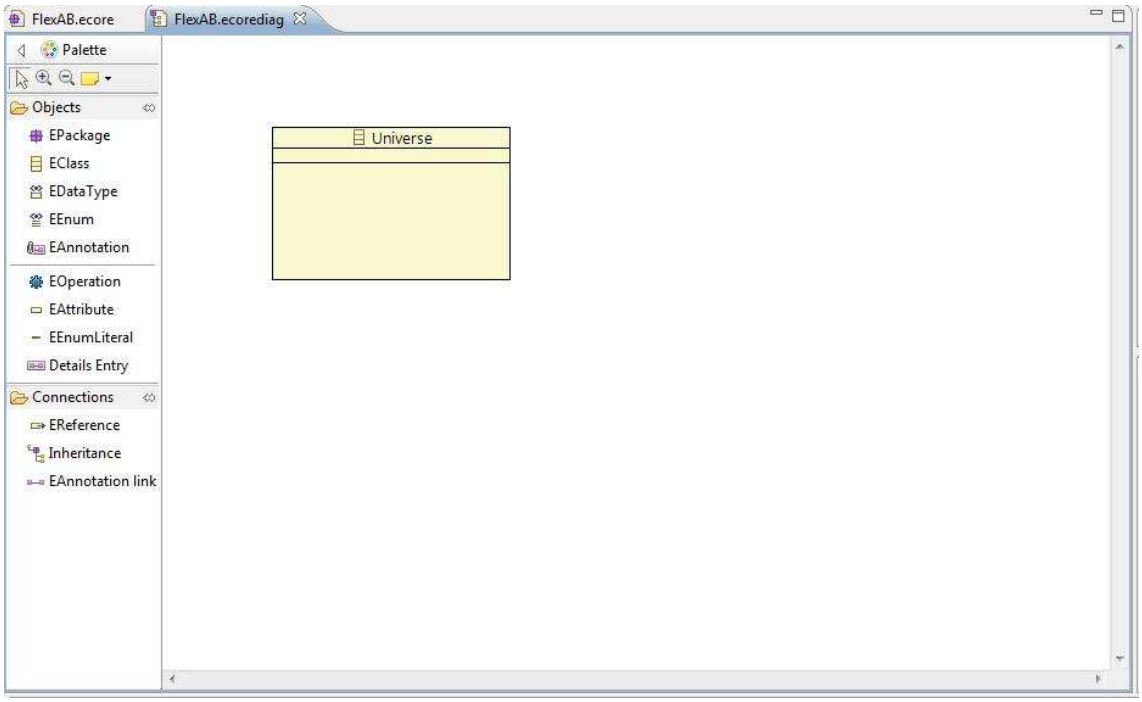

Figura 33 - Herramienta para instanciar modelo

Esta herramienta será utilizada para completar el resto de las clases del metamodelo de FlexAB. La herramienta presenta una paleta como se muestra a continuación.

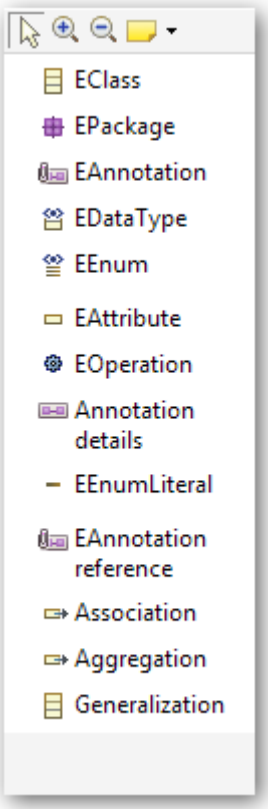

Figura 34 - Paleta de elementos

La paleta de herramientas incluye los diferentes tipos de elementos y relaciones que permiten definir un modelo Ecore. En la paleta se puede distinguir la herramienta *Eclass*, con la que podremos crear nuevas clases, la herramienta *Epackage*, con la que crearemos nuevos paquetes, *EAnnotation*, con la que se crearán anotaciones y comentarios, *EDataType* con la que crear nuevos tipos de datos, *EAttribute* con la que agregar atributos a las clases, *EOperation* con la que añadir operaciones a las clases y por último, las relaciones de asociación (*Association*), agregación (*Aggregation*) y herencia (*Generalization*). El modelado se realiza mediante "drag and drop" para cualquiera de las herramientas de la paleta, esto quiere decir que si pulsamos por ejemplo sobre la herramienta de creación de clases y acto seguido pulsamos sobre el tapiz de modelado se nos crea una nueva clase. El nombre de las clases en Eclipse debe comenzar siempre por letra mayúscula. Por otro lado, es posible añadirles tantos atributos y operaciones como se deseen. Hay dos posibilidades de hacerlo: una de ellas consiste en arrastrar desde la paleta de modelado y soltar sobre la clase los atributos y operaciones, y la otra es a través de un menú emergente que aparece superponiendo el puntero sobre la clase en cuestión.

Las relaciones entre clases se instancian gráficamente mediante las 3 últimas herramientas que aparecen en la paleta de modelado. Para poder relacionar dos clases se debe seleccionar la herramienta y acto seguido seleccionar la clase de la que parte la relación (en el caso de herencia, la clase hija, y en el caso de la agregación, la clase compuesta) y después arrastrar y seleccionar la clase destino de la relación (en el caso de herencia, la clase padre, y en el caso de la agregación, la clase componente). Se ha de tener en cuenta, que las asociaciones en EMF son siempre unidireccionales. Si se desea modelar una relación bidireccional, se realiza mediante dos asociaciones que relacionen las dos mismas clases en sentidos opuestos y fusionarlas en una sola bidireccional. Para ello, se ha de seleccionar dentro de una de las asociaciones, su opuesta en la propiedad *Eopposite*. Para definir el tipo de un atributo, clase o relación es necesario editar sus propiedades. Esto se realiza pulsando con el botón derecho encima del elemento del que se desean ver las propiedades y seleccionar la opción *Show Properties View*. A continuación, aparece una ventana en la parte inferior del entorno de Eclipse mediante la que se pueden editar todas las propiedades de cada elemento. Entre las propiedades que definen a una clase están el nombre, si pertenecen a alguna subclase, si son de algún tipo predefinido, etc. Para editar el tipo de los atributos, se ha de pinchar en la propiedad *EType* y seleccionar el tipo de datos adecuado para cada atributo, de entre todos los que se muestran en la lista, *string* para las cadenas de caracteres, *int* para los números enteros, *float* para los números reales, *char* para los caracteres y *boolean* para los booleanos, entre otros. También, podemos crear enumeraciones de elementos, es decir, listas de tipos de datos. Esta definición se realiza sobre el tapiz de modelado en el que

definimos nuestro modelo Ecore. Para ello añadimos un objeto de tipo *EEnum* de la barra de herramientas de modelado al tapiz mediante "drag and drop". Posteriormente, le damos un nombre en el campo *Name*. Este será el nombre que tendrá nuestro tipo de datos enumerado. Para crear la lista sólo nos queda definir dentro del elemento *EEnum* tantos *EEnumLiteral* como número de tipos de datos queramos introducir. *EEnumLiteral* se encuentra también en la barra de herramientas de modelado. A cada *EEnumLiteral* se le ha de asociar su nombre, que será el nombre con el que aparecerá en la enumeración. Para ello, introducimos el mismo nombre para el campo *Name* y *Literal* del *EEnumLiteral*.
Dado que EMF expresa en formato XML el modelo correspondiente a un diagrama ecore, conviene resaltar que para definir un modelo correctamente en EMF es necesario incluir un elemento raíz (una clase) que relacione mediante agregación todas las clases del modelo, en nuestro caso de estudio la clase Universe. Al crear una relación se le asigna un nombre y se especifica su cardinalidad. Para esto último, existen dos propiedades dentro de la vista de propiedades, *Lower bound* en la cual se especifica la cardinalidad mínima, y *Upper bound*, en la que se especifica la cardinalidad máxima. Cabe destacar que para conseguir la cardinalidad N, se debe poner el valor -1 y automáticamente, al dejar de editar la relación, se cambia el valor a *.

Las clases de FlexAB han sido pasadas en distintos archivos .ecorediag. A continuación se muestra la estructura de estos archivos.

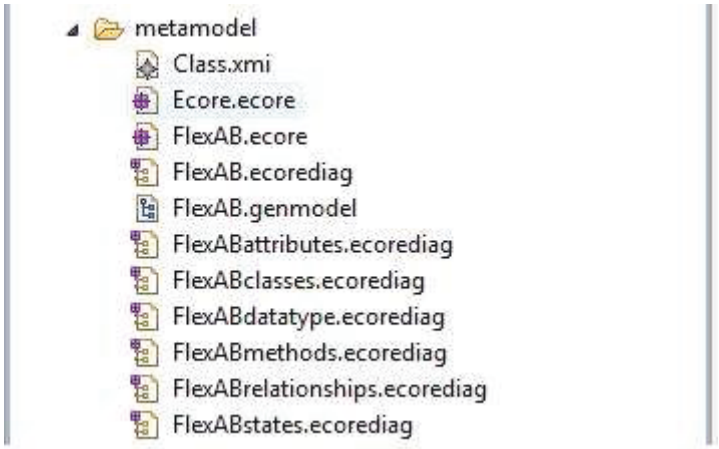

Figura 35 – Archivos ecorediag

Por último debemos crear el archivo FlexAB.genmodel, para lograr esto nos paramos sobre el archivo FlexAB.ecore seleccionamos *New EMF Model*.

- Seleccionamos *Ecore model* como modelo de entrada y pulsamos *Next*.

- Pulsamos *Browse Workspace*, si el modelo Ecore se encuentra en nuestro espacio de trabajo en un proyecto abierto, o *Browse File System* si el modelo está fuera del espacio de trabajo, seleccionamos FlexAB.ecore como modelo de entrada dentro de la carpeta metamodel, y pulsamos *Next*.
- Pulsamos *Finish*.

Si desplegamos el explorador de paquetes, debemos tener una vista similar a la de la siguiente figura.

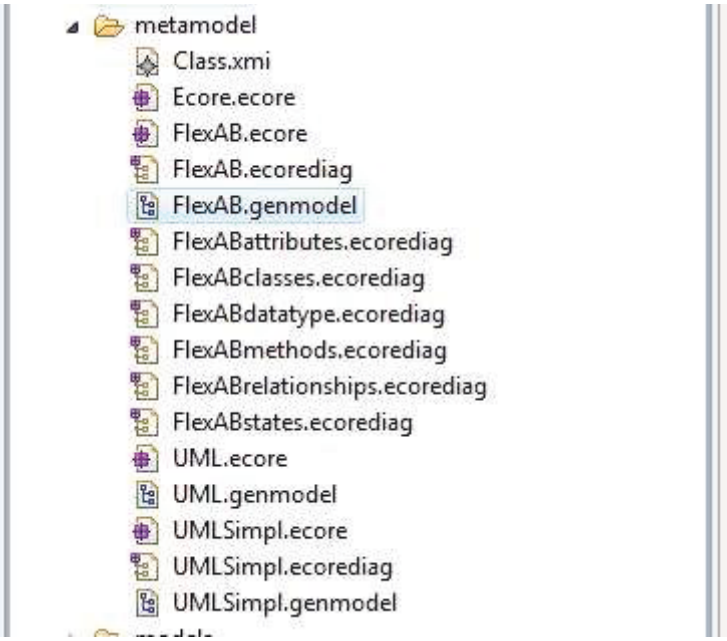

Figura 36 - Contenido de carpeta metamodel

De manera similar a lo que hemos realizado para generar el modelo de FlexAB, debemos hacer lo mismo para generar el modelo de UML. Recordemos que se definió utilizar un modelo de UML simplificado. Como muestra la figura anterior nos deben quedar los modelos de FlexAB y UMLSimpl (UML simplificado) con sus respectivos archivos.

7.2 Implementación de capa para realizar las transformaciones

En esta sección explicaremos la parte necesaria para realizar las transformaciones.
Como primer paso debemos crear un proyecto ATL como se indica en la figura a continuación.

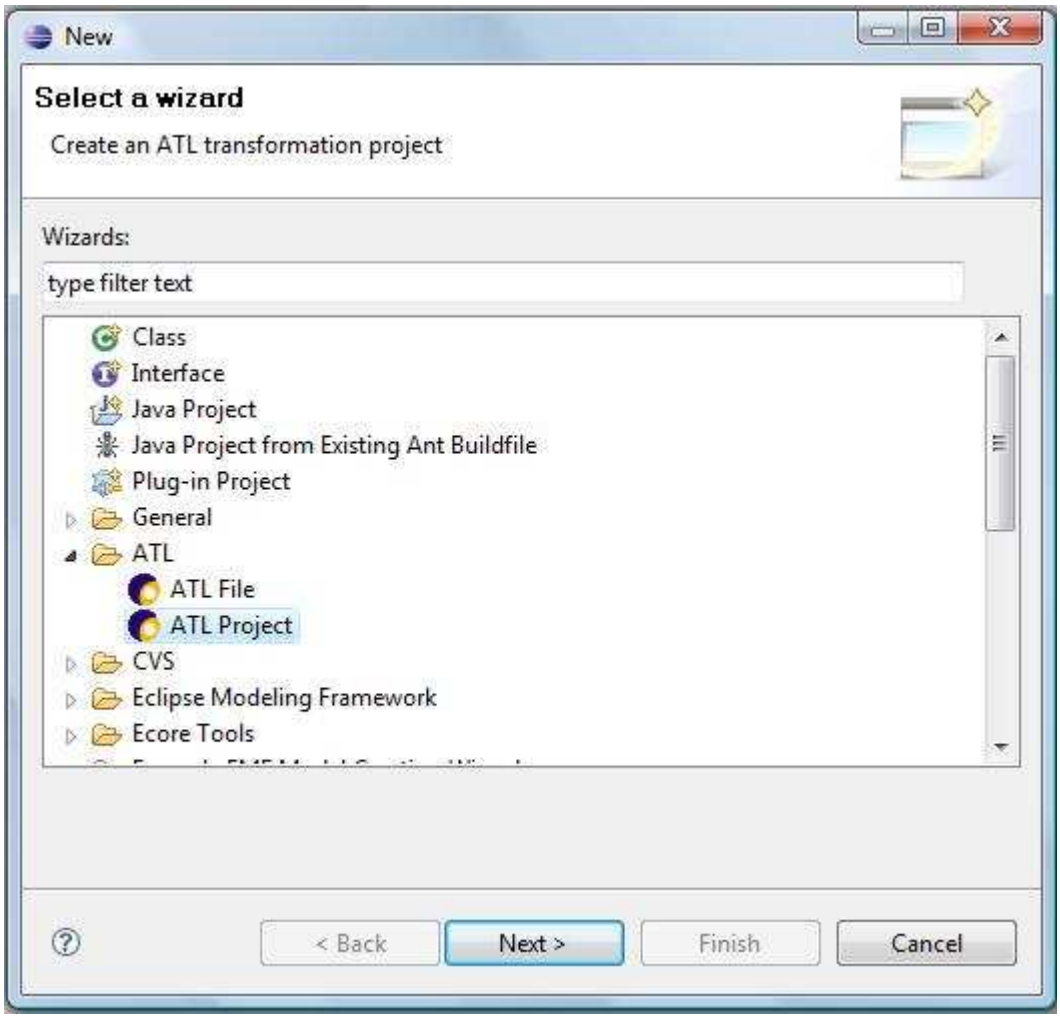

Figura 37 - Crear Proyecto ATL

Debemos seleccionar la opción ATL Project y al presionar next se abrirá un diálogo como el que se muestra a continuación.

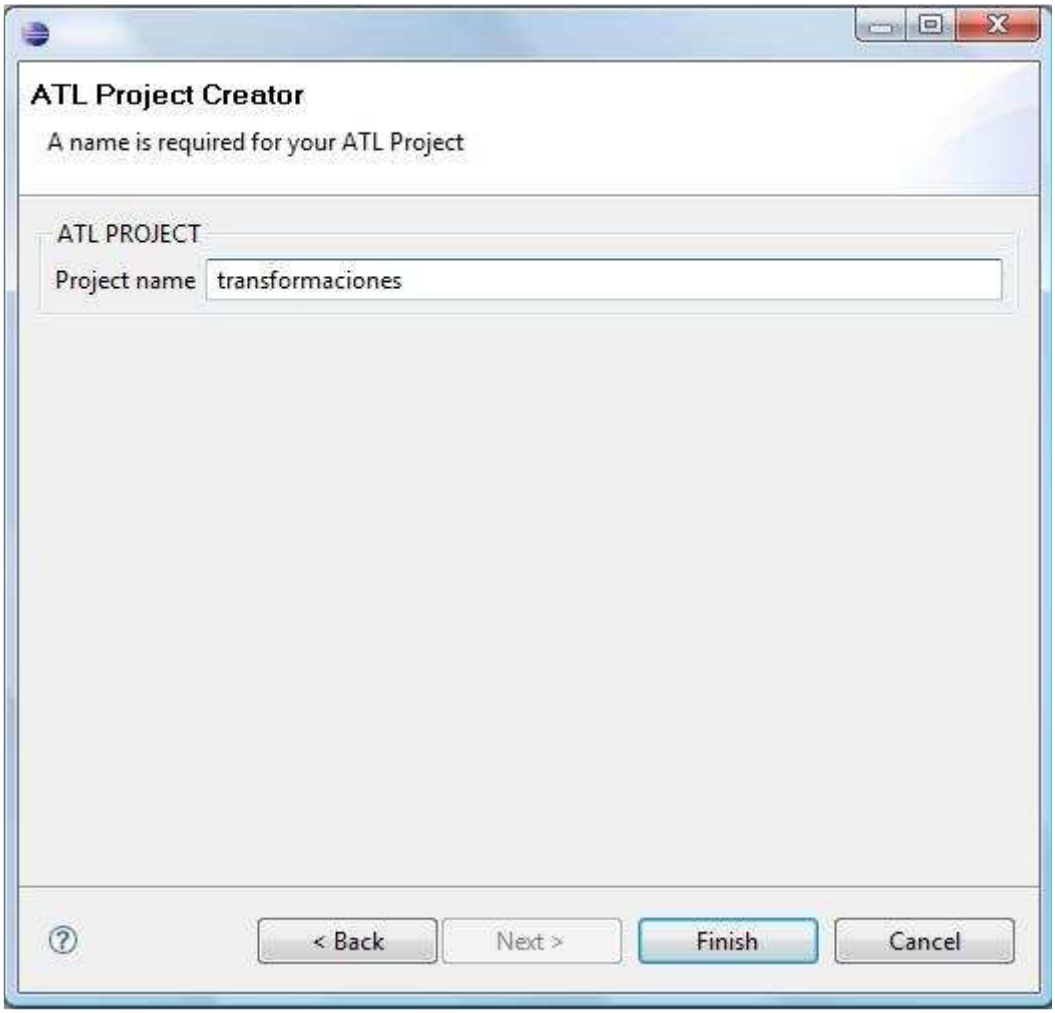

Figura 38 - Configuración proyecto ATL

Por último debemos completar el nombre del proyecto y presionar *Finish*. Con estos pasos hemos creado el proyecto para poder realizar las transformaciones.

Una vez creado el proyecto debemos crear las transformaciones propiamente dichas. Para poder realizar esto nos debemos parar sobre el proyecto anteriormente creado (transformaciones) como se muestra en la siguiente figura.

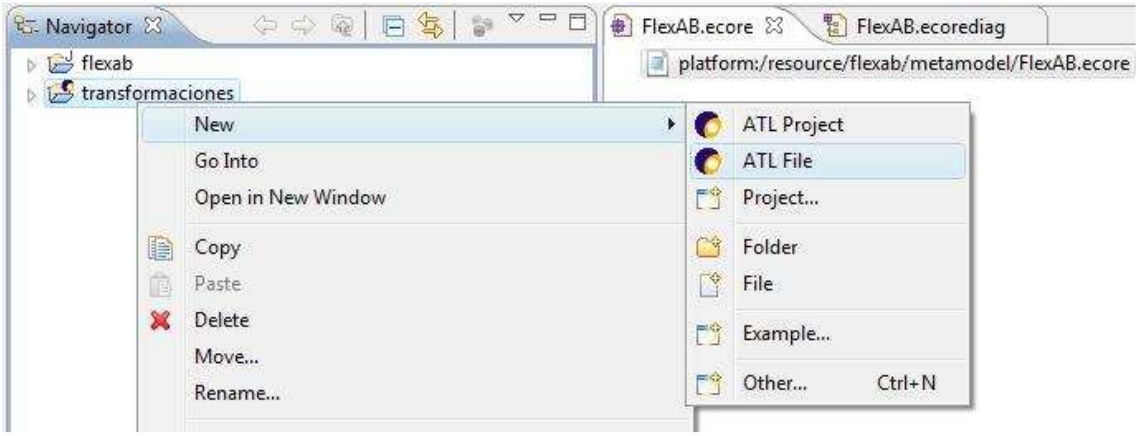

Figura 39 - Crear transformación

Debemos seleccionar la opción new-> ATL File y nos abrirá la siguiente pantalla.

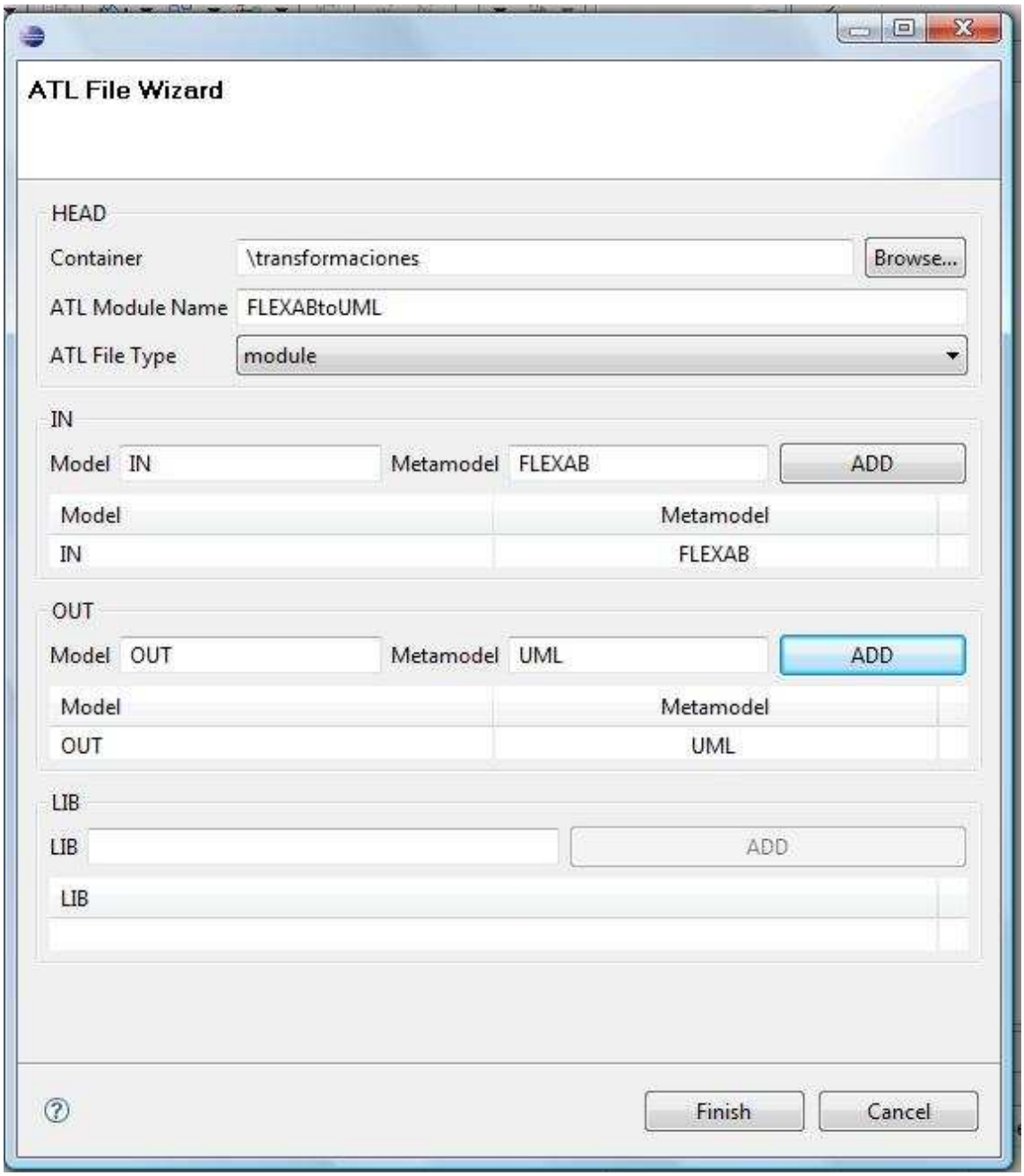

Figura 40 - Configuración de archivo ATL

Se deberán completar los datos como se muestra en la figura anterior. Después de completar los datos nos que da presionar *Finish.* Al presionar *Finish* quedará creado el archivo para realizar la transformación desde FlexAB a UML. A continuación se muestra como debería quedar creado el archivo de la transformación creada.

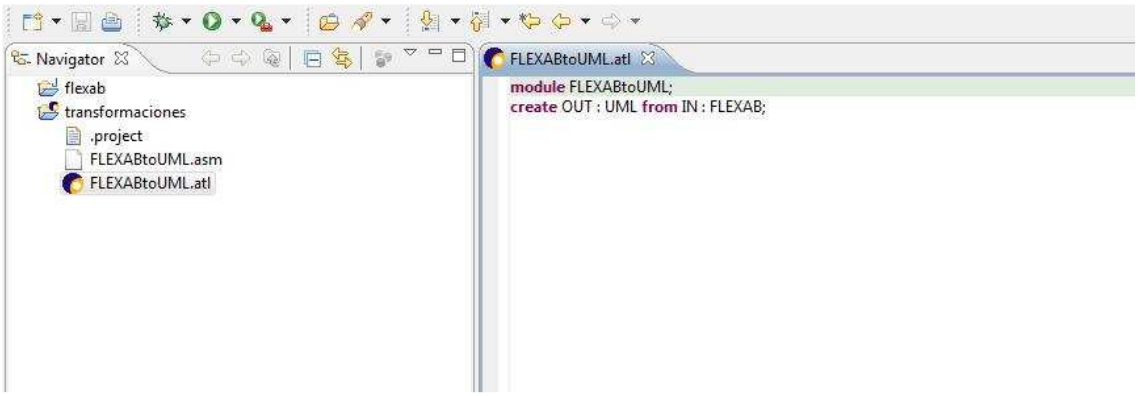

Figura 41 - Transformación FlexAB a UML

Los mismos pasos debemos seguir para generar la transformación de UML a FlexAB. A continuación mostramos la estructura con la que debe quedar el proyecto de transformaciones.

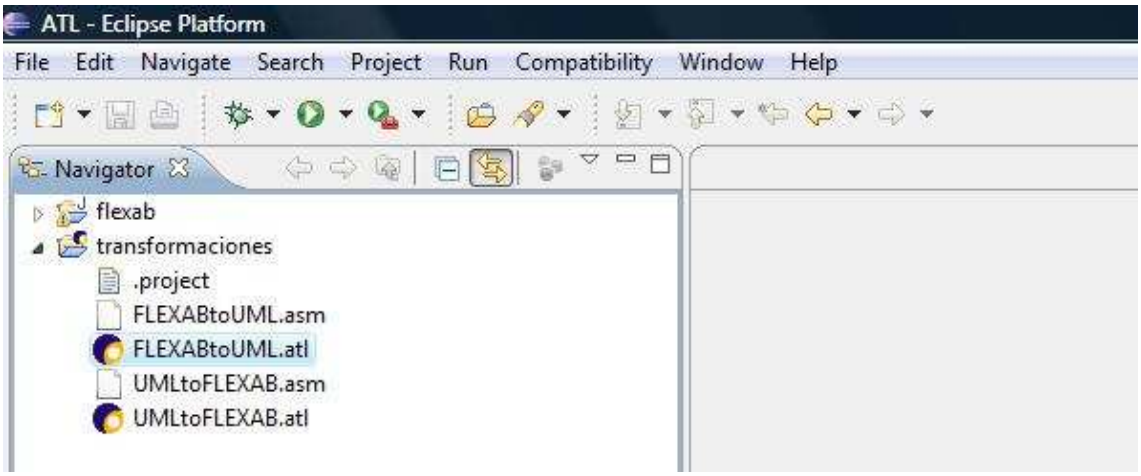

Figura 42 - Estructura Proyecto ATL

Hasta acá hemos armado los dos proyectos, ahora nos queda escribir las transformaciones que serán explicadas en la siguiente sección.

7.3 Implementación de reglas en transformaciones ATL

En esta sección no hablamos de la correspondencia entre las clases del modelo FlexAB y UML, sino que hacemos una breve explicación de cómo se escriben las reglas en las transformaciones.

En la figura a continuación mostramos la regla más común que se puede escribir en una transformación con ATL.

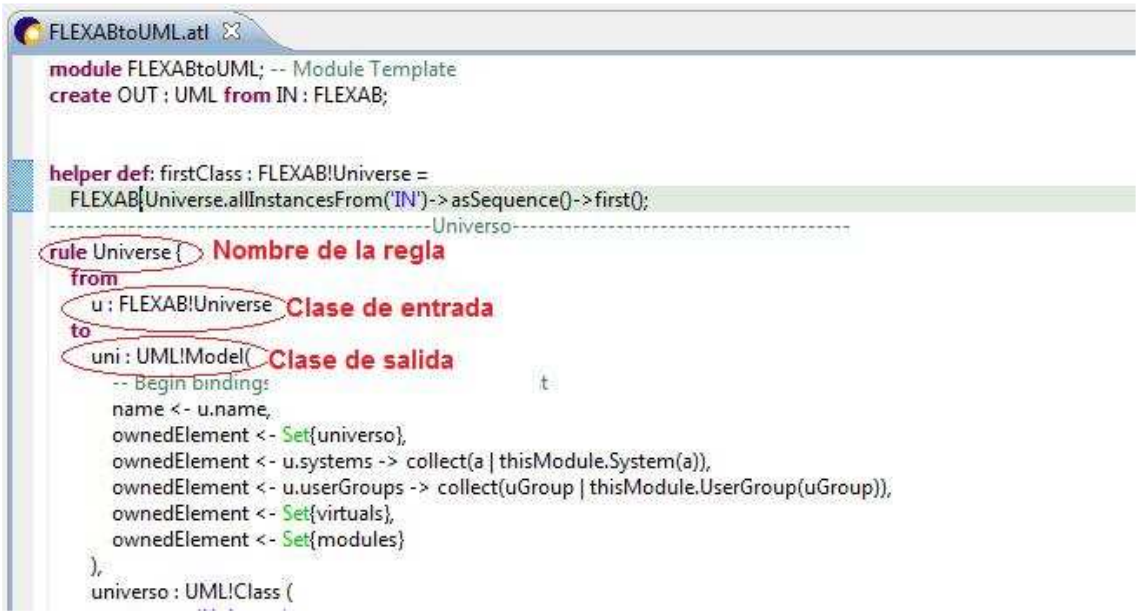

Figura 43 – Regla ATL Universe

Este tipo de reglas se ejecutan siempre, es decir que no necesitan ser invocadas de otra parte de la transformación. En la figura anterior se detallan el nombre de la regla, la clase de entrada de la transformación y la clase de salida que será producida. Cada regla tiene un cuerpo que es donde se define el mapeo entre la clase de entrada y la clase de salida.

Otro tipo regla son las que se ejecutan únicamente cuando son invocadas desde otra línea de la transformación. Este tipo de reglas se escribe de la misma forma que la regla anterior pero anteponiendoles la palabra lazy como se muestra en la siguiente figura.

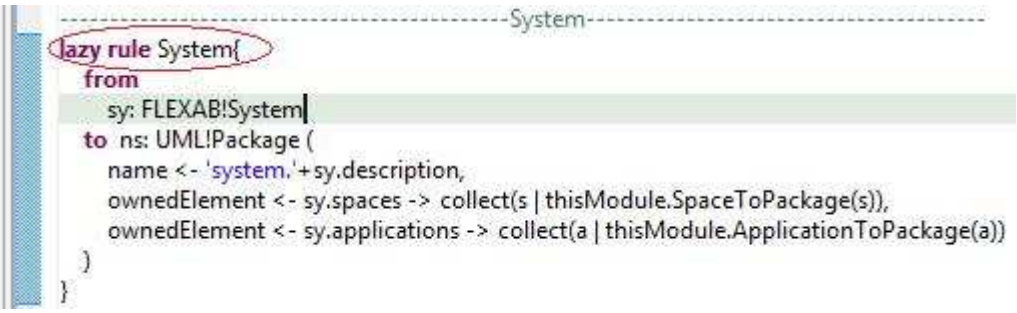

Figura 44 - Regla ATL System

Este tipo de reglas son invocadas escribiendo thisModule.”nombre de la regla”(parámetro). El parámetro debe coincidir con la clase de entrada de la regla escrita.

Otra de las cosas que podemos hacer en las transformaciones es definir funciones. A continuación se muestra un ejemplo de una función.

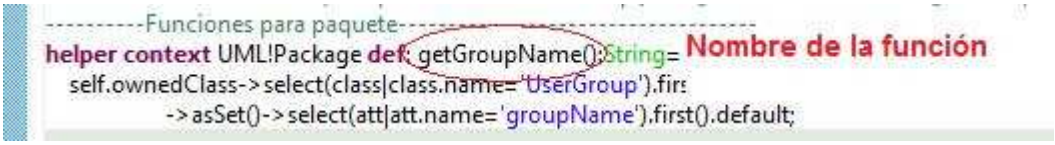

Figura 45 - Función ATL

En el caso anterior se define la función getGroupName() para la clase Package en el metamodelo UML que devuelve un atributo de tipo String. En la figura a continuación se muestra como invocar a la función anterior.

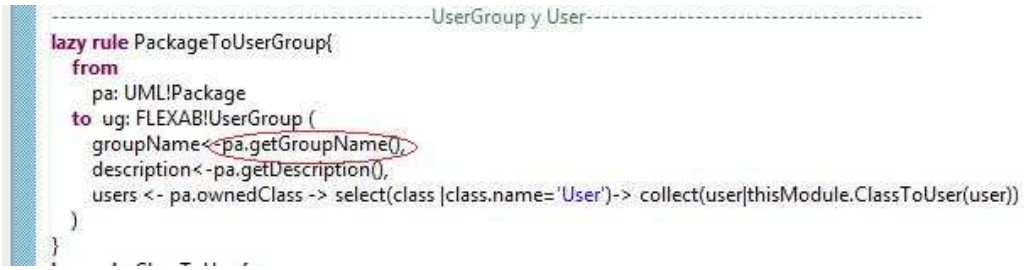

Figura 46 - Invocación de función ATL

Hemos visto en esta sección los distintos tipos de reglas que se han utilizados y la forma de declarar funciones e invocarlas. En la siguiente sección veremos como ejecutar las transformaciones.

7.4 Ejecución de transformaciones

Una vez configurado ambos proyectos (flexab y transformaciones), lo que debemos hacer es ejecutar cada una de las transformaciones. A continuación se detallan una serie de pasos que debemos seguir para configurar la ejecución de cada una de las transformaciones.

Lo primero que debemos hacer es ir a la siguiente opción de menú como se muestra en la siguiente figura.

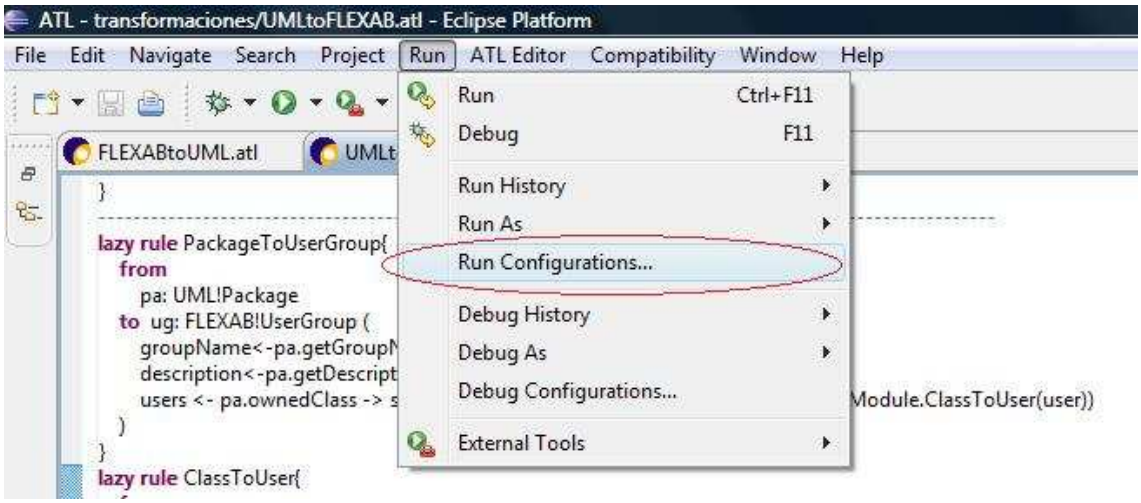

Figura 47 - Configuración de transformaciones

Una vez elegida la opción Run Configurations… se abrirá el siguiente diálogo.

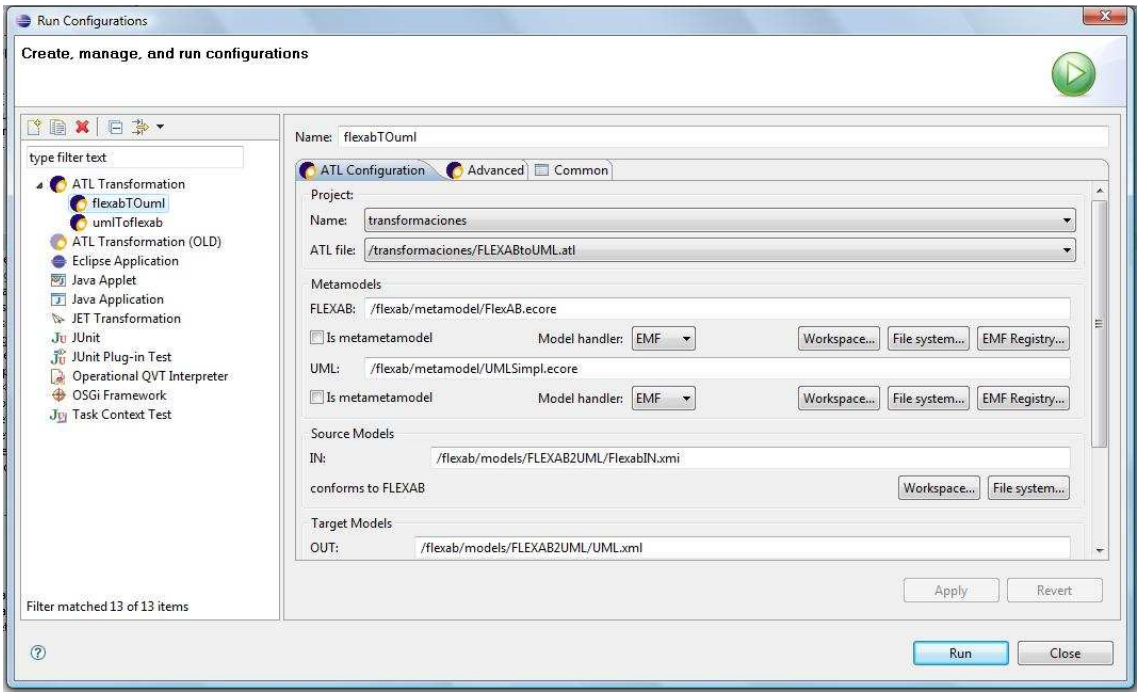

Figura 48 - Configuración transformación FlexAB

Para agregar una nueva configuración nos posicionamos sobre ATL Transformation, presionamos botón derecho y elegimos la opción new. A continuación debemos completar los datos como se muestra en la figura anterior. Los datos a completar corresponden a la transformación a correr, los metamodelos que participan en dicha transformación (el metamodelo de entrada y el de salida) y las instancias del modelo de entrada (Flexab.xmi) y cuál será el archivo de salida producido.

Lo anteriormente explicado hace referencia a la transformación que va desde FlexAB hacia UML.

Para la transformación de UML hacia FlexAB debemos configurar algo similar pero cambiando algunos datos. Los pasos a seguir son los mismos que ya se detallaron, sin embargo cambia cuál es el metamodelo de entrada, el metamodelo de salida y la transformación a correr. A continuación se muestra una figura con los datos correspondientes.

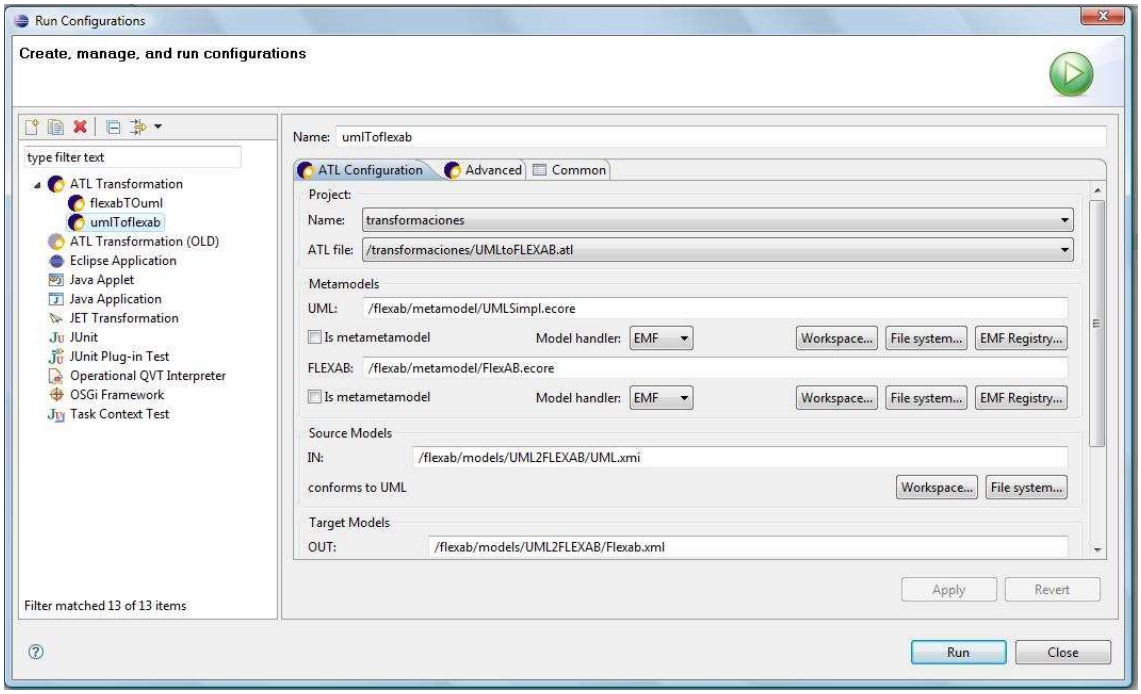

Figura 49 - Configuración de transformación UML

Ahora solo nos queda correr las transformaciones, para esto hay dos opciones. Una es presionando el botón Run en la pantalla de diálogos mostradas anteriormente. La otra opción es como se muestra en la siguiente figura.

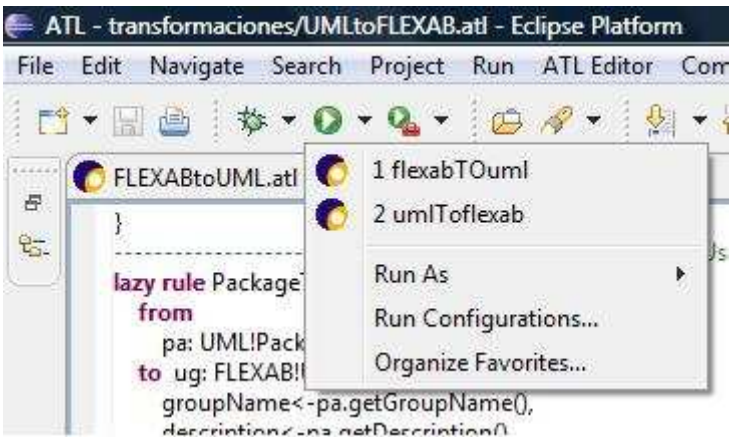

Figura 50 - Correr transformación

7.5 Relaciones de los elementos de FlexAB con los de UML

En este capítulo se explicará la correspondencia entre las clases de FlexAB y clases de UML. Se detallará clase por clase cuál es la correspondencia entre modelos. También se mostrará el código de la transformación utilizado para cada correspondencia de clase.

7.5.1 Comparación entre el concepto de sistema en UML y el concepto de universo en FlexAB

Los conceptos de sistema en UML y universo en FlexAB son muy similares. Tanto el sistema como el universo son elementos que actúan como contenedores de todos los elementos y relaciones necesarios para el desarrollo. Y ambos permiten organizar los modelos para su almacenamiento y manipulación.
Para el manejo de desarrollos complejos, UML permite que un sistema pueda ser dividido en subsistemas, mientras que FlexAB permite definir sistemas FlexAB. Por lo anteriormente detallado la correspondencia en UML de la clase Universo será la clase Model. La clase Model en UML cumple con el concepto de Universo es FlexAB porque es la clase raíz que puede contener paquetes, clases. En la figura siguiente se muestra como queda la regla escrita para la transformación desde FlexAB a UML.

```
-----------------------------------------Universo-----------------------------------------
rule Universe {
  from
    u : FLEXAB!Universe
  to
    uni : UML!Model(
      -- Begin bindings inherited from ModelElement
      name <- u.name,
      ownedElement <- Set{universo},
      ownedElement <- u.systems -> collect(a | thisModule.System(a)),
      ownedElement <- u.userGroups -> collect(uGroup | thisModule.UserGroup(uGroup)),
      ownedElement <- Set{virtuals},
      ownedElement <- Set{modules}
    ),
    universo : UML!Class (
      name <- 'Universo',
      ownedAttribute <- Set{name,serverHost,dbName,dbAudit}
    ),
    name : UML!Property (
      name <- 'name',
      default <- u.name
    ),
    serverHost : UML!Property (
      name <- 'serverHost',
      default <- u.serverHost
    ),
    dbName : UML!Property (
      name <- 'dbName',
      default <- u.dbName
    ),
    dbAudit : UML!Property (
      name <- 'dbAudit',
      default <- u.dbAudit
    )
```

Figura 51 - Regla Universe en FlexAB a UML

En la siguiente figura se muestra la transformación de UML a FlexAB para la clase Universo.

```
-----------------------------------------Universo-----------------------------------------
rule Universe {
  from
    uni : UML!Model
  to
    u : distinct FLEXAB!Universe foreach(c in uni.ownedElement->select(u|u.name='Universo'))(
      name <-c.ownedAttribute -> select(att |att.name='name')->collect(att|att.default),
      serverHost <-c.ownedAttribute -> select(att |att.name='serverHost')->collect(att|att.default),
      dbName <-c.ownedAttribute -> select(att |att.name='dbName')->collect(att|att.default),
      dbAudit <-c.ownedAttribute -> select(att |att.name='dbAudit')->collect(att|att.default),
      userGroups <- uni.ownedElement->select(group |group.name.substring(1,4)='user')->collect(group|thisModule.PackageToUserGroup(group)),
      virtuals <- uni.ownedElement->select(virtual|virtual.name='virtuals').first().ownedClass->collect(virtual|thisModule.ClassToVirtual(virtual)),
      modules <- uni.ownedElement->select(package|package.name='modules').first().ownedClass->collect(classModule|thisModule.ClassToModule(classModule))
    )
}
```

Figura 52 - Regla Universe en UML a FlexAB

7.5.2 Comparación entre el concepto de subsistema en UML y el concepto de sistema en FlexAB

Los conceptos de subsistema en UML y universo en FlexAB son muy parecidos.
Ambos lenguajes permiten dividir la complejidad del dominio, UML mediante los subsistemas y FlexAB mediante los sistemas. Estos elementos permiten agrupar objetos relacionados para lograr cierta funcionalidad delimitada por el objetivo general del subsistema. En los dos casos se busca tener un fuerte acoplamiento funcional dentro de

cada subsistema y un acoplamiento débil entre subsistemas, en otras palabras, se busca tener la mínima comunicación entre los diferentes subsistemas.
UML permite que cada subsistema se pueda subdividir en subsistemas adicionales, posibilitando un anidamiento de subsistemas. FlexAB permite definir dentro de los sistemas, aplicaciones y espacios de trabajo. Ambos recursos se utilizan para subdividir nuevamente la complejidad en términos más simples.
Por lo especificado anteriormente los sistemas que pertenecen a un Universo serán representados por paquetes en UML. En la figura siguiente se muestra la regla correspondiente para esta asociación.

```
----------------------------------------System----------------------------------------
lazy rule System{
  from
    sy: FLEXAB!System
  to  ns: UML!Package (
    name <- 'system.'+sy.description,
    ownedElement <- sy.spaces -> collect(s | thisModule.SpaceToPackage(s)),
    ownedElement <- sy.applications -> collect(a | thisModule.ApplicationToPackage(a))
  )
}
```

Figura 53 - Regla System en FlexAB a UML

Los sistemas serán representados en UML como paquetes y estos paquetes serán contenidos por la clase Model de UML que fue explicada en el punto 7.3.1

7.4.3 Comparación entre paquetes y el concepto de UserGroup y User

Un paquete permite organizar en grupos los elementos. Son la base del control de configuraciones, del almacenamiento y del control de accesos.
Los paquetes pueden tener relaciones de dependencia con otros paquetes. Estas relaciones indican que a los elementos del paquete cliente se les otorga permiso para relacionarse con los elementos del paquete proveedor.
En FlexAB la clase UserGroup contiene usuario (clase User en FlexAB), es por esto que las instancias de UserGroup serán representadas por paquetes, y estos paquetes contendrán instancias de clases User en UML. En FlexAB tendremos a la clase Universe que contiene una lista de UserGroup y la correspondencia en UML será la clase Model que contendrá paquetes simulando a UserGroup y cada paquete contendrá instancias de clases User en uml que se corresponde con la clase User de FlexAB. En la figura siguiente se puede apreciar lo detallado anteriormente.

```
----------------------------------------UserGroup y User----------------------------------------
lazy rule UserGroup{
  from
    us: FLEXAB!UserGroup
  to  ns: UML!Package (
    name <- 'userGroup.'+us.groupName,
    ownedClass <- Set{classUserGroup},
    ownedClass <- us.users -> collect (user | thisModule.UserToClass(user))
  ),
  classUserGroup : UML!Class (
    name <- 'UserGroup' ,
    ownedAttribute <- Set{groupName,descriptionGroup}
  ),
  groupName : UML!Property (
    name <- 'groupName' ,
    default<-us.groupName
  ),
  descriptionGroup : UML!Property (
    name <- 'description',
    default <- us.description
  )
}
lazy rule UserToClass{
  from
    userFlex: FLEXAB!User
  to class: UML!Class (
    name <- 'User',
    ownedAttribute <- Set{userID,userName,password,loginTries,state,userLanguage,passwordExpiresOn,email,charMinPassword,
                          passwordAlphanumeric,configLevel,smtpServer,smtpUser,smtpPassword}
  ),
  userID : UML!Property (
    name <- 'userID',
    default<-userFlex.userID
  ),
  userName : UML!Property (
    name <- 'userName',
    default<-userFlex.userName
  ),
  password : UML!Property (
    name <- 'password',
    default<-userFlex.password
  ),
  loginTries : UML!Property (
    name <- 'loginTries',
    default<-userFlex.loginTries
  ),
  state : UML!Property (
    name <- 'state',
    default<-userFlex.state
  ),
  userLanguage : UML!Property (
    name <- 'userLanguage',
    default<-userFlex.userLanguage
  ),
```

Figura 54 - Regla UserGroup de FlexAB a UML

Como se aprecia en la figura anterior la relación UserGroup ->User en FlexAB quedará como la relación Package->Class en UML.

7.5.3 Comparación entre aplicación en FlexAB e instancia de sistema en UML

Dado que UML es un lenguaje de modelado, no esta orientado a la instanciación de los elementos de modelado, sino a su definición. En este sentido, FlexAB va mas allá del simple modelado permitiendo la instanciación de los elementos definidos. Debido a esto, el concepto de aplicación se asemeja más al concepto de instancia de sistema en UML. Según lo explicado anteriormente las instancias de aplicación en FlexAB serán representadas en UML como paquetes. Estos paquetes a su vez contendrán clases que

representan a las instancias de ApplicationSecurity. En la figura siguiente se ve la regla con la correspondencia entre clases.

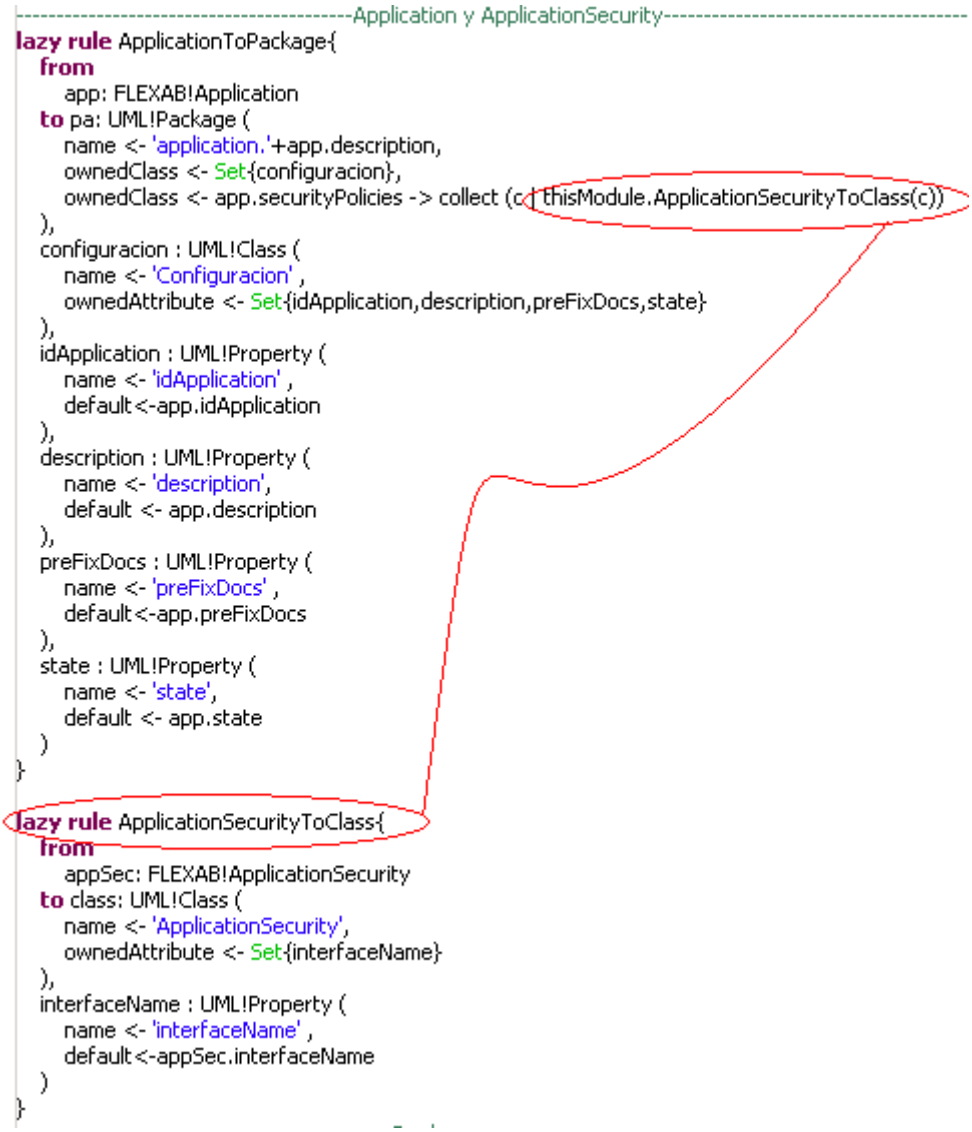

Figura 55 - Regla Application de FlexAB a UML

Por lo explicado anteriormente, en FlexAB tendremos la relación Application->ApplicationSecurity y en UML esto quedará representado por paquetes->clases.

7.5.4 Comparación entre el concepto paquete en UML y el concepto de espacio en FlexAB

Un espacio posee un conjunto de definiciones de clases y de relaciones entre éstas. Las relaciones entre clases son relativas al espacio, es decir que las definiciones de clases pueden especificar relaciones con otras clases de manera diferente en distintos espacios. Los espacios se crean dentro de los sistemas. Las aplicaciones creadas dentro de un

mismo sistema, poseen los mismos espacios.
El concepto de espacio en FlexAB se puede comparar a nivel sistema con el concepto de paquete en UML y a nivel aplicación, funciona como contenedores de objetos. En la figura a continuación se ve relación entre clases explicada anteriormente.

```
-----------------------------------------Space-----------------------------------
lazy rule SpaceToPackage{
  from
    sp: FLEXAB!Space
  to pa: UML!Package (
    name <- sp.description,
    ownedClass <- sp.classes -> collect (c | thisModule.RealToClass(c))
  )
}
```

Figura 56 - Reglas de Space de FlexAB a UML

7.5.6 Comparación entre el concepto clase en UML y el concepto clase virtual en FlexAB

Una clase virtual en FlexAB es tipo de clase que recupera datos desde otras clases. Su finalidad es ser utilizada como medio para obtener ciertos datos. Sus objetos no pueden verse en el árbol estructural.
Aunque no existe un concepto en UML similar a las clases virtuales, podría simularse mediante una clase que no defina atributos, y que tenga un método consulta que se corresponda con el resultado del FQL de la clase virtual. En la siguiente figura se observa la regla de la transformación que involucra las clases virtuales.

```
-----------------------------------------Virtual y Module-----------------------------------
lazy rule VirtualToClass{
  from
    v: FLEXAB!Virtual
  to class: UML!Class (
    name <- 'Virtual',
    isAbstract <- true,
    ownedAttribute <- Set{shortName,description,version}
  ),
  shortName : UML!Property (
    name <- 'shortName',
    default<-v.shortName
  ),
  description : UML!Property (
    name <- 'description',
    default<-v.description
  ),
  version : UML!Property (
    name <- 'version',
    default<-v.version
  )
}
```

Figura 57 - Regla Virtual y Module de FlexAB a UML

7.5.7 Comparación entre el concepto clase en UML y el concepto clase real en FlexAB

Las clases reales en FlexAB son las que permiten generar una instancia de un Objeto Real en una Aplicación, la relación entre hijos y padres; puede verse en el árbol estructural.
Debido a que el único tipo de clase que se puede instanciar por el usuario en FlexAB son las clases reales, son las que mas se asemejan a las clases concretas de UML.
Ambas permiten declarar atributos y métodos. En la figura siguiente se puede observar la regla de la transformación para la relación detallada entre modelos anteriormente.

```
----------------------------------------Real----------------------------------------
lazy rule RealToClass{
  from
    cr: FLEXAB!ClassReal
  to
    c: UML!Class (
       name <- cr.description,
       ownedAttribute <- Set{shortName,description,icon,auditObjects,version,isAbstract}
  ),
  shortName : UML!Property (
     name <- 'shortName' ,
     default<-cr.shortName
  ),
  description : UML!Property (
     name <- 'description',
     default <- cr.description
  ),
  icon : UML!Property (
     name <- 'icon' ,
     default<-cr.icon
  ),
  auditObjects : UML!Property (
     name <- 'auditObjects',
     default <- cr.auditObjects
  ),
  version : UML!Property (
     name <- 'version',
     default <- cr.version
  ),
  isAbstract : UML!Property (
     name <- 'isAbstract',
     default <- cr.isAbstract
  )

}
```

Figura 58 - Regla Real en FlexAB a UML

En este capítulo se explicó la transformación de clases de FlexAB a UML. La transformación de UML a FlexAB es lo contrario a lo que se explicó anteriormente es por eso que no se detalla la transformación clase por clase. A continuación se muestran algunas de las clases mapeadas de UML a FlexAB.

```
----------------------------------------Universo----------------------------------------
rule Universe {
  from
    uni : UML!Model
  to
    u : distinct FLEXAB!Universe  foreach(c in uni.ownedElement->select(u|u.name='Universo'))(
      name <-c.ownedAttribute -> select(att |att.name='name')->collect(att|att.default),
      serverHost <-c.ownedAttribute -> select(att |att.name='serverHost')->collect(att|att.default),
      dbName <-c.ownedAttribute -> select(att |att.name='dbName')->collect(att|att.default),
      dbAudit <-c.ownedAttribute -> select(att |att.name='dbAudit')->collect(att|att.default),
      userGroups <- uni.ownedElement->select(group |group.name.substring(1,4)='user')->collect(group|thisModule.PackageToUserGroup(group)),
      virtuals <- uni.ownedElement->select(virtual|virtual.name='virtuals').first().ownedClass->collect(virtual|thisModule.ClassToVirtual(virtual)),
      modules <- uni.ownedElement->select(package|package.name='modules').first().ownedClass->collect(classModule|thisModule.ClassToModule(classModule))
    )
}
```

Figura 59 - Regla Universe en UML a FlexAB

```
----------------------------------------UserGroup y User----------------------------------------
lazy rule PackageToUserGroup{
  from
    pa: UML!Package
  to  ug: FLEXAB!UserGroup (
    groupName<-pa.getGroupName(),
    description<-pa.getDescription(),
    users <- pa.ownedClass -> select(class |class.name='User')-> collect(user|thisModule.ClassToUser(user))
  )
}
lazy rule ClassToUser{
  from
    classUser: UML!Class
  to  ug: FLEXAB!User (
    userName <- classUser.getPropertyString('userName'),
    configLevel <- classUser.getConfigLevel(),
    passwordExpiresOn <- classUser.getPropertyInteger('passwordExpiresOn'),
    smtpPassword <- classUser.getPropertyString('smtpPassword'),
    charMinPassword <- classUser.getPropertyInteger('charMinPassword'),
    smtpServer <- classUser.getPropertyString('smtpServer'),
    userLanguage <- classUser.getPropertyString('userLanguage'),
    state <- classUser.getPropertyBoolean('states'),
    passwordAlphanumeric <- classUser.getPropertyBoolean('passwordAlphanumeric'),
    email <- classUser.getPropertyString('email'),
    password <- classUser.getPropertyString('password'),
    smtpUser <- classUser.getPropertyString('smtpUser'),
    loginTries <- classUser.getPropertyInteger('loginTries'),
    userID <- classUser.getPropertyString('userID')
  )
}
```

Figura 60 - Regla UserGroup en UML a FlexAB

```
----------------------------------------Virtual y Module----------
lazy rule ClassToVirtual{
  from
    classVirtual: UML!Class
  to class:FLEXAB!Virtual (
    shortName <- classVirtual.getPropertyString('shortName'),
    description <- classVirtual.getPropertyString('description'),
    version <- classVirtual.getPropertyString('version')
  )
}
```

Figura 61 - Regla Virtual en UML a FlexAB

8 Caso de Estudio

En este capítulo se mostrará el funcionamiento de las transformaciones, que han sido implementadas en los capítulos anteriores, con un caso de prueba específico. Se deberá instanciar el modelo de FlexAB para poder correr la transformación que tendrá como entrada dicho modelo y producirá como salida el correspondiente modelo en UML. Posteriormente para probar la segunda transformación se deberá instanciar el modelo de UML para correr la transformación que produce como salida el modelo correspondiente de FlexAB. Los modelos de salida serán generados automáticamente por las transformaciones que se presentan en esta tesis.

8.1 Crear instancias del modelo FlexAB

En esta sección explicaremos la instanciación del modelo FlexAB. Para lograr esto debemos abrir el archivo FlexAB.ecore, a continuación nos paramos sobre el paquete main y en este sobre la clase Universe. Una vez situado sobre esta clase debemos presionar botón derecho y seleccionar la opción *Create Dynamic Instance* como se indica en la figura siguiente.

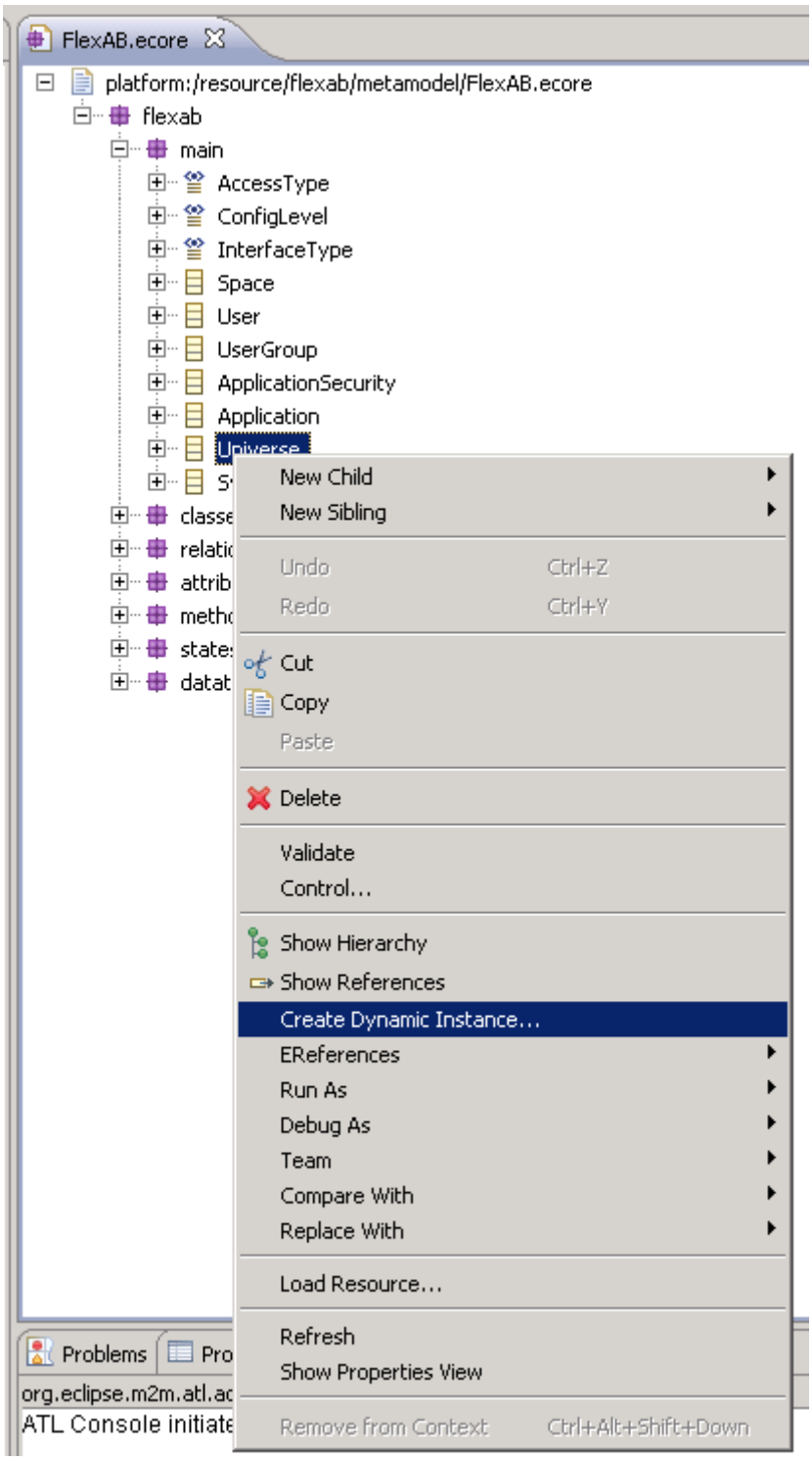

Figura 62 – Crear instancia xmi de FlexAB

En la figura anterior se aprecia que la opción *Create Dynamic Instance* se selecciona desde la clase Universe, esto es debido a que las instancias de un modelo deben ser creadas a partir de una clase raíz y como se ha explicado en los capítulos anteriores Universe es la clase raíz del metamodelo FlexAB.

A continuación se nos abrirá un diálogo donde debemos completar el nombre del archivo y la carpeta en donde se guardará este archivo. En la figura siguiente se aprecia

lo detallado anteriormente.

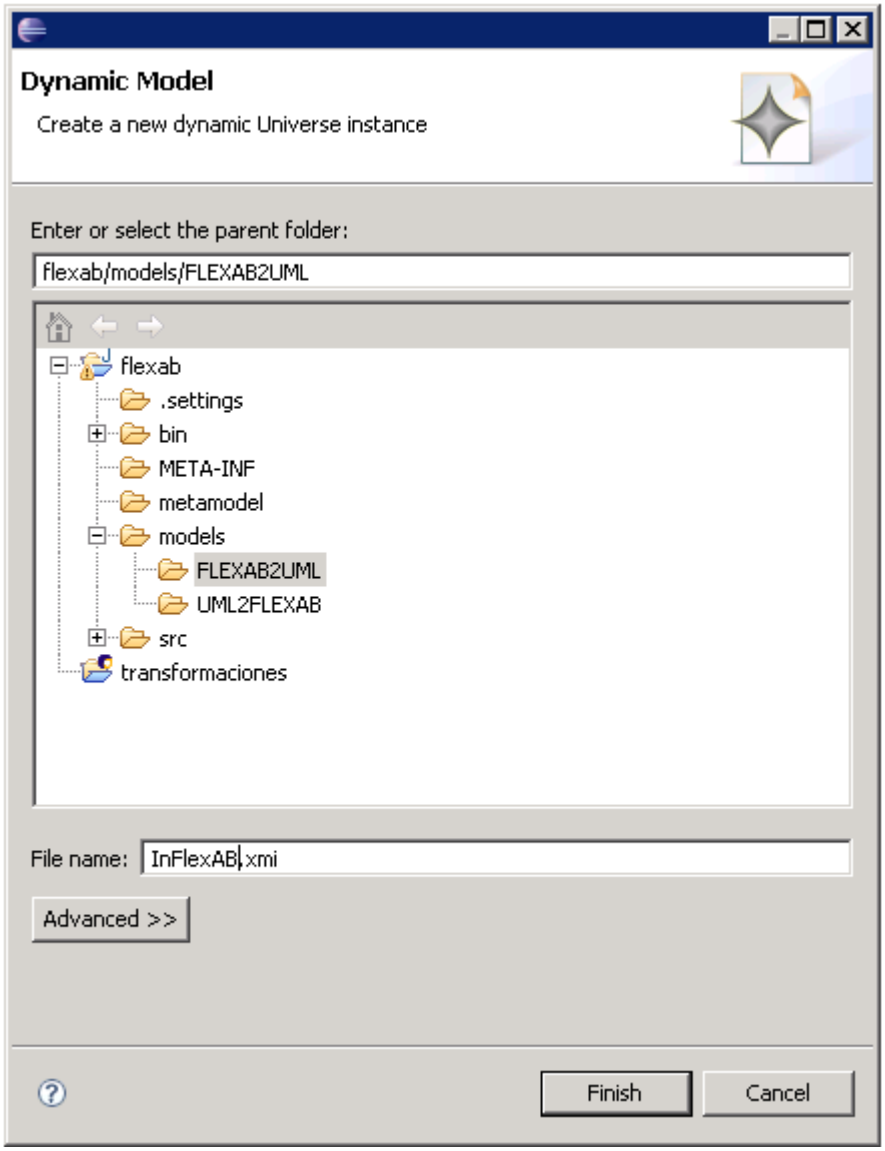

Figura 63 - Crear InFlexAB.xmi

Una vez generado el archivo debemos completar los datos de las instancias. Para completar los datos de la Instancia Universe debemos seleccionar la clase Universe en el archivo xmi anteriormente creado y en el cuadro de Properties nos aparecerá los atributos a completar como se muestra en la figura a continuación.

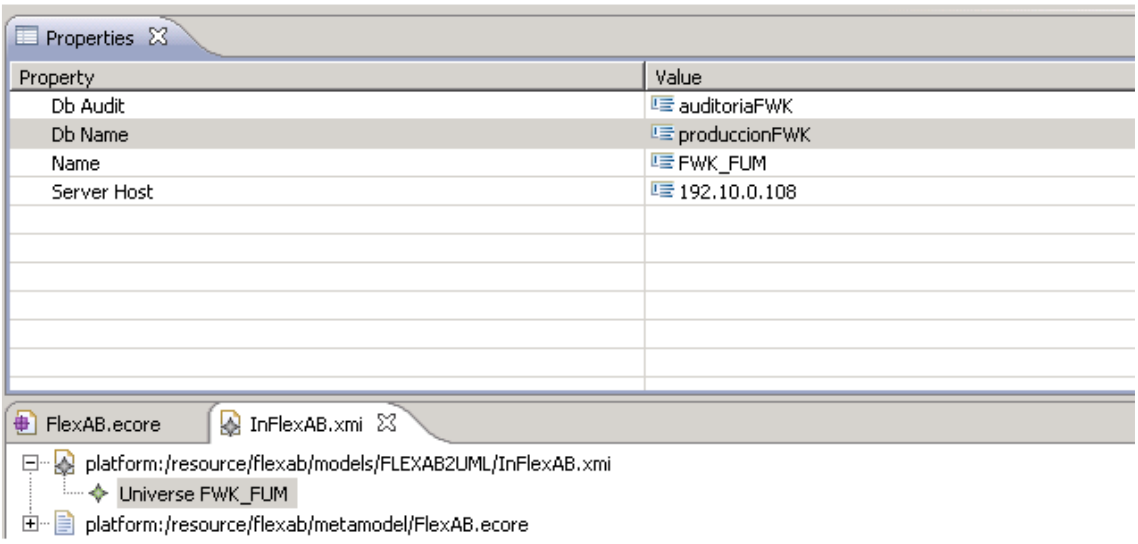

Figura 64 - Instanciar clase Universe en FlexAB

Una vez completados los datos de la instancia Universe debemos agregar el resto de las instancias, para lograr esto nos posicionamos sobre la clase Universe y al presionar botón derecho nos aparecerá la opción *New Child* como se muestra en al siguiente figura.

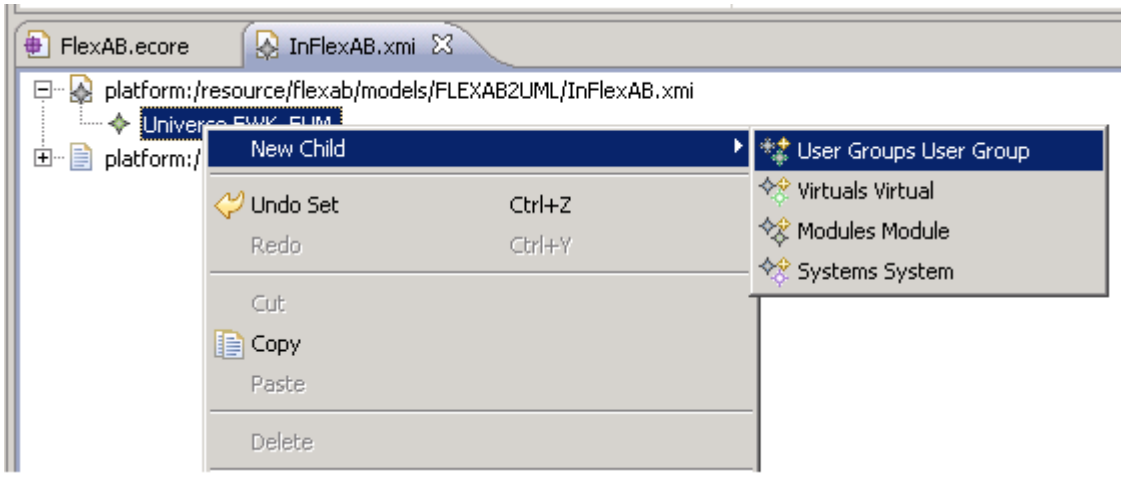

Figura 65 - Agregar relaciones a Universe

Como se puede apreciar nos aparecerán las opciones de que clases se pueden relacionar con la instancia de Universe. Está opciones se listan dependiendo de las relaciones que tenga en el metamodelo la clase a la cuál se le quiere agregar un nuevo hijo. Es decir que si la clase Universe se relaciona con una clase x, una clase e y una clase z cuando se quiera agregar un hijo a Universe se podrá agregar algunas de estas tres clases (x, e o z). A continuación elegimos agregar UserGroup a la instancia ya creada de Universe. Después de completar los datos de la instancia agregada UserGroup el gráfico quedará

como se muestra en la siguiente figura.

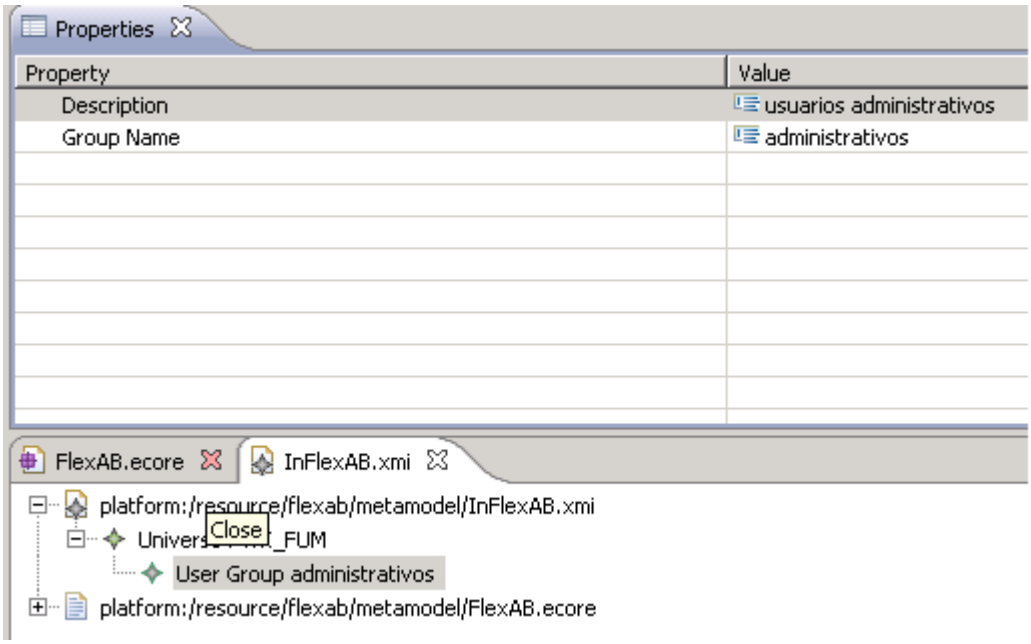

Figura 66 - Crear instancia de UserGroup en FlexAB

A continuación le agregamos un usuario al group user instanciado anteriormente. En la figura siguiente podemos ver como se instancia un usuario.

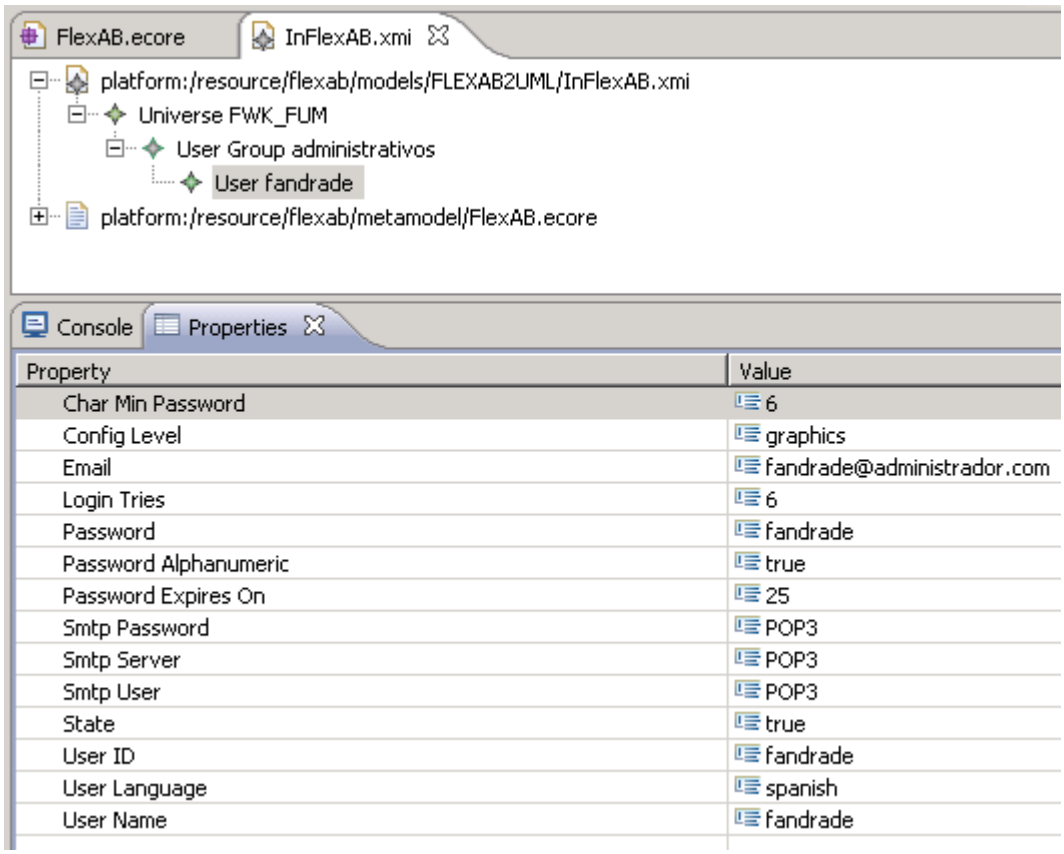

Figura 67 - Crear instancia de User en FlexAB

De la misma manera que hemos instanciado universe, usergroup y user se hace con el resto de las clases de FlexAB. En la figura siguiente se muestra el archivo InFlexAB.xmi con un conjunto de instancias que servirán como entrada en la transformación de FlexAB a UML.

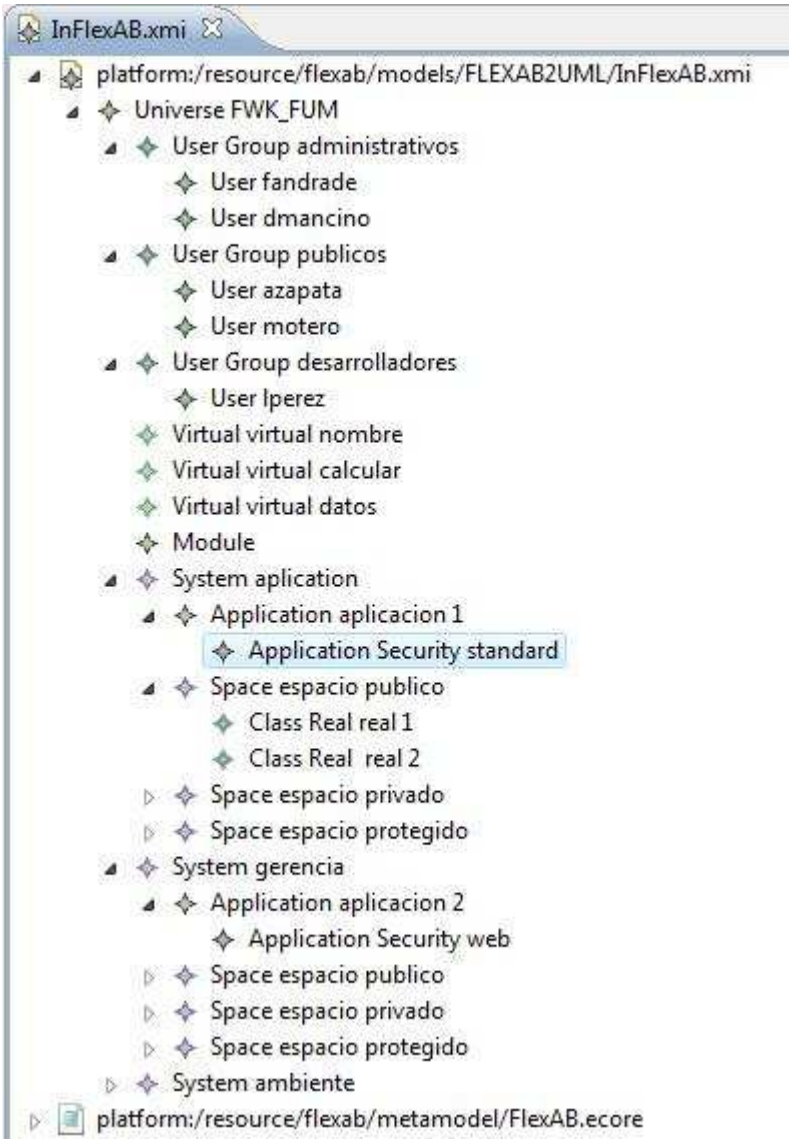

Figura 68 - Instancia de modelo FlexAB completo

8.2 Generar modelo de salida uml

El paso siguiente después de haber creado las instancias es correr la transformación. Recordemos que el capítulo 7 en la sección 7.4 se mostró como configurar la transformación a correr. En la figura a continuación se muestra la configuración para correr la transformación.

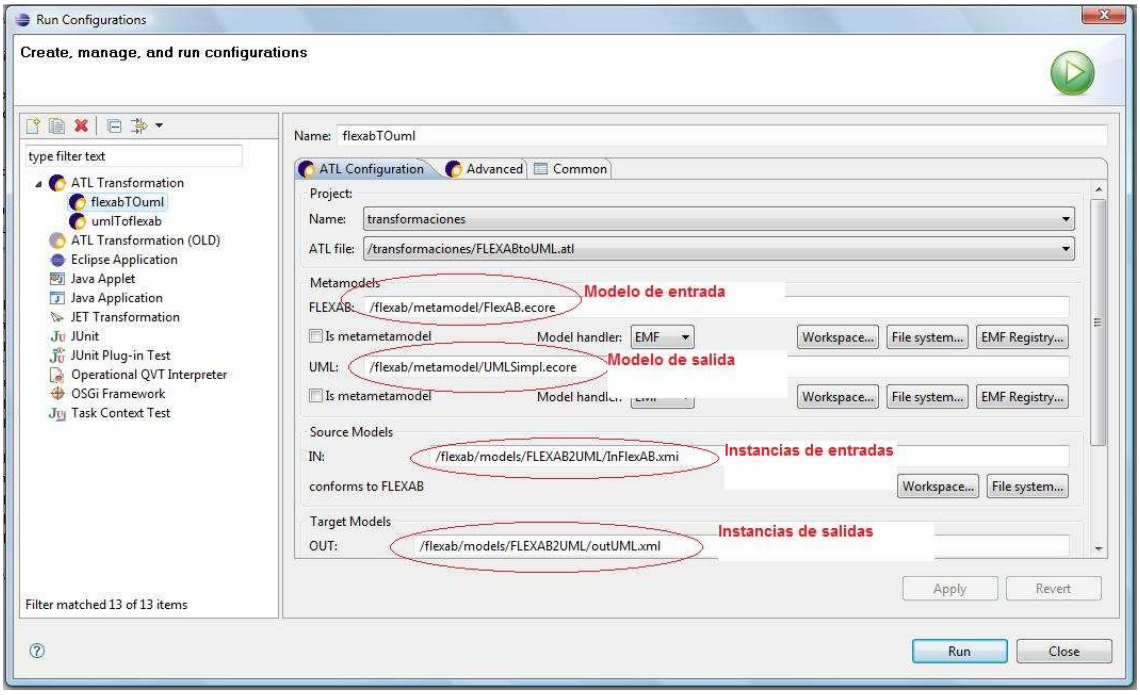

Figura 69 - Correr transformación de FlexAB a UML

Una vez corrida la transformación quedarán creadas las instancias correspondientes de UML. En la figura a continuación se puede apreciar lo anteriormente detallado. Las líneas en la figura marca la relación entre instancias de ambos modelos.

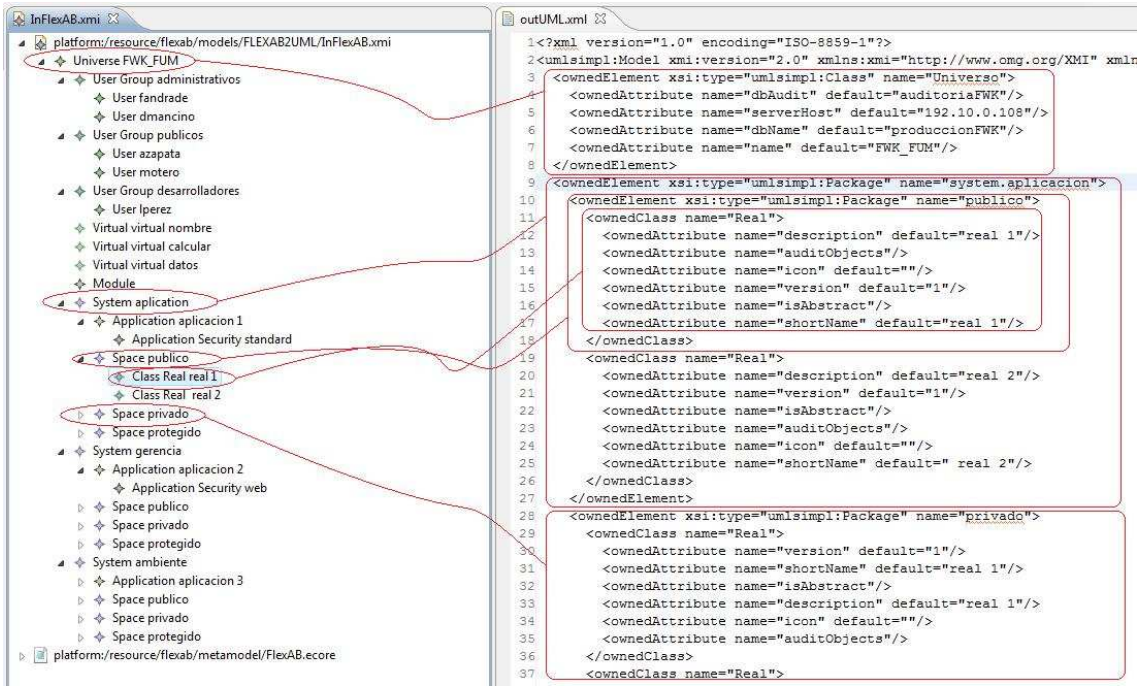

Figura 70 - Entrada y salida de transformación de FlexAB a UML

De la misma manera que hemos realizado la transformación de FlexAB a UML se realizará la prueba de UML a FlexAB en las siguientes dos secciones.

8.3 Crear instancias del modelo UML

De manera similar a como se instancia el modelo (ver 8.1) debemos hacer lo mismo para la transformación que va desde UML a FlexAB. En la figura siguiente se muestra las instancias correspondientes a clases del modelo de UML.

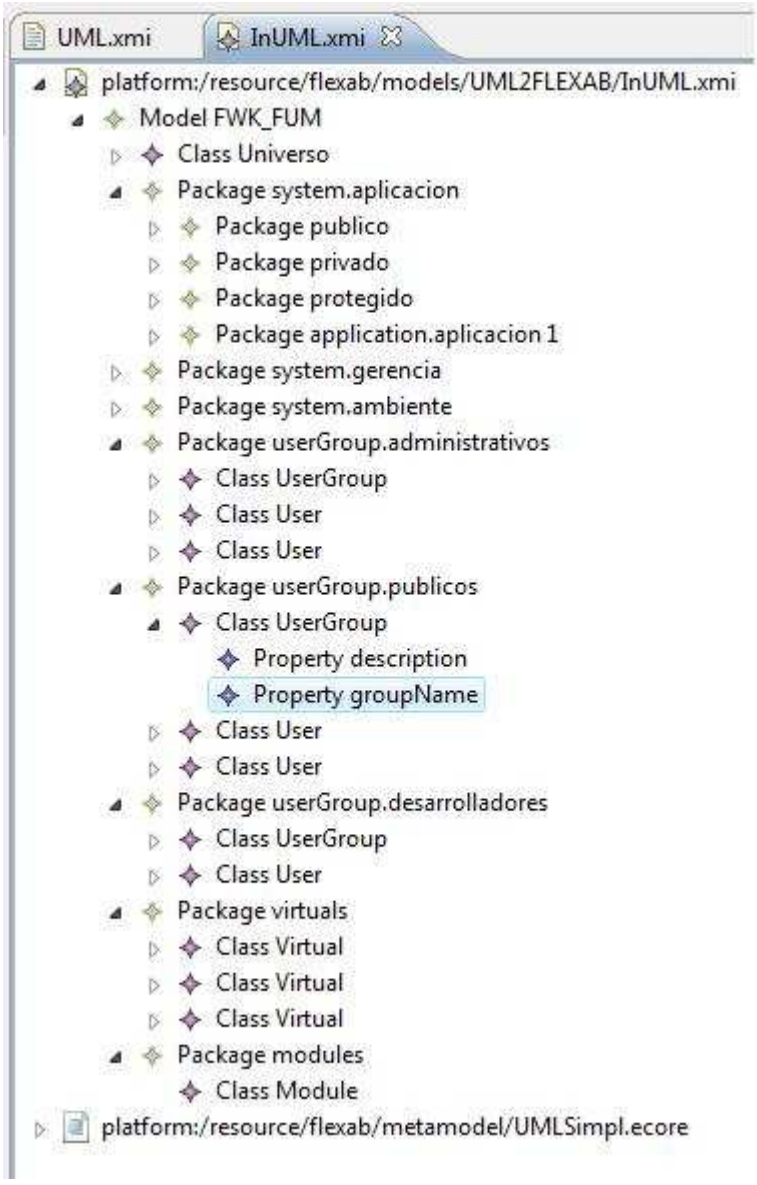

Figura 71 - InUML.xmi

Una vez que hemos instanciados las clases de UML debemos correr la transformación.

8.4 Generar modelo de salida FlexAB

Como se dijo anteriormente ahora nos queda correr la transformación. A continuación se muestra en la figura la configuración de la transformación.

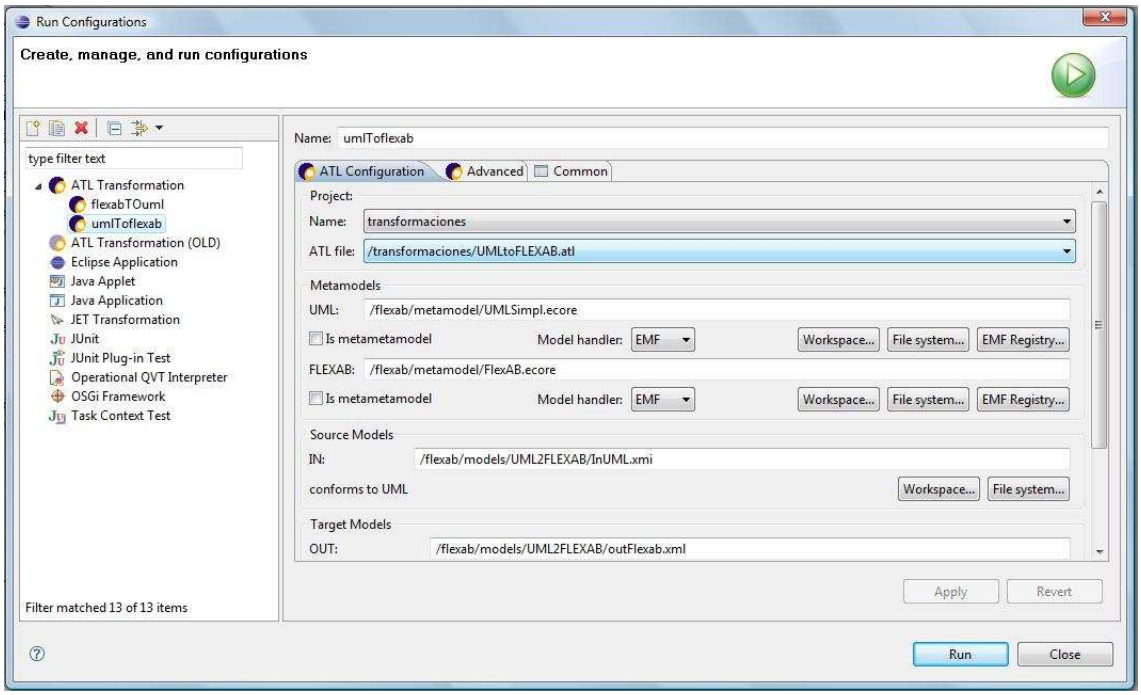

Figura 72 - Correr transformación de UML a FlexAB

Al correr la transformación obtendremos como salida las instancias correspondientes del modelo FlexAB. En la figura siguiente se puede apreciar esto.

Figura 73 - Salida de transformación de UML a FlexAB

En este capítulo hemos explicado la ejecución de un ejemplo corriendo en ambas transformaciones. Primero se explicó la transformada que va desde FlexAB hacia UML y por último se explicó la transformada inversa. No se hizo hincapié en como correr las transformaciones porque ya se había explicado en el capítulo anterior.

9 Conclusiones

El Desarrollo de Software Dirigido por Modelos MDD (por sus siglas en inglés: Model Driven software Development) se ha convertido en un nuevo paradigma de desarrollo software. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. El adjetivo "dirigido" (*driven)* en MDD, a diferencia de "basado" (*based)*, enfatiza que este paradigma asigna a los modelos un rol central y activo: son al menos tan importantes como el código fuente. Los modelos se van generando desde los más abstractos a los más concretos a través pasos de transformación y/o refinamientos, hasta llegar al código aplicando una última transformación. La transformación entre modelos constituye el motor del MDD.

Generalmente UML es el lenguaje elegido para definir los modelos en MDD. UML es un estándar abierto y es el estándar de facto para el modelado de software. UML es un lenguaje de modelado de software de propósito general que podemos aplicar de varias formas. Las técnicas de modelado apropiadas para diseñar una aplicación de telefonía, embebida y de tiempo real y las técnicas de modelado para desarrollar una aplicación de comercio electrónico son bastante diferentes, sin embargo podemos usar UML en ambos casos.
El uso de UML como lenguaje en MDD tiene los siguientes beneficios:
El uso de perfiles de UML para MDD nos permite aprovechar la experiencia que se ha ganado en el desarrollo de este lenguaje. Esto significa que podemos proveer un ambiente de modelado personalizado sin afrontar el costo de diseñarlo e implementarlo desde cero;
UML es un estándar abierto y el estándar de facto para el modelado de software en la industria;
UML ha demostrado ser durable, su primera versión apareció en 1995;
El éxito de UML ha propiciado la disponibilidad de muchos libros y cursos de muy buena calidad;
La mayoría de los estudiantes de las carreras relacionadas con la ingeniería de software han aprendido algo de UML en la universidad;
Existen muchas herramientas maduras que soportan UML;
Muchas organizaciones necesitan desarrollar varios tipos diferentes de software. Si podemos usar una solución de modelado en común, configurada de forma apropiada para describir cada uno de esos dominios de software, entonces será más fácil lidiar con su posterior integración.

Después de analizar en detalle los conceptos de UML y el metamodelo de FlexAB se ha decidido no incluir todas las clases en las transformaciones y poner todo el esfuerzo en un subconjunto de ellas a la hora de realizar la conversión.

Se analizó la correspondencia entre UML y un DSL - en particular el lenguaje de modelado de FlexAB.
A lo largo de esta tesis se ha estudiado distintas técnicas de modelado y el lenguaje de transformaciones ATL.

Como objetivo de esta tesis se ha creado una herramienta de software que brindará soporte a la traducción bi-direccional entre UML y el lenguaje específico del dominio de FlexAB. Dicha herramienta se basó en los frameworks más apropiados y en los estándares existentes en el área.

Se ha desarrollado un software complejo que logra capturar la transformación bi-direccional (en parte) de ambos lenguajes (FlexAB y UML).
El resultado es una herramienta interesante que permite cambiar el enfoque del modelador de acuerdo a sus preferencias y/o habilidades. También, las transformaciones permiten la migración de modelos escritos en FlexAB para poder presentárselos a modeladores que no conocen este lenguaje y si poseen conocimiento de UML. En ciertas ocasiones, es beneficioso tener el modelo en UML porque muchas de las herramientas utilizadas para el modelado soportan el lenguaje. Una persona que con conocimientos en UML podría trabajar con el modelo en dicho lenguaje y una vez realizadas las modificaciones se puede transformar a FlexAB para que un usuario con conocimientos en dicho modelo, y no tanto en UML, pueda trabajar con su respectivo modelo.

10 Trabajos futuros

En base a los conceptos planteados a lo largo de esta tesis y a los resultados obtenidos, se proponen las siguientes líneas como trabajos futuros:

- Completar el análisis de correspondencia de las clases de FlexAB faltantes con sus correspondientes en UML.
- Extender la herramienta para las clases que quedaron sin mapear.
- Mejorar usabilidad de la herramienta implementada.
- Adaptar alguna herramienta para la instanciación de las clases de los distintos modelos.

11 Bibliografía

1. MDA Guide, v1. 0. 1, omg/03-06-01, June 2003. http://www. omg. org.

2. Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

3. Stahl, T. and Völter, M. Model-Driven Software Development. John Wiley & Sons, Ltd. (2006)

4. Domain Specific Modeling: Enabling Full Code Generation. Kelly, Steven; Tolvanen, Juha-Pekka. 2008.

5. Czarnecki, Helsen. Feature-based survey of model transformation approaches.IBM System Journal,V.45,N3, 2006.

6. Object Management Group (OMG) http://www.omg.org

7. Meta Object Facility (MOF) 2.0. OMG Adopted Specification. October, 2003. http://www.omg.org.

8. Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/

9. Graphical Modeling Framework (GMF) - www.eclipse.org/gmf/

10. UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification. August 2003. http://www.omg.org.

11. FlexAB - Flexible Application Builder® http://www.eledisa.com.ar/

12. Jack Greenfield, Keith Short, Steve Cook, Stuart Kent. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley, 2004. ISBN: 0471202843.

13. Query/View/Transformation (QVT) Specification. Final Adopted Specification ptc/07-07-07. OMG. (2007)