Apéndice

DMOF y MOF

DMOF y MOF

DMOF es la implementación realizada por el DSTC (Distributed Systems Technology Centre) de las especificaciones MOF y XMI de la OMG http://www.dstc.edu.au/Products/CORBA/MOF/. El producto dMOF consta de una suite de tools, que se enumeran a continuación:

- Repositorio del meta modelo de DMOF. Es una implementación de la especificación del repositorio de meta modelo de MOF de OMG.
- Compilador de MODL. Este carga meta modelos MOF (especificados en lenguaje MODL) en el repositorio del meta modelo.
- Herramientas de intercambio de meta modelo MOF. Permiten intercambiar meta modelos MOF codificados en XMI.
- El generador de Moflet. Produce moflet: código java que implementa los meta-objetos del repositorio que representa una meta data del usuario.
- Generadores de XMI. Produce un DTD y software de codificación y descodificación para intercambio de meta data de usuarios via XMI.

Para generar un manejador de repositorio a partir de la especificación de un metamodelo con dMOF existen dos formatos posibles de "entrada":

- La especificación del meta modelo en el lenguaje MODL
- La especificación del meta modelo en XMI

Supongamos que tenemos un archivo Modl o un documento XMI para un meta modelo trivial: **package Trivial {}**, los comandos a ejecutar para generar el manejador de repositorio son los siguientes:

1. Si tenemos el MODL del meta modelo , entonces compilarlo con el comando **modl2mof -m $IOR_DIR/dmof.ior Trivial.modl**
2. Si tenemos el documento XMI, cargarlo al repositorio con el comando xmi2mof **-m $IOR_DIR/dmof.ior Trivial.xml**
3. Ahora se tiene el meta modelo "Trivial" cargado en el repositorio.
4. Generar la interfaz en IDL a partir del repositorio con el comando: **mof2idl -x -m $IOR_DIR/dmof.ior -p Trivial > Trivial.idl**
5. Generar los stubs y skeletons para java que implementar el IDL (denominados moflets) con el comando: **mof2moflet -x -m $IOR_DIR/dmof.ior -p Trivial**
6. Las clases generadas en este paso son generadas bajo el directorio TrivialImpl.
7. Compilar el IDL generado a codigo java, usando idl2java de Visibroker, con el comando: **idl2java -I/opt/dmof/idl -C -no_examples Trivial.idl**
8. Compilar las clases bajo TrivialImpl con: **vbjc TrivialImpl/*.java**
9. Levantar el repositorio generado con el comando **vbj -Ddmof.license.dat=/opt/dMOF/License.txt TrivialImpl.TrivialServer -f dmof.ior –new**
10. De esa forma se crea una nueva instancia del repositorio de meta data y salva esta en el archivo llamado dmof.ior.

A continuación se detalla este proceso explicando que significa cada paso y aplicado a un ejemplo en MODL , pero el proceso es similar si se trata del XMI de un meta modelo, la variación esta solo en el primer paso.

Proceso de desarrollo para generar repositorios de meta data basados en MOF.

El primer paso es definir el meta modelo en MOF (si ya no se tiene el mismo). Por ejemplo se define un diagrama de un meta modelo usando UML.

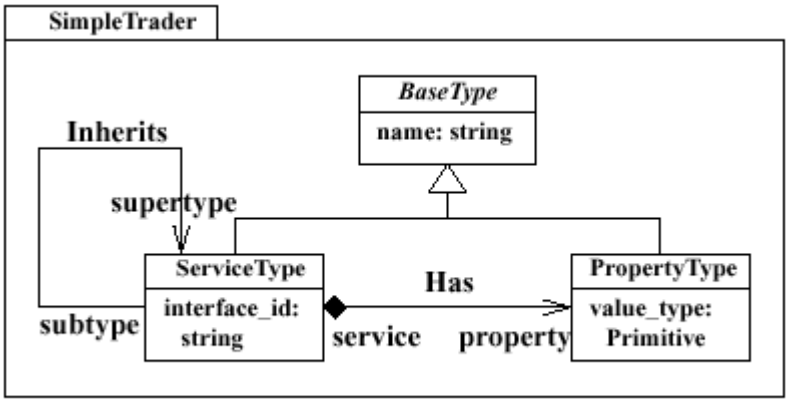


Figure 3: The SimpleTrader Meta-model in UML

Este meta modelo define un sistema de tipo simple para un objeto “vendedor” (service trader). Tiene dos clases de tipos: type service, property type. Están expresados como clases:

- La clase **ServiceType** que tiene atributos para el nombre del servicio, un identificador para un tipo de interfase externa para una instancia del servicio.
- La clase **PropertyType** representando un nombre de propiedad y un tipo simple de valor correspondiente.

Las clases están relacionadas por associations llamadas inherits y has:

- Los links en las **inherits association** definen una jerarquía de herencia (esto permite polimorfismo).
- Los links en las has **association** asocian servicios a conjuntos de propiedades especificadas como pares de nombre/valor. (Esto permite el matcheo de servicio basado en propiedades).

El meta modelo de la figura puede ser expresado en MODL como:

```
package SimpleTrader {
enum PrimitiveType {pt_bool, pt_int, pt_float};
abstract class BaseType {
attribute string name;
};
class PropertyType : BaseType {
attribute PrimitiveType value_type;
};
class ServiceType : BaseType {
attribute string interface_id;
reference supers to supertype of Inherits;
reference props to property of Has;
};
association Has {
end single ServiceType service;
composite end set [0..*] of PropertyType property;
};
association Inherits {
end set [0..*] of ServiceType supertype;
end set [0..*] of ServiceType subtype;
};
};
```

El siguiente paso es generar el repositorio que puede manejar la meta data de acuerdo a los meta modelos. Este proceso se esquematiza en la siguiente figura:

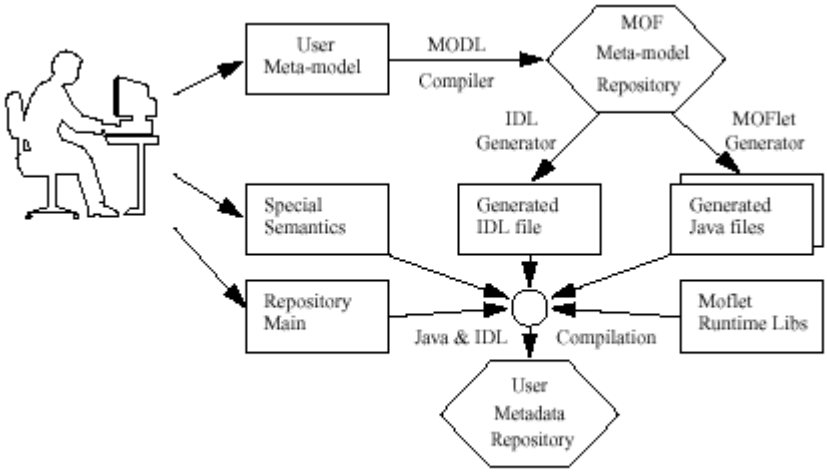


Figure 4: Generating a MOF Metadata Repository

Con el compilador modl2mof se compilan los meta modelos y son guardados en el repositorio de meta modelo MOF.

Luego se utiliza el generador IDL mof2idl para producir el IDL CORBA para un repositorio de meta data. Estos IDL generados permiten a un programa cliente basado en CORBA, hacer updates, crear, y acceder a la meta data en el repositorio. Estos IDL generados están de acuerdo a el mapeo estándar de IDL-MOF.

Posteriormente se utiliza el generador de Moflet mof2moflet para producir código java que implementa la funcionalidad del repositorio de meta data MOF. Este código moflet implementa la semántica definida del estándar de mapeo MOF a IDL.

Luego se debe implementar semántica especial para el moflet generado. Esto solo es necesario si el meta modelo utiliza características avanzadas tales como operaciones, atributos derivados o asociaciones derivadas. Y el programa servidor (main) o equivalente.

Cuando el repositorio de meta data ha sido implementado, el desarrollador necesita implementar las herramientas de meta data necesarias para el sistema.

Estas pueden incluir herramientas de input y output: como compiladores, pretty printers, editores de diagrama y/o herramientas de intercambio de meta data: como herramientas para upload y download DTDs de XMI.

Finalmente el desarrollador necesita implementar los programas de aplicación y componentes que hacen uso de la meta data.

DMOF implementa los mapeos posibles de MOF

La siguiente figura resume los mapeos que se pueden obtener a partir de un meta modelo en MOF (esta también mencionado en el anexo de meta data).

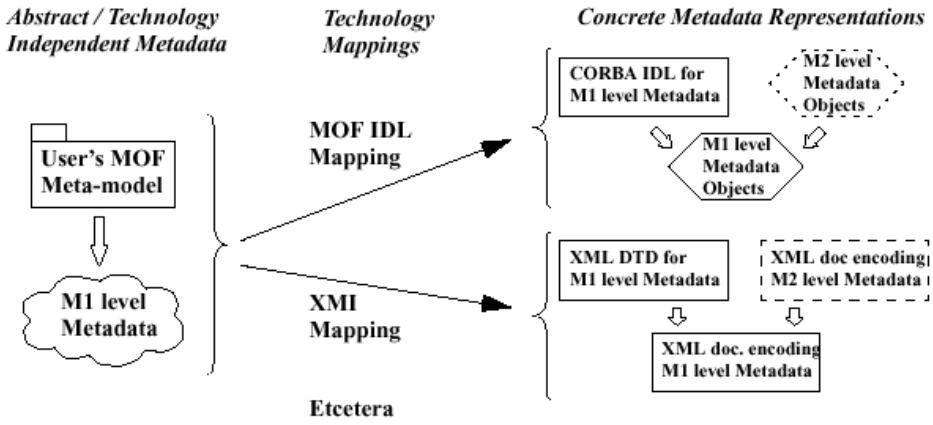


Figure 5: Standard MOF Technology Mappings

La OMG tiene estandarizadas formalmente dos tecnologías de mapeos de datos. Estas están ilustradas en la figura anterior y son:

MOF IDL Mapping: mapea un meta modelo MOF a un IDL CORBA y semántica asociada para un servicio específico del depósito del metadata del meta-modelo CORBA.

XMI: mapea un meta modelo MOF a un DTD de XML y un conjunto asociado de reglas de producción de documento XML. XMI permite el intercambio de cualquier tipo de meta data MOF (incluyendo meta modelos MOF) entre repositorios de meta data.

MOF IDL Mapping

El **MOF IDL Mapping** es un estándar que mapea un metamodelo MOF en un conjunto de interfaces CORBA IDL. Si la entrada para el mapeo es el metamodelo para una clase dada de metadata, la resultante interfaz IDL son objetos CORBA que pueden representar esa metadata. La IDL mapeada son usados en un repositorio para almacenar la metadata.

La principal correspondencia entre elementos en un metamodelo MOF (M2) y los objetos CORBA que representa metadata (M1) son:

Una Clase en el metamodelo es mapeada sobre una interfaz IDL para objetos de la metadata y una metadata class proxy. Estas interfaces soportan las Operaciones, Atributos y Referencias definidas en el metamodelo.

Una Asociación es mapeada sobre una interfaz para una metadata package proxy

Objetos de la metadata MOF comparten un conjunto común de interfaces. Estas interfaces permiten a un programa cliente genérico acceder y actualizar la metadata sin ser compilado cada vez que el metamodelo es generado. El propósito es definir interfaces IDL CORBA para un servicio de administración de meta data que soporte el input de meta model. Proporciona APIs CORBA que permiten a un cliente basado en CORBA, crear, acceder y actualizar la meta data en un repositorio de meta data basado en CORBA. Esta diseñado para satisfacer los requerimientos como:

- Meta data representada como objetos, es decir meta objetos

- Ciclo de vida completo de los objetos, es decir: creación, actualizaciones, borrados.
- Soporte a modos de consulta y navegación de acceso a meta data
- Soporte de creación de meta data por ambos: “compilación” y “edición”

Las interfaces IDL generadas tienen una jerarquía de herencia que sigue el patrón de la figura siguiente, la notación <XxxName> indica que el nombre de la interfase IDL es derivado del nombre del elemento en el meta modelo, es decir algún package, class o association.

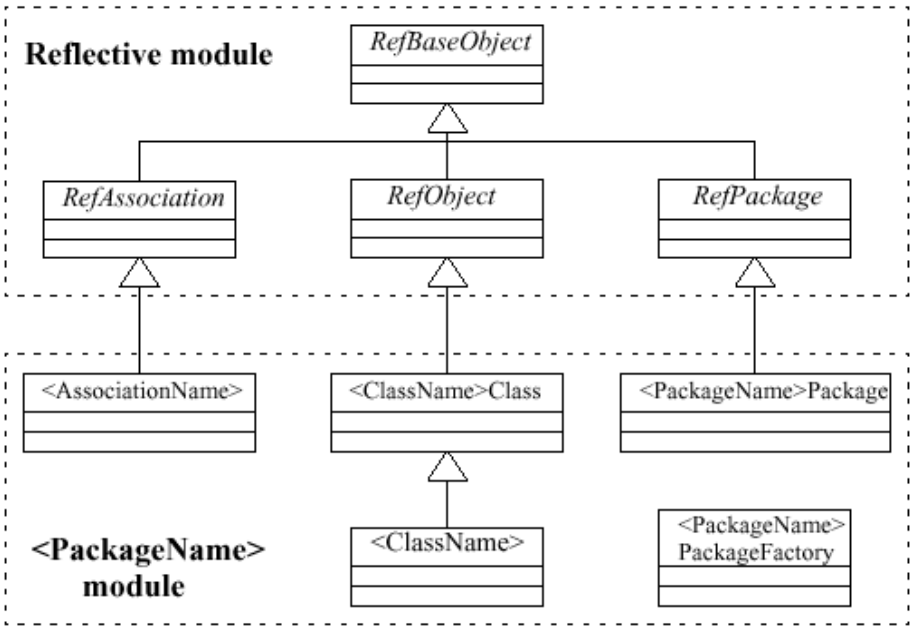


Figure 6: The Inheritance Hierarchy for MOF IDL

Los IDL generados para un meta modelo dado tienen dos partes:

El modulo “reflector” fijo, que contiene un conjunto de interfaces base que son aplicables a todos los meta modelos. Estos proporcionan las APIs genéricas de la meta data junto con un pequeño numero de operaciones house-keeping.

Un modulo “especifico” , que contiene APIs “user-friendly” para el meta modelo.

Como lo muestra la figura , existen 5 tipos distintos de interfaces en un modulo <packageName> generado:

<ClassName> Las instancias de una interface <ClassName> son meta objetos (objetos de instancia) que representan nodos de la metadata.

<ClassName>Class. Una instancia de <ClassName>Class es un meta objeto que contiene el estado y comportamiento asociado con una Class de nodos de la meta data en lugar de un nodo individual.

<AssociationName>. Una instancia de <AssociationName> es un meta objeto que maneja el conjunto de links que pertencen a una magnitud

<PackageName>Package es un meta objeto que maneja la magnitud de un package, es decir, esto es un contenedor para nodos y links de la meta data.

<PackageName>PackageFactory. Esta interfase es usada para crear objetos package.

Las APIs que mencionamos, pueden ser usadas para intercambio entre repositorios, pero no es un propósito ideal debido a que no soportan intercambio de meta data con otros repositorios de meta data NO-CORBA y se asume un acceso on-line, lo cual impide el intercambio de meta data via e-mail, disquete o similares.

MOF XMI Mapping

Una solución alternativa a estas limitaciones de las IDLs es el mapeo a XMI. El propósito de XMI es permitir el intercambio de modelos en una forma serializada. El mapeo XMI es en esencia un estándar para codificar colecciones de meta data basadas en MOF como documentos XML. Haciendo de esta forma intercambio independiente del middleware utilizado. El mapeo XMI consiste en 2 partes:

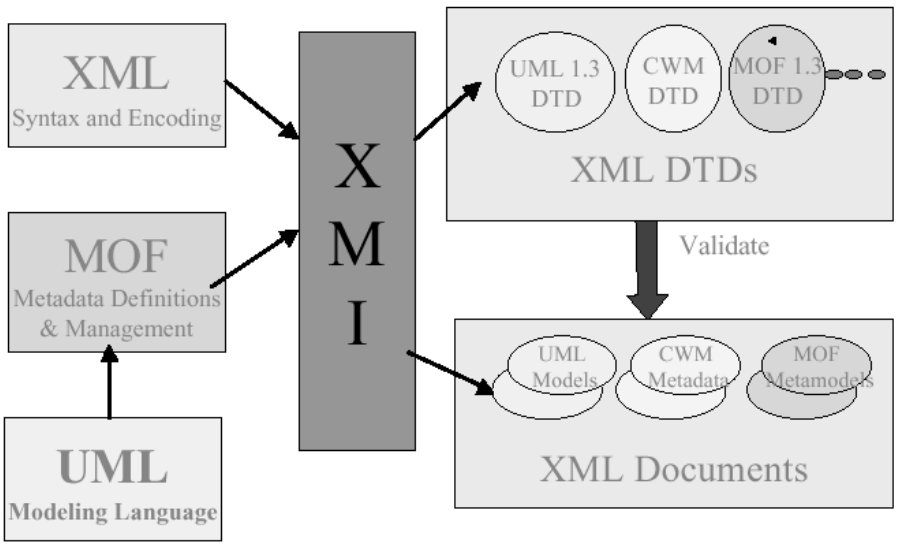
Reglas de producción del DTD: reglas que definen cómo un meta modelo basado en MOF puede ser traducido en un DTD de XML.

Reglas de producción del XML: definen cómo una meta data basada en MOF que conforma a un meta modelo dado es codificado como un documento XML que se adapta al DTD.

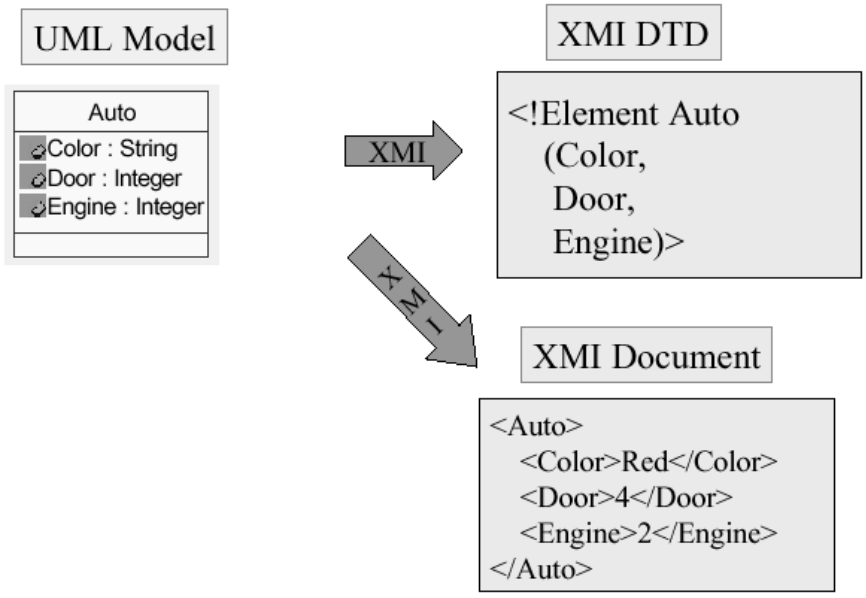
Las reglas de Producción de XML DTD para producir definiciones de tipos de documentos XML (DTDs) para XMI codifica metadata. XMI DTDs sirve para especificar sintaxis para documentos XMI, y permite usar herramientas genéricas de XML para componer y validar documentos XMI.

Las reglas de producción pueden ser aplicadas en reversa para decodificar documentos XMI y reconstruir la metadata.

El mapeo entre MOF y XMI se esquematiza en la siguiente figura:



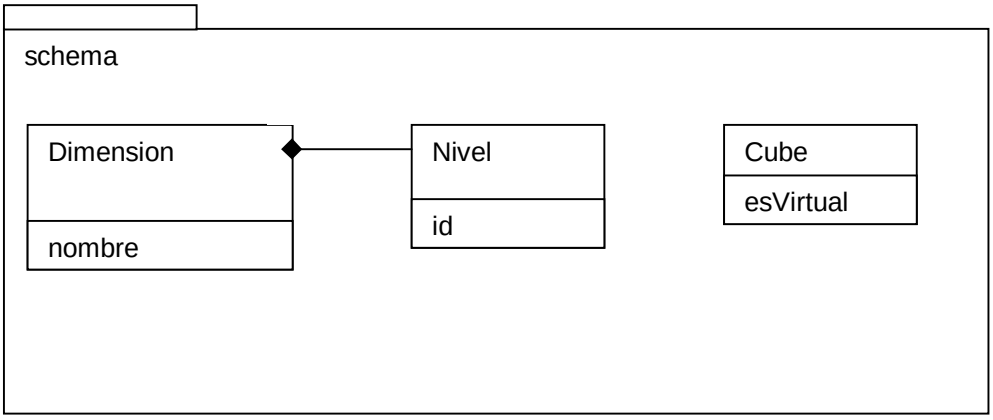
Un ejemplo en concreto de ese mapeo se puede ver con la siguiente figura:



Un ejemplo mas completo de utilización de dMOF

Habiendo detallado los posibles mapeos de meta modelos en MOF, a continuación se da un nuevo ejemplo de utilización del producto dMOF, esta vez mostrando como se pueden obtener los documentos XMI de instancias de meta modelo cargadas en un repositorio, y como hacer persistentes los modelos almacenados en el repositorio.

Supongamos que tenemos el siguiente meta modelo especificado en UML .



Y la siguiente especificacion en MODL equivalente:

```
package schema {
        class Dimension {
                attribute string nombre;
        };
        class Cube{
                attribute boolean esVirtual;
        };
        class Nivel{
                attribute string id;
```

```
        };
        class Has {
                end single Dimension d;
                composite end set [0..*] of Nivel n;
        };
};
```

Al igual que en el ejemplo anterior el primer paso es usar el compilador de MODL para compilar el metamodelo y cargarlo en el repositorio.

modl2mof -m dmof.ior TradingRepos.modl -m dmof.ior

Con este comando se indica que el archivo IOR para el repositorio de meta modelo se llama dmof.ior. Esto lleva a cabo chequeos de sintaxis sobre el archivo MODL, y crea nuevos meta objetos que representan el meta modelo en el repositorio. Luego se genera la IDL

call mof2idl.bat -x -m dmof.ior -A -p TradingRepos -f TradingRepos.idl

Este comando genera un archivo IDL CORBA que define las interfaces para el repositorio de meta data para el meta modelo TradingRepos.
-x Indica que se genera un IDL para las extensiones propietarias para el estándar de MOF IDL Mapping.
-m Provee el nombre del archivo IOR para el repositorio de meta modelo dMOF. El IOR en el archivo debe corresponder a una instancia del repositorio accesible. Este argumento es obligatorio.
-A –p Provee el nombre del paquete para el meta modelo para el cual el IDL sera generado. Este nombre de paquete debe haber sido previamente cargado en el repositorio de meta modelo dMOF. Este argumento es obligatorio.
-f Provee el nombre del archivo generado IDL. Si esta opción no esta, el IDL es escrito a la salida estándar.

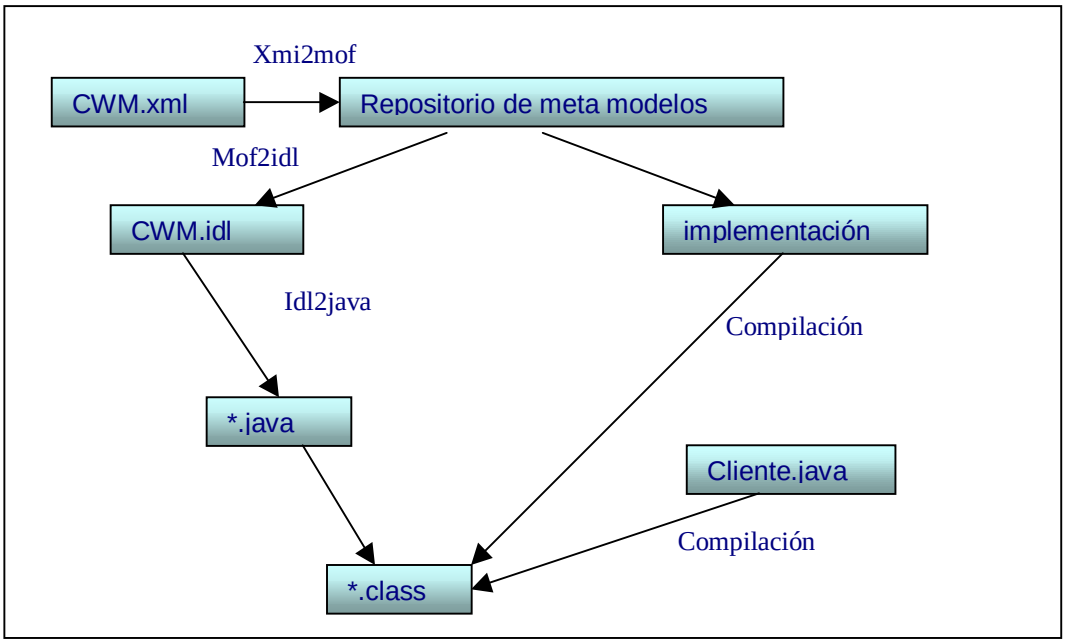
call mof2moflet -x -m dmof.ior -p TradingRepos
Este comando genera una colección de archivos java que (conjuntamente con las librerías de runtime de dMOF) implementan el IDL. Estos archivos serán escritos en un subdirectorio llamado TradingRepos.

idl2java -I"C:\Program Files\dMOF_1_1\idl" -C -no_comments -no_examples TradingRepos.idl
Este comando compila el IDL generado.
-C Indica que cualquier comentario en el IDL sea llevado dentro de los stubs y skeletons de manera apropiada.
-I Setea el path de búsqueda de interfase del compilador IDL para incluir el core de los archivos IDL de MOF.
-no_examples Suprime la generación de clases ejemplo.
Luego se compilan los archivos Java moflet generados en el directorio TradingRepos con: **vbjc TradingRepos*.java.**Y el server.java con **vbjc TradingReposImpl\TradingReposServer.java**
Compilar luego el cliente con: **vbjc TradingReposClient.java**

Y se levanta el manejador en forma pesistente con el comando:
vbj -VBJprop dmof.license.dat="C:\Program Files\dMOF_1_1\License.txt" TradingReposImpl.TradingReposServer -c jdbc.config -f TradingRepos.ior -P 1234

De esta forma se crea una instancia persistente del moflet corriendo en el puerto TCP/IP 1234. Para mas detalle ver tutorial de dMOF.
Si no esta el comando –new entonces restartea la instancia del moflet.

Para el caso de CWM, del cual teníamos su especificación en XMI el proceso de generación del manejador de repositorio fue muy similar , y se esquematiza en la siguiente figura:



Cliente que envía un request a una implementación de un objeto