Universidad de Granada

“Técnicas Avanzadas de Modelado de Sistemas de Control y Telecomunicaciones” (2ª parte)

Manuel I. Capel Tuñón
Software Engineering Department,
Concurrent Systems Group,
University of Granada, Spain.
manuelcapel@ugr.es

Índice

- Revisión de MOF/QVT: objetivos, organización e introducción al lenguaje
- El metamodelo de las notaciones QVT: especificaciones declarativas e imperativas
- Transformaciones en MDA
- QVT – Nuclear
- Lenguaje de Relaciones QVT
- Casos de estudio

Revisión de MOF/QVT

Actualmente existe un nuevo estándar: QVT = "Query-View-Transformations", que da respuesta a la necesidad de transformaciones entre modelos cuyos lenguajes están definidos con MOF

•Query (Consulta): es una expresión evaluada sobre un modelo dado que tiene como resultado 1 ó más instancias de tipos (del *modelo fuente* o en el *lenguaje de consulta*):

–Listar todos los paquetes que no contengan paquetes-hijos

–¿Posee el atributo A del modelo fuente visibilidad pública?

•View (Vista): es un modelo derivado completamente de otro modelo. Son más generales que las *Consultas*. Suelen ser entidades de sólo lectura.

•Transformation: genera un *modelo destino* desde un *modelo fuente*. Las *relaciones* especifican las transformaciones y las *correspondencias* (mappings) las implementan.

•*Requisitos de las transformaciones*


Objetivos de QVT

- Requisitos obligatorios:
  - Lenguaje de Consulta
  - Lenguaje de Transformación del lenguaje *fuente* (F) al lenguaje *destino* (D)
  - Definido con metamodelos MOF
  - Ejecutable / Transportable *versus* Caja-Negra XForms
  - Definición de vistas
  - Transformaciones *declarativas / incrementales*

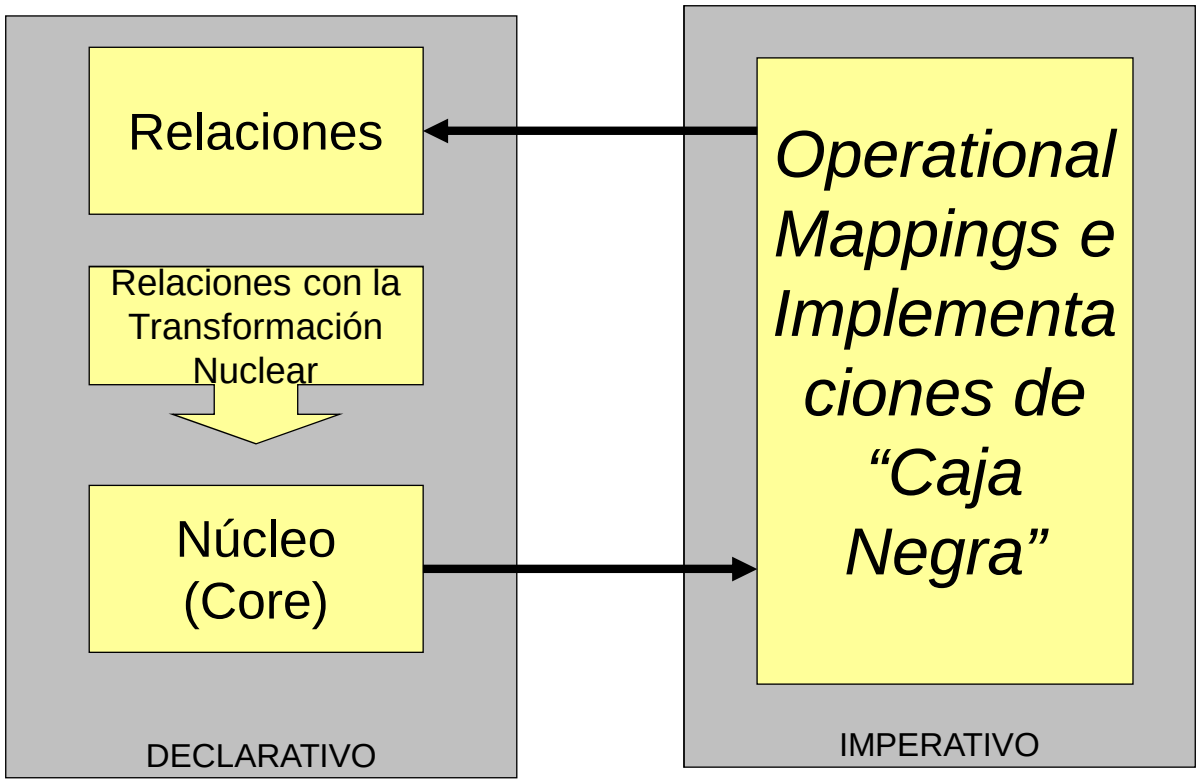

Objetivos de QVT (2)

- Requisitos opcionales:
  - Transformaciones bidireccionales
  - Registro de las transformaciones
  - Reutilización y extensión de transformaciones genéricas
  - Las transformaciones con semántica de *transacciones: commit*, *rollback*
  - Transformaciones dentro de un Lenguaje específico


Organización de QVT

- QVT: naturaleza de lenguaje declarativo e imperativo a la vez
- QVT *declarativo*
  1. *Metamodelo de* Relaciones de uso amistoso, *concordancia de patrones*, *creación de plantillas de objetos*
  2. *Metamodelo Nuclear* (Core Metamodel), *expresiones escritas con* EMOF y OCL
- QVT imperativo
  - *MAPPINGS,* es la forma estándar de proporcionar implementaciones imperativas para trasladar los modelos QVT
  - *CAJA-NEGRA*, permiten *enchufar* cualquier implementación de una operación MOF en un modelo, siempre que tenga la misma signatura que la operación especificada con el lenguaje *Relations*

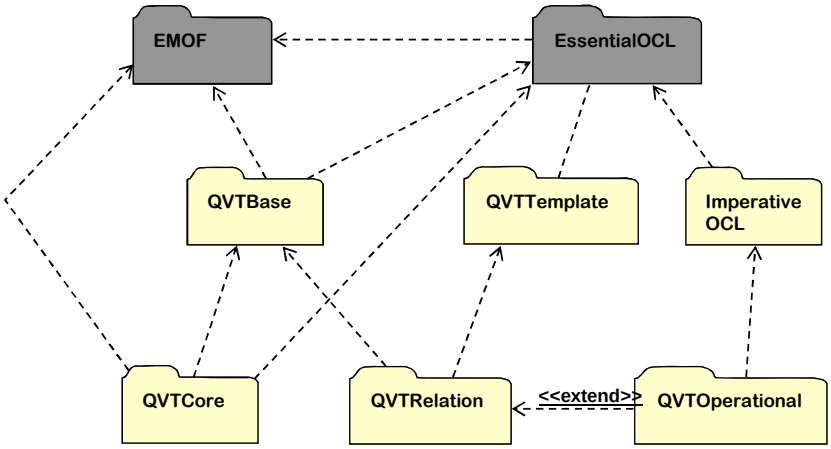

Introducción al lenguaje



Metamodelo de notaciones QVT

- Los metamodelos MOF se pueden describir mediante los lenguajes definidos en los siguientes *paquetes*QVT:
  - QVT Core (Lenguaje Nuclear)
  - QVT Relation (Lenguaje de Relaciones)
  - QVT Operational (Lenguaje Imperativo)

Dependencias de paquetes en la especificación QVT



Relaciones (especificaciones declarativas)

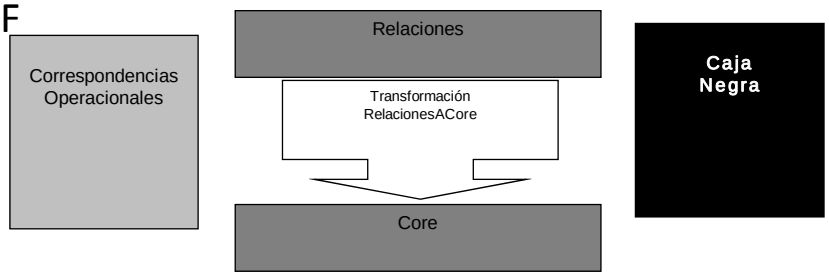
- Relaciones: especificación declarativa de las relaciones entre modelos MOF
- Ofrece:
  - Concordancia de patrones de objetos complejos
  - Creación implícita de *clases de registro* y sus instancias (para registrar los que ocurre durante la ejecución de una transformación)
- Las relaciones pueden afirmar que también se cumplen otras relaciones entre elementos particulares del modelo que se localizan mediante concordancia con sus patrones


Núcleo (especificaciones declarativas)

- Núcleo : Lenguaje/Modelo pequeño
  - Sólo soporta concordancia de patrones con respecto a un *conjunto plano* de variables
  - mediante la evaluación de condiciones definidas con esas variables y respecto de un conjunto de modelos
- Trata todos los elementos del modelo (*fuente*, *destino* y *registro*) simétricamente
- Tiene una potencia equivalente a la del *lenguaje de relaciones*, pero su definición semántica es más simple
- Máquina virtual para ejecutar las transformaciones: se podría traer la analogía entre lenguaje QVT Nuclear y el lenguaje de *byte codes* de Java:
  - Semántica de QVT Nuclear → Especificación comportamiento de JVM
  - Lenguaje QVT de Relaciones → Lenguaje Java
  - Transformación (relaciones/núcleo) → compilador Java-Bytecodes


Registro de correspondencias QVT

- En el nivel de **Relaciones** el *registro de las correspondencias* entre los modelos de una transformación es creado implícitamente
- En el nivel **Nuclear** el *registro* se describe mediante modelos MOF



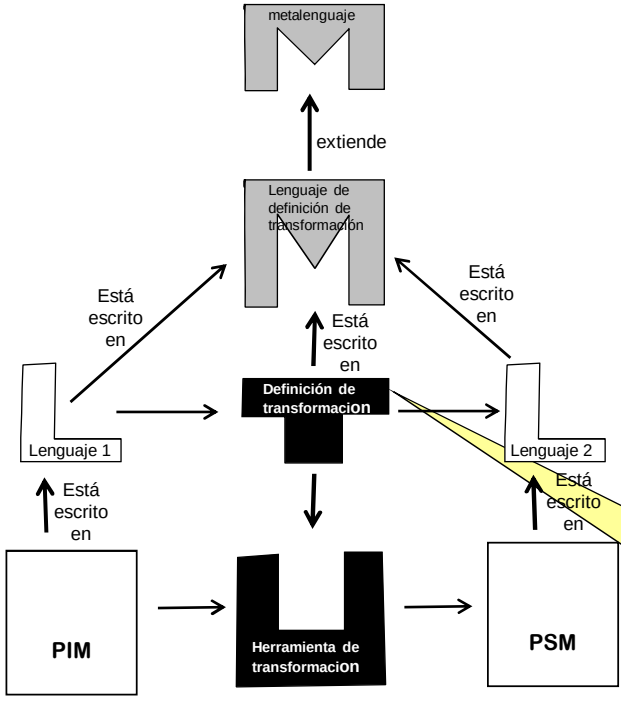
Mappings (especificaciones imperativas)

- Lenguaje estándar: *Operational Mappings*
- *Proporcionan:*
  - Extensiones en OCL con efectos laterales que permiten un estilo más procedural
  - Las expresiones de este lenguaje son *transformaciones operacionales*
  - Una sintaxis concreta que resulta familiar a los programadores de lenguajes *imperativos* (C++, Java, etc.)
- Utilización:
  - Para implementar 1 ó más relaciones, a partir de una especificación de relaciones, que no puede ser definida de forma *totalmente declarativa*
  - Cuando resulta difícil dar una especificación puramente declarativa acerca de cómo se *trasladaría* una *relación.*

“Caja- Negra” (especificaciones imperativas)

- Implementaciones no-estándar de transformaciones: *Black-box MOF Operation*
- El *Lenguaje de Relaciones QVT* hace posible ensamblar cualquier implementación a una Operación MOF si se mantiene la signatura de dicha operación
- Objetivos:
  - Permite la codificación de algoritmos complejos en cualquier lenguaje de programación que tenga un *puente* a MOF (o que pueda ser ejecutado dentro de un lenguaje que lo tenga)
  - Permite la utilización de bibliotecas específicas de dominio para calcular los valores de las propiedades de un modelo (en lugar de OCL)
  - Permite que las implementaciones de algunas partes de una transformación sean opacas
  - Cada *caja negra* debe implementar explícitamente 1 relación (responsable de mantener el *registro* entre elementos del modelo)

Transformaciones



- Falta de *interoperabilidad* entre las tecnologías de desarrollo de software actuales
- Falta de facilidades de evaluación frente al *desempeño*
- Evolución tecnológica difícil de gestionar(*obsolescencia*)
- Independencia de la *lógica de negocio* respecto de las tecnologías que la soportan

Las **transformaciones en MDA**, como ahora se verá, dan respuesta estas necesidades de los usuarios de tecnologías.

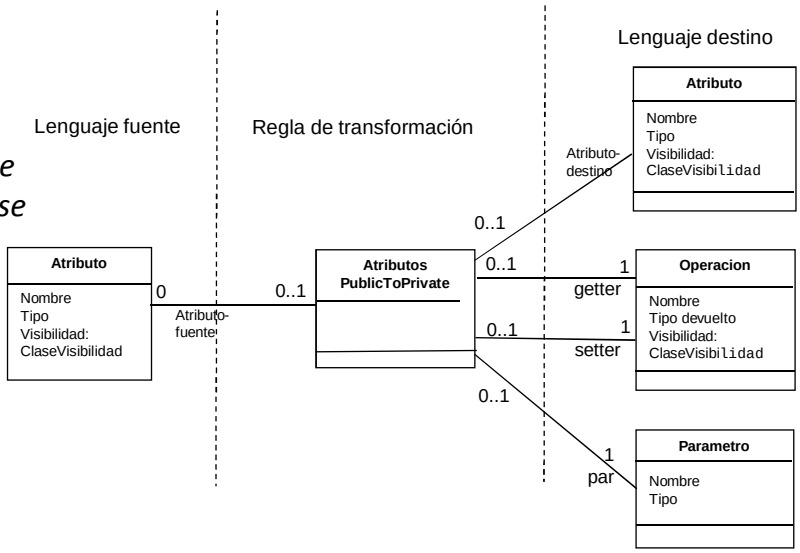
Transformaciones (I)

- Diferenciar entre *definición* y *aplicación* de una transformación
- Las *transformaciones* pueden ser aplicadas automáticamente, independiente del modelo de entrada, por una herramienta

Transformación	Generación automática de un modelo objetivo desde un modelo de entrada (o fuente), de acuerdo con una definición de transformación
Definición de transformación	**Conjunto de reglas de transformación que todas juntas describen cómo un modelo en el *lenguaje fuente* puede ser transformado en un modelo en el lenguaje *objetivo***
Regla de transformación	**Una regla de transformación es una descripción de cómo una o más construcciones del *lenguaje fuente* pueden ser transformadas en una o más construcciones del *lenguaje objetivo***

Definición de transformaciones

- (i) Relacionar cada uno de los elementos del modelo fuente con uno o más elementos del modelo destino:
  - **Identificar** la *metaclase* del elemento en el modelo *fuente*
  - **Corresponder** con la *metaclase* del elemento en *destino*
  - **Asegurar** que se mantiene la consistencia de las reglas establecidas en el modelo fuente
- MDA apoya la consistencia semántica de lo que se transforma mediante unas **reglas de transformación**



Reglas de transformación MDA

- Incluidas en MDA, se utilizan informalmente:

i. Para cada clase denominada *nombreClase* en el PIM, existe una clase denominada de la misma manera en el PSM

ii. Para cada atributo público denominado *nombreAtributo* : *Tipo* de la clase *nombreClase* en el PIM, existen, como parte de la clase *nombreClase*, una serie de atributos y operaciones en el modelo de

a. Un atributo p… …e: nombreAtribut…
b. Una operación… …el nombre de atrib… …bo de …ibuto … …o: …ibuto() : Tipo

iii. Una operació… …a con el nombre de atributo precedido con… …buto como valor y sin devolver ningún valor: *setNombreAtributo(at: Tipo)*

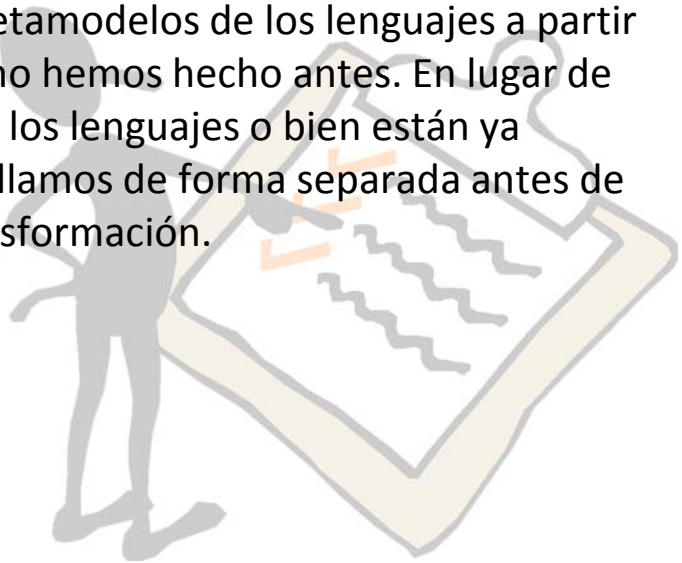
Meta-atributo "visibilidad"

Meta-atributos

Metaclase del PSM

De forma práctica…

- Nunca se deducen los metamodelos de los lenguajes a partir de las descripciones, como hemos hecho antes. En lugar de eso, los metamodelos de los lenguajes o bien están ya disponibles o los desarrollamos de forma separada antes de escribir las reglas de transformación.



Lenguaje de definición de transformaciones

- Una regla de transformación debe contener la siguiente información:
  i. El **lenguaje fuente** de referencia (conjunto de elementos S)
  ii. El **lenguaje destino** de referencia (conjunto T)
  iii. Los **parámetros** de transformación (optativos)
  iv. Un **indicador direccional** (booleano), par establecer si el modelo *fuente* puede ser regenerado en el lenguaje *destino*
  v. Condiciones del lenguaje fuente (*invariantes*)
  vi. Reglas de **mapping** (*correspondencia)*


Primera notación para las reglas

- Posible formalización general de las reglas de transformación

Transformation <*Nombre*> **(**<*L. Fuente*>**,** <*L.Destino*>**) {**
params
{<Nombre>: **String = '**<Nombre>**';}***
target
{<Nombre> **:** <Nombre::cualificado>**;}***
target condition
//expresiones booleanas OCL que hacen referencia a los //nombres cualificados anteriores; incluyen *meta-operadores*
mapping
<Nombre(en 'source')> **<~>** <Nombre(en 'target')>
(**unidirectional | bidirectional);**


Ejemplo de transformación

```
Transformation ClaseTabla (UML, SQL){
  params
    prefijosetter: String = 'set';
    prefijogetter: String = 'get';

  target
    c: SQL::Columna;
    f: SQL::TeclaExtranjera;

  target condition
    f.valor = c and c.tipo = f.refersTo.valor.tipo;

  mapping
    c.nombre <~> t.nombre;

  unidireccional;

  mapping
    clases -> forAll (c | tablas -> exists (t | c <~> t);
}
```

A veces, sin embargo, las definiciones de transformación se definen de una manera más útil mediante la aplicación de otras definiciones de transformación que se ejecutan en secuencia. Una definición de sub-transformación tiene como resultado (al menos conceptualmente) un modelo intermedio


Transformación de atributos públicos y privados

- La segunda regla especifica una transformación de atributos *públicos* a *privados y* la especificación de las operaciones *getter* & *setter.*
- Se podría ver como una clase de relación entre las metaclases de los lenguajes PIM y PSM.
- Se utilizarán *roles* (*atributo-fuente*, *atributo-destino*) en las asociaciones
- Lo anteriormente especificado no es lo suficientemente preciso para ser considerado *regla de transformación*
- *Resulta necesario* aumentar lo especificado con parámetros, condiciones y demás elementos


Ejemplo de transformación (2)

La definición de la transformación completa para hacer corresponder un modelo que posee atributos públicos a un modelo que sólo tenga atributos privados necesita un número mayor de reglas

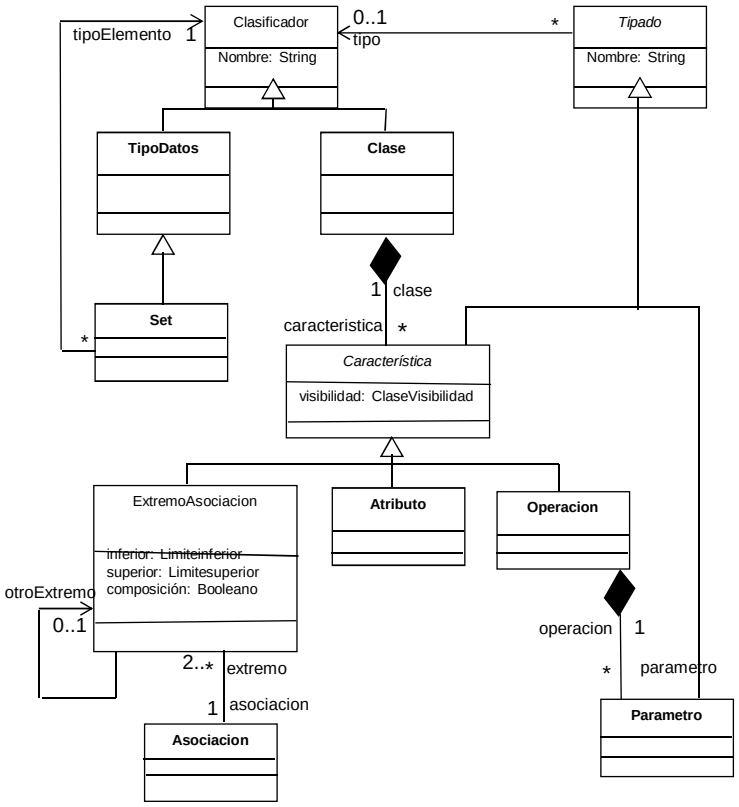
La regla se puede ver como una clase de relación entre las metaclases de los lenguajes PIM y PSM de la infraestructura MDA:

```
Transformation PublicToPrivateAttri
    params
        setterprefix: String = 'set';
        getterprefix: String = 'get';
    source
        sourceAttribute : UML::Attribute;
    target
      targetAttribute : UML::Attribute;
            setter : UML::Operation;
            getter : UML::Operation;
    rce condition
        sourceAttribute.visibility = VisibilityKind::publi
    t condition
        target.Attribute.visibility = visibilityKind::priva
        setter.name = setterprefix.concat (targetAttrib
        setter.parameters -> exists ( p |
                                p.name = target.Attrib
                                p.type = target.Attribute.t
        setter.type = OclVoid

        getter.name = getterprefix.concat (targetAttribute.name) and
        getter.parameters -> isEmpty()                            and
        getter.type = targetAttribute.class
    nd
        target.Attribute.class = setter.class                     and
        target.Attribute.class = getter.class                and
    bidirectional;
    mapping
        sourceAttribute.name <~> target.Attribute.name;
        sourceAttribute.type <~> target.Attribute.type;

    mapping
        sourceAttribute.name <~> targetAttribute.name;
        sourceAttribute.type <~> targetAttribute.type;
}
```


Ejemplo (2)

Las nuevas reglas simplemente tomarían un elemento (diferente de un atributo público) en el modelo fuente y producirían exactamente el mismo elemento en el modelo destino y no están, por tanto, formalizadas.

El metamodelo simplificado de UML



Transformación de asociaciones

- Los extremos de una asociación se convierten en atributos privados con operaciones ***get*** y ***set***.
- Se combina el uso de un modelo intermedio y facilidad para combinar definiciones de transformaciones.
- No se pueden definir reglas de correspondencia condicionalmente, tenemos que escribir dos reglas:

```
Transformation Ma...
        params                  -
        source
                ae : UML::AssociationEnd;
        target
                att : UML::Attribute;
        source condition
                ae.upper >= 1;
        target condition
                att.visibility = VisibilityKind::pu
                att.type.isTypeOf(Set);
        unidireccional;
        mapping
                ae.name <> att.name;
                ae.type <> att.type.elementTyp
}

    Transformation SimpleAssociationToAttribute (UML,UML);
          params              -- ninguno
          source
                ae : UML::AssociationEnd;
          target
                att : UML::Attribute;
          source condition
                ae.upper <= 1;
          target condition
                att.visibility = VisibilityKind::public     and
                att.type.isTypeOf(Class);
          unidireccional;
          mapping
                ae.name <> att.name;
                ae.type <> att.type.elementType;
    }
```

Podemos, por tanto, o bien construir una definición de transformación que especifique la transformación completa, o podemos definir una que transforme los extremos de una asociación en atributos públicos y, después, aplicar la regla de transformación de los atributos públicos a privados.


Transformación de clases

- Queremos es definir una transformación de clases a clases, donde los extremos de las asociaciones y los atributos son hechos corresponder como se describe:
  - Para cada atributo público llamado *nombreAtributo: Tipo* de la clase *Nombre_clase* en el PIM, se definen atributos y operaciones para la clase del mismo nombre en el destino
  - Para cada extremo de asociación existe un atributo privado del mismo nombre en la clase conectada al extremo opuesto (i.e., por la clase generada por el propietario del extremo)
  - No queremos definir sólo una transformación individual en atributos o desde atributos a operaciones getter/setter
  - Lo que realmente queremos es definir una transformación de clases a clases, donde los extremos de las asociaciones y los atributos son hechos corresponder según las reglas anteriores y todos los demás atributos y operaciones se mantienen!

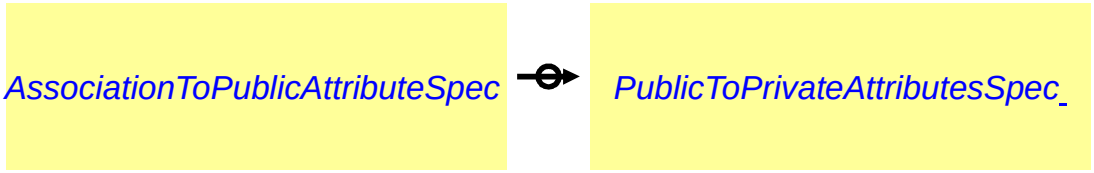

Transformación de clases (2)

- Definimos tres operaciones adicionales para el *Clasificador*:***associationEnds***, ***attributes***, y ***operations***, que devuelven las características del clasificador de tipo ***associationEnd***, ***Attribute***, y ***Operation***, respectivamente
- No tenemos que elegir explícitamente qué sub-regla de la parte derecha necesitamos para transformar los extremos de la asociación
- No tenemos que preocuparnos con el caso en que no haya extremos de asociación, o atributos, u operaciones.

```
Transformation ClassToClass (UML, UML) {
        params  --ninguno
        source
                c1: UML::Class;
        target
                c2: UML::Class;
        source condition  --  ninguna
        target condition  --  ninguna
        unidirectional;
        mapping
                c1.associationEnds() <~> c2.associationEnds();
                c1.attributes() <~> c2.attributes();
                c1.operations() <~> c2.operations();
        }
Las operaciones restantes se definen en OCL, como sigue:
        context Class def
                attributes() = feature->select( istypeOf(Attribute));
                operations() = feature->select( istypeOf(Attribute));
associationEnds() = feature->select( istypeOf(AssociationEnd));
```


Terminando la definición de transformación

- Por último, se han de incluir las reglas para hacer corresponder *asociaciones* a *atributos públicos:*
  - ManyAssociationToAttribute, SimpleAssociationToAttribute, ClassToClass,
  - Correspondencia entre atributos y operaciones existentes a los nuevos.
- La definición de transformación completa queda como una composición de 2 transformaciones más simples:




Transformaciones de modelos aplicando MDA

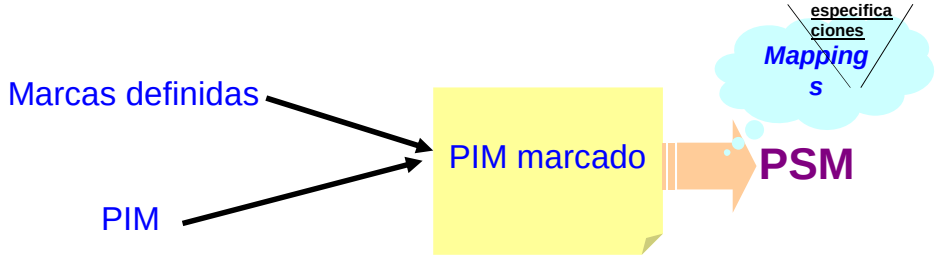
- El patrón MDA incluye al menos 3 modelos:
  - PIM
  - De plataforma
  - Transformación
  - PSM
- Ejemplo de modelos MDA:

Ejemplos de modelos MDA:
CIM: la entrega de paquetes ha de hacerse dando prioridad a los que se hayan enviado antes.
PIM: las entregas han de estar ordenadas de acuerdo con la fecha de envío.
PSM/OOD: las entregas están ordenadas para *enviar()* en orden creciente, utilizando el atributo *fecha*. **PSM/OOP: Utilizar *quicksort()* para ordenar las entregas, utilizando *Entrega.fecha* como clave de ordenación antes de pasarlas a *enviar()*.**


Aplicación del patrón de transformación

1. Marcar
2. Transformar metamodelos
3. Mezclar modelos
4. Añadir información

1 Marcar
Se utilizan perfiles UML para definir las *marcas* y para marcar los PIMs, utilizándose los lenguajes MOF, QVT para definir los *mappings*. Una *marca* representa un concepto que concierne al PSM , que puede aplicarse a un elemento del PIM para indicar cómo va a ser transformado ese elemento.




Aplicación del patrón de transformación (2)

- Los *mappings* pueden incluir *templates* (plantillas)
  - Template: *patrón de diseño* específico
    - Patrón de diseño: modelo parametrizado

```
interface MiClase{
int obtenerAtributoUno();
void ajustarAtributoUno(int v);
int operacionUno();
}
```

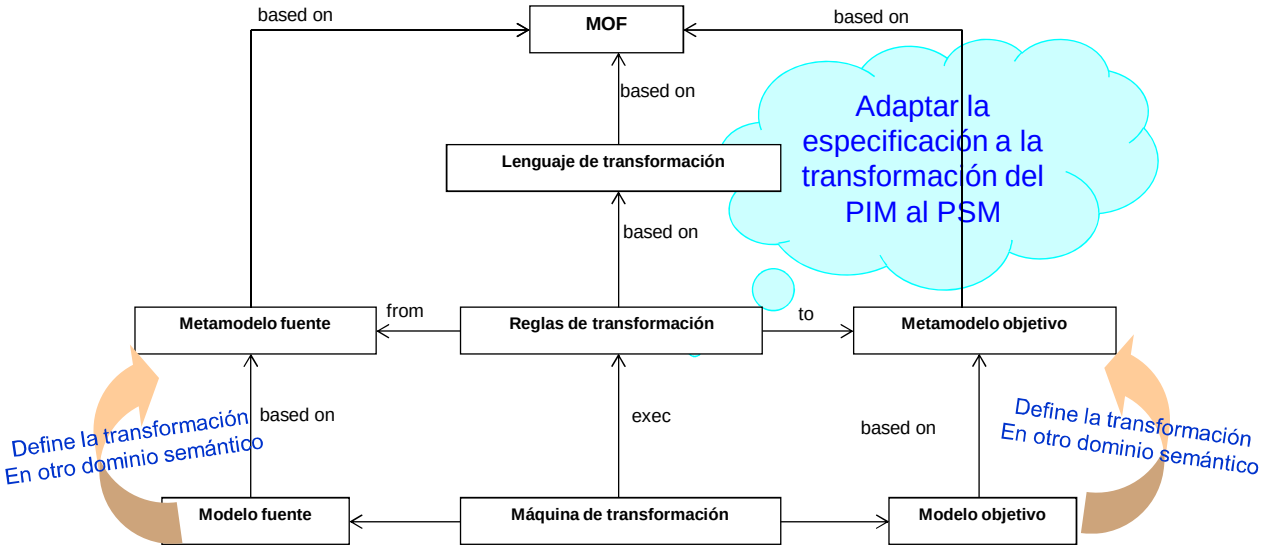
Plantilla: perfil "Security" de UML

transformación

Hay varias clases de marcas:
▪ Discriminadores y enumeradores: [isRemote | isBoolean]
▪ Cantidades: (if (numInstances <> Q and frecuency <F) LinkedList | HashTable)
▪ Entradas: (append "db_" to all operation names)

Transformación de metamodelos

- Adaptar una especificación de transformación entre 2 metamodelos



Transformación de metamodelos(2)

- **Ejemplo**: se trataría de transformar la entidad *atributo* de UML a XML:
- Para lo cual hay que definir la *relación* y el *mapping:*

Transformation AtoX		
UML	**XML**	
Attribute	element	**attribute**
name: String	name: String	**name: String**
type: string		**value: String**

```
Relation AtoX{
        Domain  {(UML.Attribute) [name= n, type= t]}
        Domain  {(XML.Element) [
                Name= "Attribute",
                Attrs={(XML.Attribute)[ name="name", value=n],
                (XML.Attribute)[name="type", value=t]}]
        }
}

Mapping MAtoX refines AtoX{
        Domain  {(UML.Attribute) [name= n, type= t]}
        body  {(XML.Element) [
                Name= "Attribute",
                Attrs={(XML.Attribute)[ name="name", value=n],
                (XML.Attribute)[name="type", value=t]}]
        }
}
```


Mezcla de modelos y adición de información

- Un patrón MDA se aplica varias veces: el PSM de una aplicación de patrón se convierte en el PIM de la siguiente
- Hay sistemas que se pueden construir en múltiples plataformas

Adición de información
Se consigue en una transformación que mezcla información de un PIM junto con la de otros modelos, así como determinada información complementaria, para llegar a obtener un PSM.

Mezcla de modelos
La mezcla de modelos es un proceso que se admite en MDA. Consiste en una clase de transformación que combina la información de un PIM con la de otro modelo para obtener un PSM.


Adición de información (2)

- Utilizando marcas desde varios modelos se transforma un PIM en un PSM, que tiene partes especificadas respecto de diferentes plataformas
- Cada modelo es independiente del resto, ya que cada uno se define de forma separada, con sus propias entidades, y reside en un nivel de abstracción bien definido.

Modelo Independiente de la Plataforma			
Modelo CORBA	Modelo Java / EJB	Modelo XML/SOAP	**Otro modelo**
CORBA	**Java/EJB**	**XML/SOAP**	**Otro**

- El desarrollo de software se convierte, de esta forma, en un proceso de *transformación de modelos* (cada paso transforma un PIM de un nivel a un PSM del nivel siguiente)
- Los modelos de aplicaciones que capturan la *lógica del negocio* son el activo más importante de la empresa, independientemente de las tecnologías que luego se utilicen para implementarlo


Proceso MDA de desarrollo de software

1.	**Definir un CIM que muestre el sistema dentro del entorno en el que va a operar, este modelo nos ayudará a entender exactamente lo que el sistema va a hacer independientemente de cómo se implementará.**
2.	**Construir un PIM, que describe el sistema, pero no muestra los detalles de su implementación en ninguna plataforma.**
3.	**En este estadío del proceso el arquitecto de software ha de escoger una o varias plataformas que permitan la implementación del sistema con las cualidades arquitectónicas deseadas.**
4.	**El arquitecto marca los elementos del PIM para indicar los mappings que han de ser usados para llevar a cabo la transformación de ese PIM en un PSM; o bien, si se utilizan *transformaciones de metamodelos*, se utilizará una máquina de transformación.**
5.	**Transformar el PIM marcado en un PSM –que puede ser realizado manualmente o con ayuda de una herramienta- ; la entrada de la transformación será el PIM marcado y el *mapping*; la salida es el PSM y el registro de la transformación.**
6.	**Un PSM puede proporcionar más o menos detalle, dependiendo de su propósito, ya que un PIM puede ser una *implementación* si proporciona toda la información necesaria para construir un sistema y ponerlo en operación, o bien puede ser el PIM de la siguiente iteración del proceso MDA hasta que se consiga una implementación adecuada del sistema.**


QVT Núcleo (Core)

- Las transformaciones se definen entre modelos mediante un conjunto de restricciones y derivaciones
- Las transformaciones tienen direcciones (una o más):
  - Cada dirección define un (modelo) fuente y/o destino de una transformación
- Un usuario al ejecutar una transformación puede:
  - Comprobar todas las restricciones (ningún efecto lateral), o
  - Reforzar (aplicar) todas las derivaciones de esa transformación en una dirección escogida
- Importación de paquetes (*packages*)

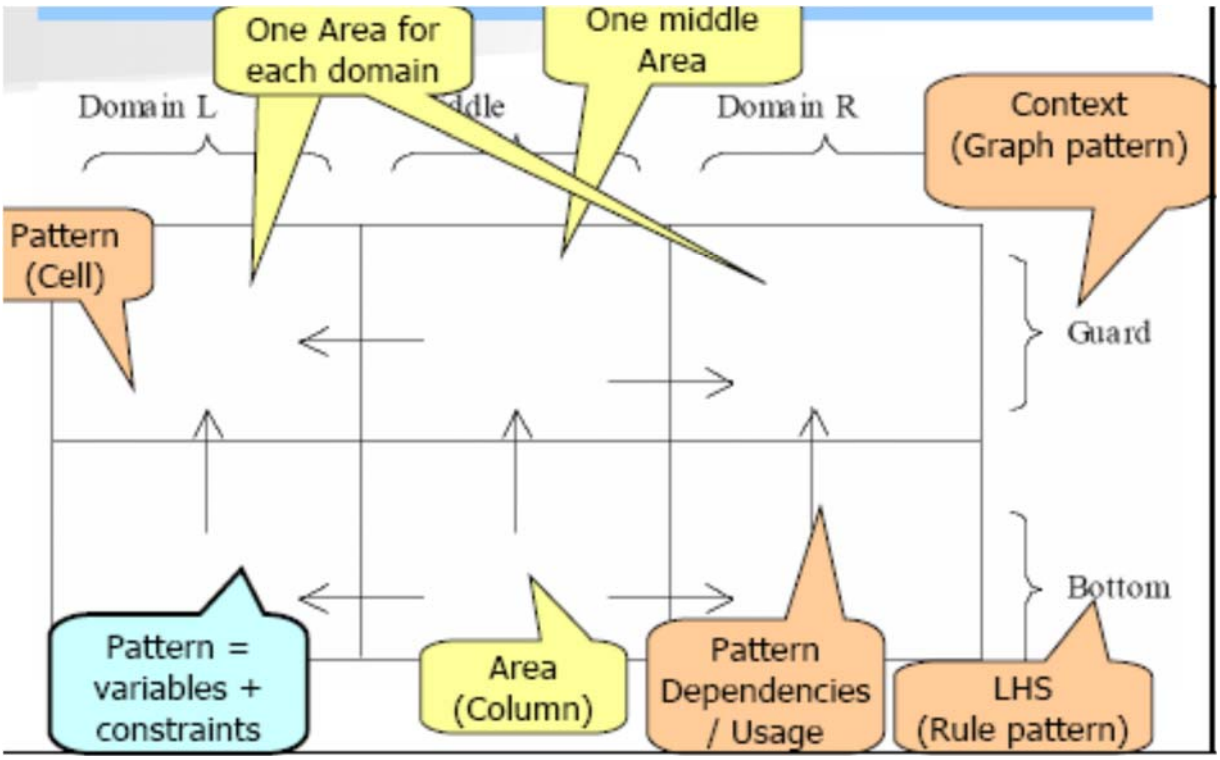

Escenarios de ejecución para el Lenguaje Nuclear

- Con transformaciones **check-only** sólo permite verificar que los modelos están relacionados de una forma perviamente especificada
- Con transformaciones **unidireccionales**
- Con transformaciones **bidireccionales**
- Estableciendo relaciones entre modelos pre-existentes
- Con actualizaciones **incrementales** (en cualquier dirección): uno de los modelos relacionados se cambia después de una ejecución inicial
- Con **creación** y **destrucción** de objetos y valores:
  - O bien especificando qué objetos y qué valores no deben ser modificados


Mapping

- Un *mapping* (correspondencia):
  - es parte de una transformación
  - posee cero ó más dominios
- Cada dominio (de un mapping)
  - está asignado a una dirección de la transformación
  - los dominios no tienen nombres:
    - el nombre de la dirección identifica de manera unívoca un dominio dentro de un *mapping*
- Cuando se ejecuta una transformación
  - todos los *mappings* de esa transformación serán ejecutados
  - los dominios son *inferidos* (derivados) en una dirección elegida

Estructura de los mappings



Restricciones aplicables a los patrones

- Toda variable debe tener un tipo asignado (por ejemplo en las *importaciones desde packages*)
- Si un patrón C depende de un patrón S, entonces C puede utilizar variables declaradas en S para las definiciones de las restricciones de C
- Resulta confusa la semántica de las dependencias entre patrones si se mezclan con las direcciones de dominios:
  - Cuando una dirección A usa otra dirección B, ¿los patrones del dA que tiene dirección A dependerán de los patrones correspondientes de dB que sin embargo poseen la dirección B?
  - ¿O nunca existirán dependencias entre los patrones de dominios que no usan las direcciones de los otros?


Ligadura (*Matching*)

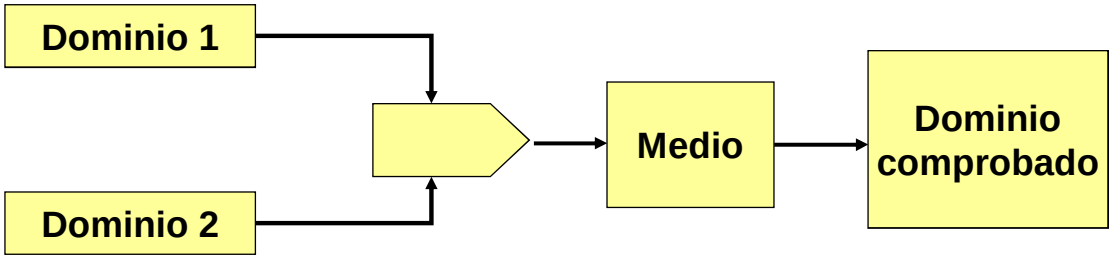
- Una ligadura válida (cuando se establece concordancia con un *patrón*) es una ligadura en la que
  - todas las variables estarán ligadas a un valor NO INDEFINIDO y
  - todas las restricciones se evalúan como TRUE
- Una ligadura parcial es una ligadura en la que
  - una o más variables están ligadas a un valor NO INDEFINIDO o bien una o más restricciones se evalúan como TRUE
  - Ninguna restricción se evalúa como FALSE
- Una ligadura inválida es una ligadura en la que al menos una de las restricciones se evalúa como FALSE
- Consecuencia:
  - Cualquier ligadura válida (de un patrón no vacío) es una ligadura parcialmente válida
  - No existen ligaduras parcialmente válidas de patrones vacíos


Dependencias entre ligaduras

- Si un patrón C depende de los patrones S1 … Sn, entonces una ligadura válida de C necesita una ligadura válida de cada S1 … Sn
- Guardas
  - definen el contexto de una concordancia
  - la concordancia de todos los patrones de base (*bottom*) sólo tiene lugar en el contexto de una ligadura válida de todos los patrones-guarda


Comprobación

- Cada dominio de una correspondencia (*mapping*) puede ser *comprobado*
- Regla de comprobación:
  - Debe haber (exactamente) una combinación válida de una *ligadura válida* del patrón medio-base (*middle-bottom*) y
  - Una ligadura válida del patrón de dominio-base (bottom-domain) del dominio comprobado
  - Para toda combinación válida de las ligaduras válidas de todos los patrones-dominio-base de todos los dominios que no coincidan con el dominio comprobado




Imposición (enforcement)

- La imposición (enforcement)
  - Altera las ligaduras de patrón-base para que se lleguen a cumplir las restricciones de comprobación
  - Las ligaduras de patrones sólo son cambiadas cuando las restricciones de comprobación de un *mapping* no se cumplen
  - La **dirección de imposición** se puede elegir en el momento de la evaluación
- Las restricciones (en el contexto de una ligadura) pueden ser expresadas como
  - predicados
  - asignaciones


Imposición (2)

- Un predicado se evalúa como TRUE, FALSE o *indefinido*
  - los predicados sólo se utilizan para concordancia
- Las asignaciones asignan valores a las propiedades
  - Las asignaciones son usadas para proporcionar valores a las variables,
  - propiedades a las variables que poseen valores-objeto
  - para hacer una comprobación de los patrones se evalúan como TRUE
- Existen problemas semánticos acerca de si las asignaciones pueden ser asignaciones por defecto:
  - Una asignación que no es una asignación por defecto también posee el significado de un predicado si el operador de asignación se sustituye por el operador '='
  - Las asignaciones por defecto sólo asignan valores para satisfacer la semántica de comprobación, no poseen un significado como predicados

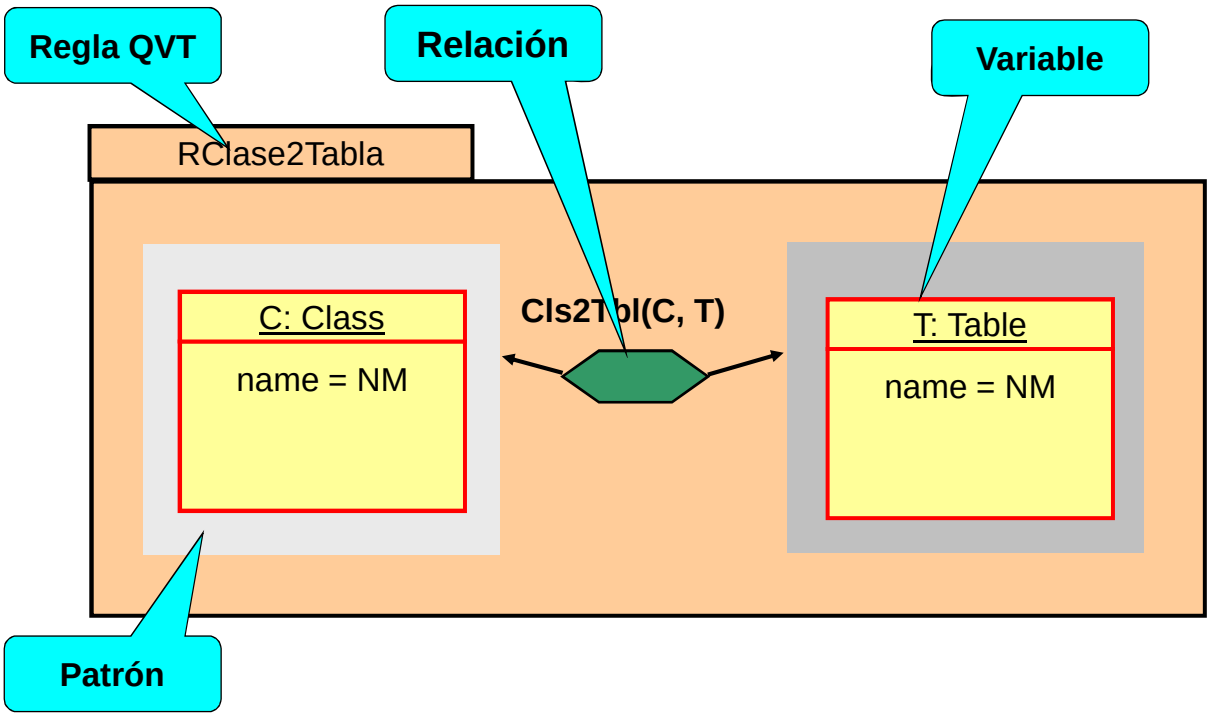

Imposición (Creación y Destrucción)

- Las variables pueden ser realizadas
- Cuando se *realiza* una variable:
  - se podría crear una nueva instancia del tipo de esa variable, o bien
  - se podría destruir un valor previamente existente de esa variable
- Rol de las variables realizadas:
  - Las variables *no-realizadas* sólo se usan para *concordancia (matching)*
  - Las *variables realizadas* se usan tanto para *concordancia* como para *imposición*
- Las asignaciones y las realizaciones de variables pueden ser sólo definidas en:
  - Patrones del *dominio-base* que sufren una *imposición* y
  - En el patrón *medio-base*

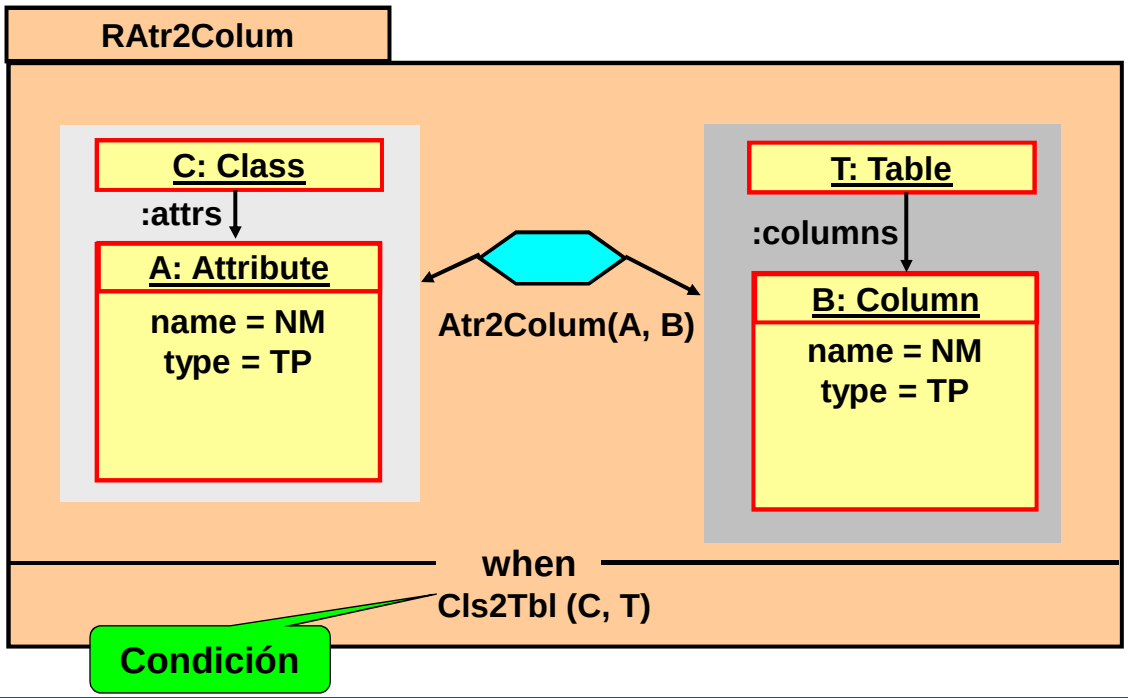

Imposición (Creación y Destrucción) -2

- Regla de creación:
  - Si no existiera una ligadura de un determinado patrón y
  - Una ligadura válida de ese patrón fuera requerida por la semántica de comprobación,
  - Entonces se crearán nuevas instancias de todos los tipos variables realizadas no ligadas y se ligarán a valores cuando sean ejecutadas todas las asignaciones
- Regla de destrucción:
  - Si existe una ligadura válida de un patrón y
  - La comprobación requiere que no exista una ligadura válida para ese patrón,
  - Entonces todas las asignaciones serán anuladas y todos los valores de las variables realizadas serán destruidos


Ejemplo de utilización de QVT



Ejemplo 2 de utilización de QVT

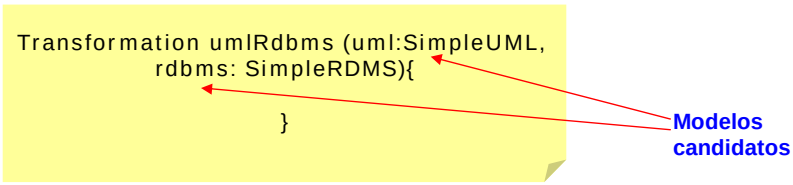



Relaciones QVT

- Constituyen una especificación declarativa de las relaciones entre modelos MOF
- El lenguajes de relaciones QVT incluye:
  - Constructos para la *concordancia* entre patrones complejos de objetos
  - Creación implícita de clases que registran (*trace*) lo que ocurre durante la ejecución de una transformación
- El lenguaje es una combinación entre Lenguajes Natural y Lógica de Predicados de Primer Orden
- También se define una transformación estándar entre relaciones definidas sobre modelos *transportables* y un *modelo nuclear* con una semántica equivalente
- El *modelo nuclear* es ejecutado en 1 sola "máquina" que implementa la semántica del citado *núcleo*


El Lenguaje de Relaciones

- Una transformación entre *modelos candidatos* es especificada como un conjunto de relaciones que deben mantenerse:
  - Los tipos de elementos que pueden contener están restringidos a los de los paquetes referenciados



Los paquetes simples **uml** y **rdbms** son los metamodelos

Una transformación puede ser utilizada bien para comprobar la consistencia entre modelos bien para modificar un modelo y reforzar sus consistencia

Elementos del lenguaje Relacional

- **Relación** = restricción que ha de ser satisfecha por los elementos de los modelos candidatos
- `when`, `where` especifican restricciones que se mantienen entre los elementos de los modelos candidatos
- **Dominio** = variable de tipo distinguible con la que se ha de concordar exactamente dentro de un tipo de modelo
- **Patrón** = plantilla para los objetos y sus propiedades, que debe ser identificado en el modelo candidato para satisfacer una relación
- **Patrón** = {{variables}, {restricciones}}. Los valores de las variables han de satisfacer las restricciones para ser aceptados como ligaduras válidas del patrón

Ejemplos de expresiones en el lenguaje relacional

- Dos dominios que concuerdan con elementos de los modelos UML y RDMS:

```
relation PackageToSchema // hace corresponder un esquema a cada paquete
 {
   domain uml p: Package {name=pn}
   domain rdbms {name=pn}
 }
```

Patrón:
dos propiedades 'name' que estén ligadas a la misma variable 'pn' han de tener el mismo valor

Restricciones:
Dos conjuntos de predicados
-> when(se mantiene la relación)
->where(satisfecha por todos los elementos

Ejemplos de expresiones en el lenguaje relacional (2)

- Implementación de la relación anterior:

```
relation ClassToTable //hace corresponder cada clase persistente a una tabla
{
        domain uml c: Class{
                namespace = p:Package{},
                kind = 'Persistent',
                name = cn
        }
        domain rdbms t : Table{
                schema = s : Schema {},
                name = cn,
                column = cl:Column {
                        name = cn+' _tid',
                        type = 'NUMBER'},
                        primaryKey = k : PrimaryKey{
                        name = cn+'_pk',
                        column = cl}
                }
        }
        when{
                PackageToSchema(p, s);
                }
        where{
                AttributeToColumn(c, t);
                }
        }
```

Clases de relaciones

(1) **Alto nivel**→ la ejecución de una transformación obliga a que todas sus relaciones de alto nivel se mantengan
(2) **Bajo nivel** →se mantienen sólo cuando son invocadas desde la claúsula **where** de otra relación

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
        top relation PackageToSchema{ ... }
        top relation ClassToTable{ ... }
}
```

(1) checkonly domain modelo:sólo comprueba si hay concordancia en el modelo de ese dominio que satisfaga la relación

(2) enforce domain modelo si la comprobación de concordancia falla, el modelo destino es modificado para satisfacer la relación

```
relation PackageToSchema //hacer corresponder cada paquete aun esquema
{
        checkonly domain uml p : Package {name = pn}
        enforce domain rdbms s : Schema {name = pn }
}
```


Concordancia de patrones

- Los patrones asociados a los dominios (*object template expressions*) han de concordar en los modelos candidatos:
  - La *concordancia* resulta de una ligadura de los elementos del modelo candidato a las variables declaradas en el dominio
  - La concordancia con la expresión *plantilla* encuentra ligaduras sólo para las variables *libres* del dominio
- Ejemplo: la expresión *plantilla de objetos tipo Class* que se asocia al dominio *uml:*

```
c : Class{  namespace = p : Package{},
        kind = 'Persistent',
        name = cn
        }
```

Cualquier clase con su propiedad **kind** *distinta* de **persistent** será eliminada

Si la variable **cn** es libre, entonces está variable conseguirá una ligadura en el valor de **name**


Dirección de ejecución

- **Multiplicidad:** Cada asignación de valores a las tuplas de variables de la plantilla (p.e.: *(c, p, cn) de la plantilla Class* del modelo UML) resultará en una concordancia válida
- **Tratamiento de la multiplicidad:** depende de la dirección de ejecución de la relación:
  - **En el ejemplo anterior** ClassToTable (dirección de ejecución hacia *rdbms*):
    - **Para cada concordancia válida que represente 1 *clase* en UML, ha de existir una tabla válida en el dominio *rdbms*, si no: SE CREARÁ una tabla que concuerde con la expresión plantilla**
    - **Si NO EXISTE al menos 1 *clase* válida en UML para 1 tabla válida del dominio *rdbms*, entonces se eliminará la tabla del modelo para que deje de ser una concordancia válida**
- Esto es así porque el dominio destino (*rdbms*) tiene la propiedad de ser ***enforced*** y el dominio fuente (*uml*) se ha marcado como ***checkonly***


Creación de objetos con patrones

- Una expresión *plantilla de objetos* también sirve como una plantilla para crear un objeto en un modelo destino cuando para una concordancia *válida* del patrón dominio *fuente* no existe una concordancia válida del patrón dominio *destino*
- ***Ejemplo:*** *ClassToTable* ejecutada con *rdbms* como destino

```
t: Table{
        schema = s:Schema {},
        name = cn,
        column = cl : Column {name = cn+'_tid', type='NUMBER'},
        primaryKey = k:PrimaryKey{name = cn+'_pk', column = cl}
}
```

Problema: cuando creamos objetos queremos asegurar que no se dupliquen objetos si estos ya existen


Creación de objetos con patrones (2)

- MOF permite que una única propiedad de una clase sea señalada como *identificante*
- Pero no esto no es suficiente, en la mayoría de los modelos, para poder identificar a los objetos unívocamente
- **Solución**: utilizar un conjunto de propiedades (concepto de *clave*) para identificar de una manera única a un objeto
  - Una *clase* puede poseer múltiples *claves*, tal como ocurre con las bases de datos relacionales, para identificar instancias

Por ejemplo, si seguimos con la relación **ClassToTable**, podríamos desear especificar que en modelos de ***simpleRDBMS*** una tabla es identificada de forma única mediante dos propiedades: su nombre y el esquema al que pertenece:
key Table {schema, name};


Creación de objetos con patrones (3)

- Caso práctico de uso de claves en el que no es necesaria la creación de objetos:

En nuestro ejemplo, considérese el caso en el que tenemos una clase persistente con el nombre "foo" en el modelo *uml* y existe una tabla con un nombre concordante "foo" en un esquema concordante en el modelo *rdbms*, pero la tabla no tiene una concordancia válida del patrón asociado con *Table* (ya que dos de sus propiedades no casan) y necesitamos crear objetos que satisfagan la relación.

Ejecución de la relación

puesto que la tabla existente concuerda con las propiedades clave especificadas (esto es, concuerda con "nombre" y "esquema"), no tenemos por qué crear una nueva tabla, sino únicamente actualizarla asignando las propiedades "column" y "primaryKey".


Restricciones sobre expresiones (1)

- Las expresiones que aparezcan en una *relación* cumplirán:
  - Las claúsula **when**, los dominios **fuente** y la cláusula **where** han de aparecer en un orden secuencial que contenga sólo las expresiones:

    <objeto>.<propiedad> = <variable> | <expresion>(nota)
- Esta expresión proporciona una ligadura para la *variable libre* **<variable>**. **<objeto>** es una variable ligada a una expresión plantilla de objeto del dominio *fuente*
- Ninguna otra expresión posee apariciones de variables libres (todas las apariciones de sus variables han de haber sido ligadas en las expresiones precedentes)

Nota: No hay ninguna aparición de variables libres en <expresión>


Restricciones sobre expresiones (2)

- Las expresiones que aparecen en el dominio *destino* se organizan en un orden secuencial que contiene sólo los siguientes tipos de expresiones:

  <objeto>.<propiedad> = <expresion>(nota)
  - Esta expresión proporciona una ligadura para la *variable libre* **<variable>**. **<objeto>** es una variable ligada a una expresión plantilla de objeto del dominio *destino*
  - Ninguna otra expresión posee apariciones de variables libres (todas las apariciones de sus variables han de haber sido ligadas en las expresiones precedentes)


Cambio de propagación

- El efecto de la semántica *comprobar-previo-reforzar* es que los elementos del modelo *destino* que concuerden con las reglas no serán tocados
- La selección de objetos basada en claves asegura que los objetos existentes son actualizados cuando sea permitido
- Un objeto es destruido sólo cuando ninguna otra regla exige que el objeto exista.

Las implementaciones son libres de utilizar cualquier algoritmo eficiente para cambiar la propagación siempre que sea consistente con la semántica anterior. Las correspondencias relaciones-a-núcleo (relations-to-core) proporcionan tales opciones de implementación, ya que las correspondencias al núcleo soportan más directamente la propagación de cambios incremental .


Integración de operaciones caja-negra con relaciones

- Opcionalmente una relación puede tener una implementación operacional de *caja-negra*
- La operación *caja-negra* si la relación se ejecuta en la dirección del d. reforzado y asigna *falso* en su semántica de comprobación
- La operación anterior es responsable de hacer los cambios al modelo para que satisfaga la relación especificada
- La signatura de la operación se puede derivar de la especificación del dominio de la relación como un parámetro de salida (para el dominio *reforzado*) y de entrada para los otros dominios
- Estas restricciones permiten una semántica de llamada simple, que no necesita ninguna evaluación de restricciones previa
- Existe una restricción para las relaciones que pueden ser implementadas como operaciones de *caja-negra*:
  - Su dominio debe ser primitivo o contener una plantilla de objeto simple (sin sub-elementos)
  - Las claúsulas when y where no deben definir variables

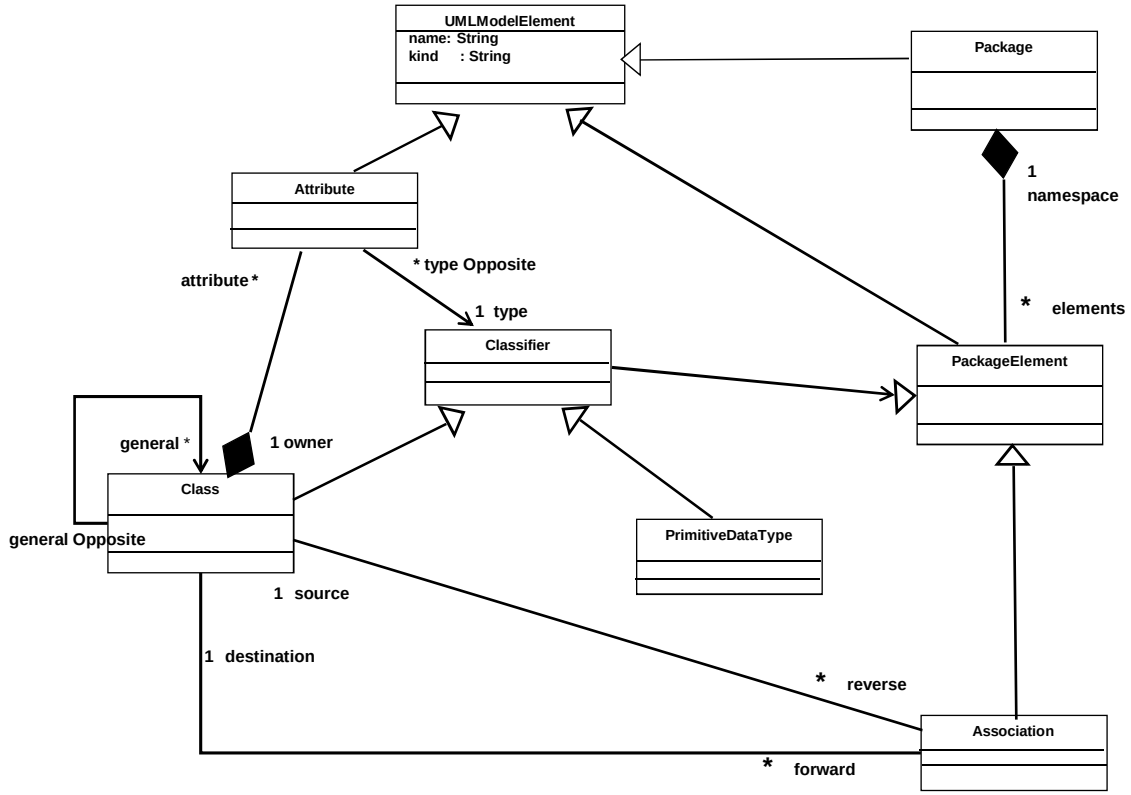

Ejecución sólo en modo *checkonly*

- Una transformación puede ser ejecutada en modo "checkonly"
- Simplemente comprueba si la relación se mantiene en todas las direcciones y reporta errores cuando no
- No se hace reforzamiento en ninguna dirección, independientemente de que los dominios se marquen como **checkonly** o **enforced**.

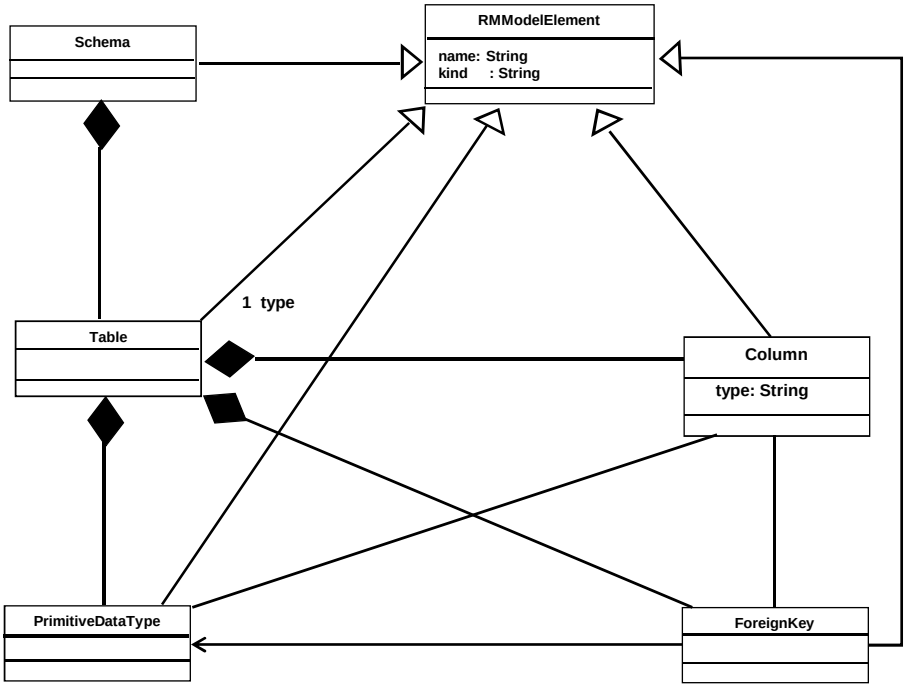

UML→RDBMS

- Ejemplo de Mapping UML → RDBMS
  - El objetivo es hacer corresponder clases persistentes de un modelo simple de UML a tablas de un modelo simple RBDMS
  - Una clase *persistente* se hace corresponder con: (1) tabla, (2) clave primaria, (3) columna(de la tabla) para identificar
  - 1 *atributo* de la clase se hacen corresponder a 1 columna si su tipo es primitivo, sino (atributo de un tipo complejo) se hace corresponder a un conjunto de columnas
  - Atributos heredados son hechos corresponder a las columnas
  - Las asociaciones entre 2 clases persistentes se hacen corresponder a una relación de clave externa entre tablas

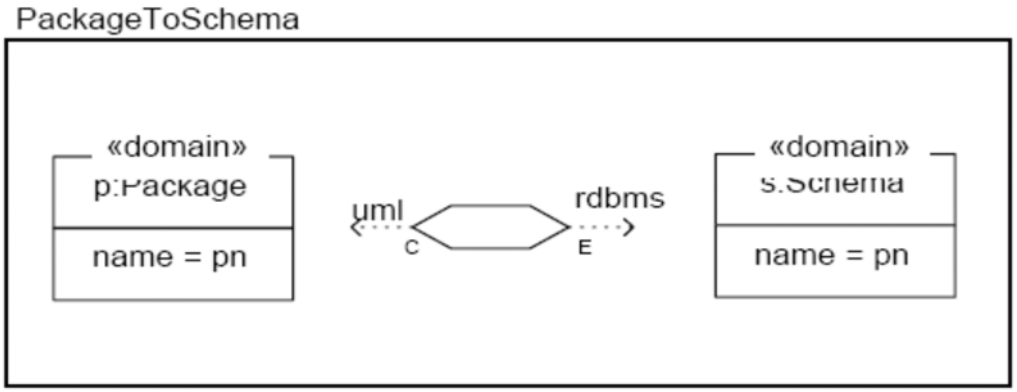

Metamodelo UML simple



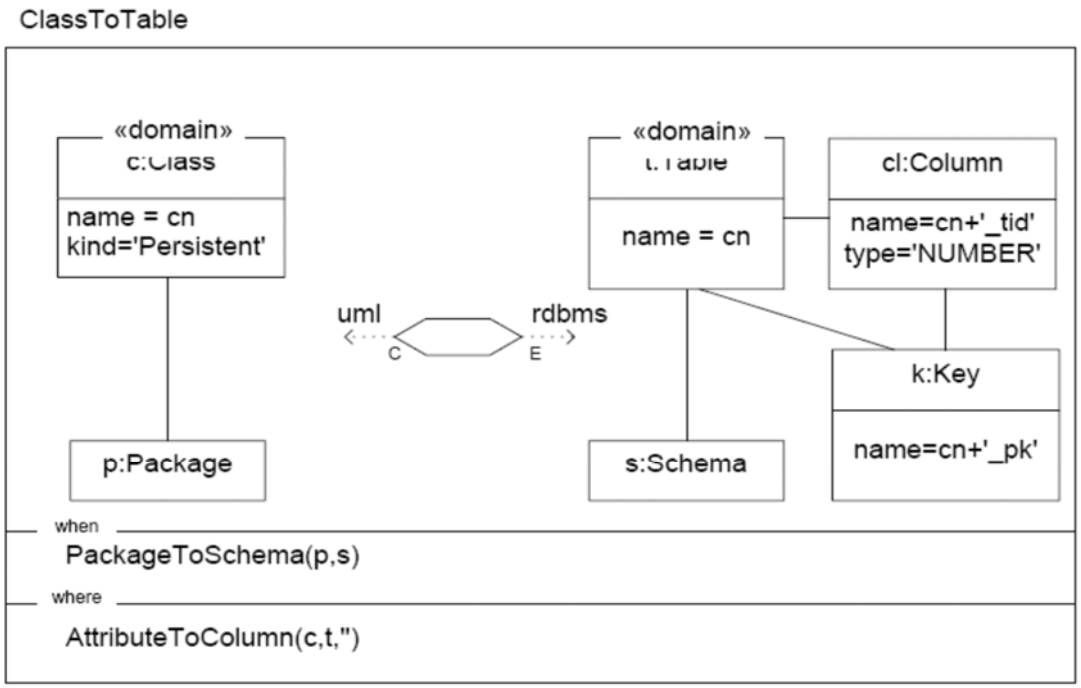
Metamodelo RDBMS



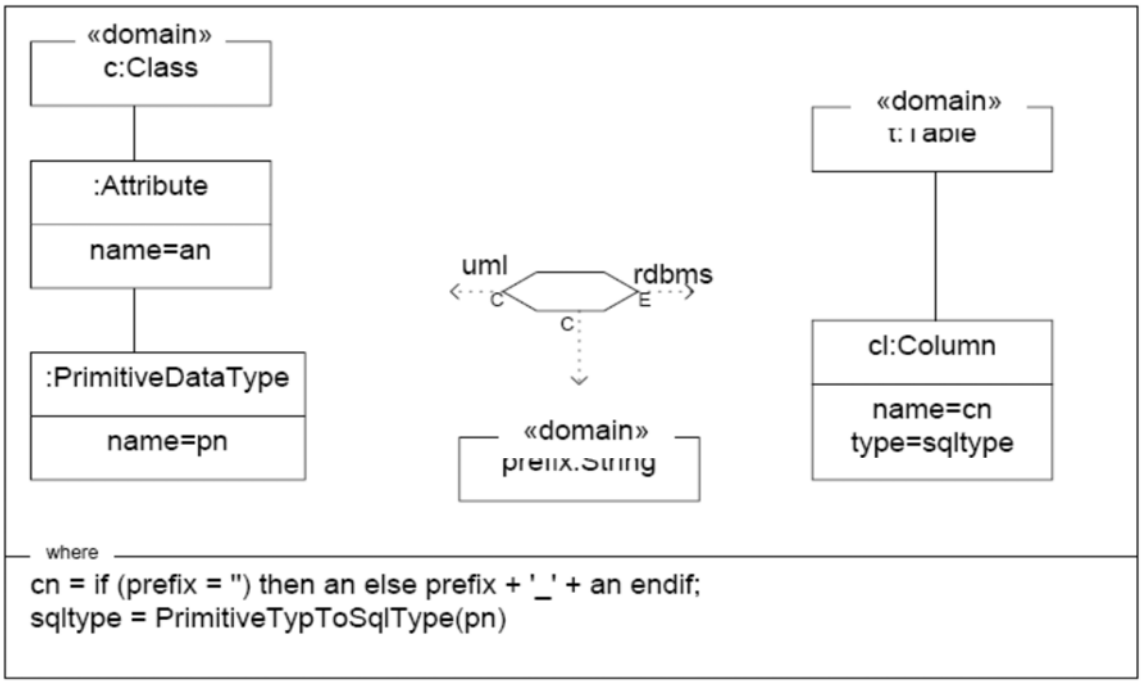
Relación PackageToSchema



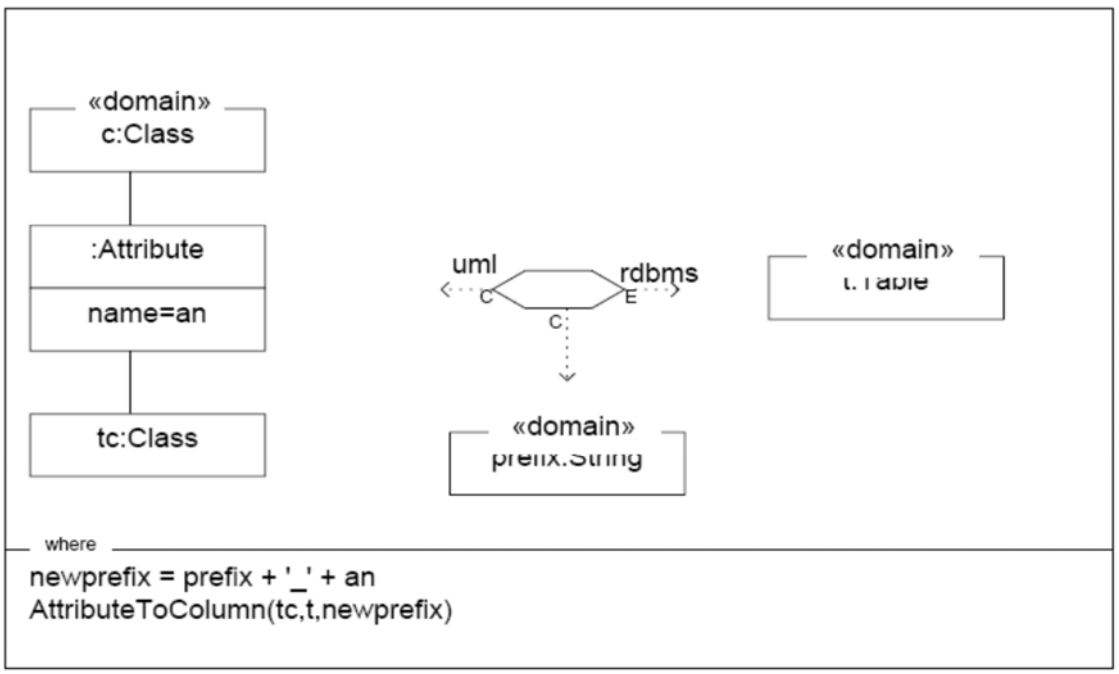
Relación ClassToTable



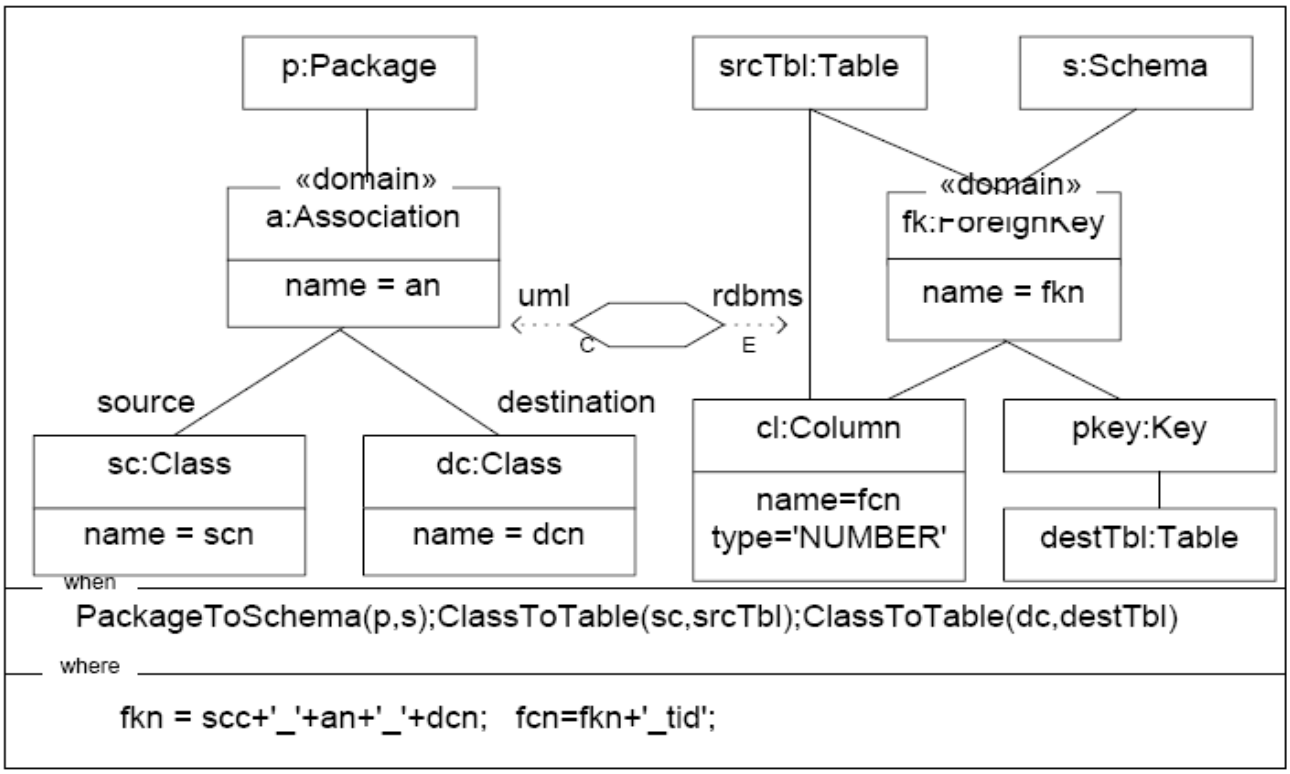
Relación PrimitiveAttributeToColumn



Relación ComplexAttributeToColumn



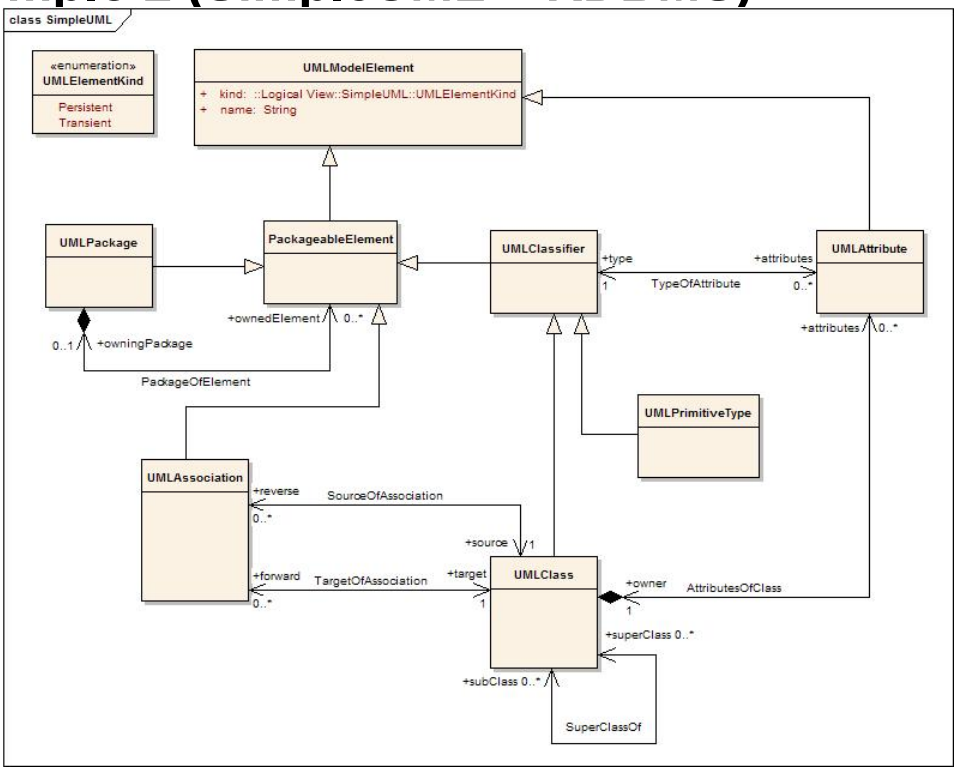
Relación AssocToFKey



Estado actual de QVT

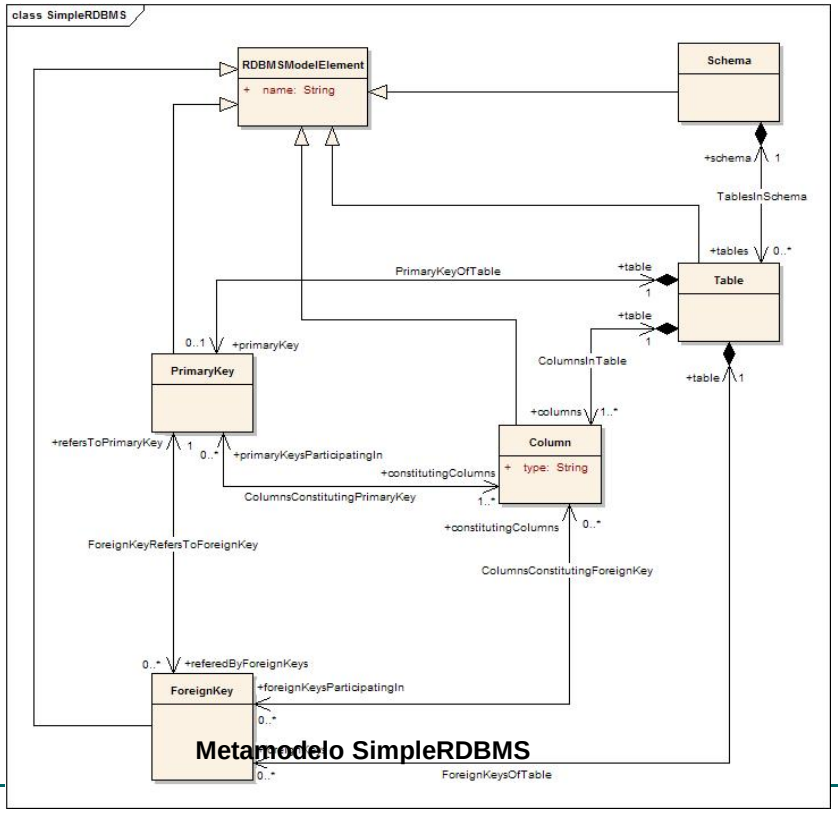
- QVT= Revised submission for MOF 2.0 Query/View/Transformation RFP Final (marzo de 2007)
- ¿Está la tecnología QVT lo suficientemente madura en la actualidad para ser utilizada en empresas?
  - No existe todavía ninguna herramienta software que soporte todo QVT propiamente, PERO
  - Tarde o temprano la habrá
- Si QVT se convierte en un estándar, ¿sobrevivirán otros enfoques de Transformación de Modelos?
  - En concreto ¿qué será de GT?
- Herramienta recomendada **MEDINI/QVT** (ver en http://projects.ikv.de/qvt/downloads)

Ejemplo 2 (SimpleUML-> RDBMS)



Metamodelo SimpleUML

Ejemplo 2 (SimpleUML-> RDBMS)



Metamodelo SimpleRDBMS

Relaciones en Ejemplo 2 (SimpleUML-> RDBMS)

- Cualquier *package* en el modelo candidato cuyo atributo *name* esté asignado será ligado a la variable del patrón "p" y transformado a un esquema cuyo nombre se liga a "s"; a través de la que el nombre del esquema se ligará al del paquete.
  - **Cualquier esquema pre-existente que no tenga un paquete con el que se corresponda será eliminado**

Las relaciones "top-level" ha de cumplirse y son siempre ejecutadas

El patrón de objeto "c" sólo concuerda con clases cuyos paquetes estén ligados a "s"

Sólo se ejecutan cuando son llamadas desde otra relación o desde el "where"

```
transfor
rdbms:Si
top relation Package2Schema {
    theName: String;
    checkonly domain uml p:UMLP
        name = theName };
    enforce domain rdbms s:Schema {
    name = theName };
}
 top relation Class2Table {
    theName: String;
    prefix: String;
    checkonly domain uml c:UMLClass {
        name = theName,
        owningPackage = p:UMLPackage {},
        kind = UMLElementKind::Persistent
    };
    enforce domain rdbms t:Table {
        name = theName,
        schema = s:Schema {},
        columns = cl:Column {
            name = theName + '_PrimaryKeyColumn',
            type = 'NUMBER' },
        primaryKey = pk:PrimaryKey {
            name = prefix+ 'PrimaryKey'
        titutingColumnc = cl }

    hen { Package2Schema(p, s); }
    here { prefix = theName + '_';
        Attribute2Column(c, t, prefix); }

relation Attribute2Column {
    checkonly domain uml c:UMLClass {};
    enforce domain rdbms t:Table {};
    primitive domain prefix:String;
    where {  ComplexAttribute2Column(c, t, prefix);
        PrimitiveAttribute2Column(c, t, prefix);
        SuperAttribute2Column(c, t, prefix);
    }
} } /*Fin de transformación
```


Consultas en Ejemplo 2 (SimpleUML-> RDBMS)

- La **consulta PrimitiveTypeToSqlType** es usada en la relación **PrimitiveAttribute2Column** para calcular el tipo SQL de una columna basándonos en el tipo de su atributo

```
transformation umlToRdbms(uml:SimpleUML,
rdbms:SimpleRDBMS)
{
   /* All same as before */
   relation PrimitiveAttribute2Column
   {
      attributeName, primitiveTypeName, className,
sqltype: String;

      checkonly domain uml c:UMLClass {
         attributes = a:UMLAttribute {
            name = attributeName,
            type = p:UMLPrimitiveType {
               name = primitiveTypeName
            }
         }
      };

      enforce domain rdbms t:Table {
            columns = cl:Column {
               name = columnName,
               type = sqlType
            }};
         primitive domain prefix:String;
         where {
            columnName = if (prefix = '') then attributeName
                  else prefix + attributeName
               endif;
            sqltype =
PrimitiveTypeToSqlType(primitiveTypeName);
}}
query PrimitiveTypeToSqlType(typename : String) :
String {  if (typename = 'INTEGER') then 'NUMBER'
         else if (typename = 'BOOLEAN') then 'BOOLEAN'
            else 'VARCHAR'
         endif
      endif
   }
```

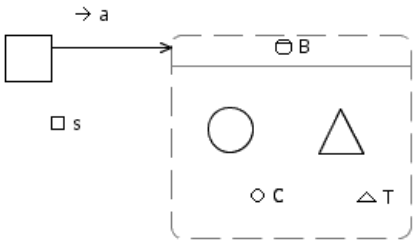
Importación de transformaciones

```
transformation common(uml:SimpleUML,
rdbms:SimpleRDBMS) {
    top relation ClassToPrimarykey {
        theName: String;
        checkonly domain uml c:UmlClass {
        name = theName };
        enforce domain rdbms k:RdbmsKey {
        name = theName+ '_PrimaryKey' }; } }
```

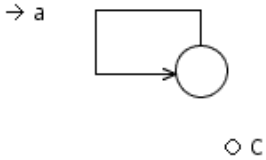
```
import common.qvt;
transformation umlToRdbmsExtended(
    uml:SimpleUML,
    rdbms:SimpleRDBMS) extends umlToRdbms {
    top relation OverridingClass2Table overrides
Class2Table {
 /* Relation variables and uml domain same as before */
        enforce domain rdbms t:Table { /* other features
same as before */
        primaryKey = pk:PrimaryKey {
        constitutingColumnc=cl } }; /* when clause same as
before */
    where { /* Other where expressions same as before
*/ common::ClassToPrimaryKey(c, pk); } }
top relation AssociationToForeignKey { /* Content
omitted here */ } }
```


Caso de estudio: metamodelo *Shapes* ¶

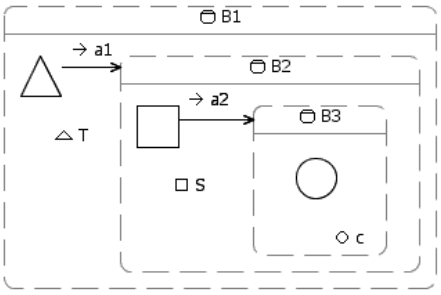
- El lenguaje “Shapes” es un lenguaje sintáctico simple que se ha diseñado para ilustrar patrones de transformación de modelos. El lenguaje consiste en tres formas (“shapes”) simples: *Squares*, *Circles* y *Triangles*. Además, el lenguaje contiene constructos relacionados con el *agrupamiento* llamados *Block* para expresar jerarquía. Un *Block* puede contener formas simples y otros bloques. Las Relaciones entre formas simples y bloques se expresan mediante flechas (*Arrows*). Debido a que las transformaciones entre modelos son sólo definidas sobre su estructura y sintaxis, el lenguaje *Shapes* no tiene definida ninguna semántica. El metamodelo para el lenguaje *Shapes* ➔



Relación entre una forma simple y un bloque



Una referencia cíclica de Circle c hacía sí mismo usando "Arrow" a



Anidación de bloques que contienen relaciones

Mapeo de nodos

- Sirve para establecer relaciones uno-a-uno entre elementos del modelo *fuente* a elementos del modelo *destino*.
  - La transformación de *mapping* entre un nodo *x* de tipo *X* y nodo *y* de tipo *Y* en caso de que sus respectivos contextos estén relacionados

```
top relation NodeMapping {
  nm: String;
  enforce domain left x: X {
    context = c1:X Context {},
    name = nm    };
  enforce domain right y: Y {
    context = c2:Y Context {},
    name = nm };
  when {
    ContextMapping(c1,c2);
  }
}
```

- La transformación del mapping *relation* es similar a la transformación del mapping *node* salvo que añade algunas restricciones (claúsula **when**)
  - Si los nombres y contexto se pueden mapear, y si los *fuentes* y *destino* de las relaciones se pueden mapear.

```
top relation RelationshipMapping {
   nm: String;
   enforce domain left a: A {
    context = c1 : A Context {},
    name = nm,
    source = as : AS {},
    target = at : AT {}
   };
   enforce domain right b: B {
     context = c2 : B Context {},
     name = nm, source = bs : BS {},
     target = bt : BT {}
   };
   when {
     ContextMapping(c1,c2);
     ElementMapping(as,bs);
     ElementMapping(at,bt);
   } }
```

Mapping *identity*

- *mapea* todos los elementos del dominio *fuente* al dominio *destino* y por ese medio se preserva la estructura.
  - hace uso de 2 relaciones: RootBlockMapping y BlockMapping. Los bloques que son hechos corresponder son forzados a satisfacer las restricciones especificadas en ElementMapping (claúsula **where**)

```
relation ElementMapping {
   nm: String;
   enforce domain left e1: Shapes::ModelElement {
     name = nm,
     block = b1 : Block {}
   };
   enforce domain right e2: Shapes::ModelElement {
     name = nm,
     block = b2 : Block {}
   };
   when {
     RootBlockMapping(b1,b2) or BlockMapping(b1,b2);
   }
}
```

Mapping *identity* (2)

```
-- map RootBlock elements
 top relation RootBlockMapping {
   nm: String;
 enforce domain left b1:RootBlock{
     name = nm    };
 enforce domain right b2: RootBlock{
     name = nm };
}
```

```
-- Map Block element that are not
RootBlock elements
  top relation BlockMapping {
    enforce domain left e1: Block {};
    enforce domain right e2: Block {};
    when {
      not(RootBlockMapping(e1,e2));
    }
    where {
      ElementMapping(e1,e2);
    }
  }
```


El patrón de aplanamiento (*flattening*)

- **Objetivo:** Eliminar la jerarquía del modelo fuente.
- **Motivación:** Los modelos son a menudo estructurados jerárquicamente. Considerar por ejemplo la jerarquía de paquetes en UML, los estados compuestos en Statecharts o las Petri Nets jerárquicas. La intención de tal estructuración jerárquica usualmente es hacer los modelos más fáciles de entender. Sin embargo, el analizar tales modelos podría ser necesario para aplanar el modelo a uno sin jerarquía.
- **Flattening:** La siguiente transformación indica cómo aplanar un modelo fuente compuesto a un modelo destino. Se supone que el modelo *fuente y* el modelo *destino* son especificados por el mismo metamodelo. La idea del mapping es como sigue. Todos los *Composites* en el modelo *fuente* están relacionados con el elemento *RootElement* en el modelo *destino*. El *CompositeContext* es bien el *RootElement* u otro *Composite*. Entonces, el *CompositeContext c1* debería ser relacionado con el *RootElement r* mediante *RootMapping* o mediante el mismo *CompositeFlattening*. Todos los elementos del modelo son hechos corresponder del modelo *fuente* al modelo *destino* usando instancias de la transformación *CompositeFlattening*.


El patrón de aplanamiento (*2*)

```
top relation CompositeFlattening {
 checkonly domain left c:Composite {
    context = c1 : CompositeContext {}
  };
enforce domain right r: RootElement{};
  when {
    RootMapping(c1,r) or
CompositeFlattening(c1,r);
  }
}
```

```
relation ElementMapping {
  nm: String;
  enforce domain left x: Element {
    name = nm,
    context = c1 :Context {}
  };
  enforce domain right y: Element {
    name = nm,
    context = c2 : Context {}
  };
  when {
    RootMapping(c1,c2) or
CompositeFlattening(c1,c2);
  }
}
```


Patrón para refinar una jerarquía

- **Objetivo:** Obtener un modelo objetivo más detallado refinando un nodo dado en un *nodo jerárquico* que contiene subestructuras.

- **Motivación:** el refinamiento de jerarquías de nodos se utiliza de una manera típica para *hacer zoom* en la estructura de un *supernodo* y de este modo se revela la estructura interna de un *subnodo*. En modelos de *comportamiento*, el refinamiento *jerárquico* puede ser utilizado para hacer zoom en los comportamientos de los subnodos.

- **Patrón de transformación jerárquico**: este patrón está caracterizado por una *regla de mapeo* simple que toma un nodo del dominio *fuente* y lo hace corresponder a una estructura jerárquica (**calledb_sub** en el patrón) que puede contener subnodos (**calledsub_node** en el patrón). La especificación del patrón de refinamiento jerárquico se muestra también.


Patrón para refinar una jerarquía (código)

```
top relation HierarchyRefinementMapping {
  n : String;
  enforce domain left node: Node {
    name = n,
    block = b_left : Block {}
  };
  enforce domain right b_sub: Block {
    name = 'Block(' + n + ')',
    block = b_right : Block {}
  };
}
```

```
enforce domain right sub_node : Node {
    name = 'e',
    block = b_sub
  };
  when {
    RootBlockMapping(b_left,b_right) or BlockMapping(b_left,b_right);
  }
  where {
    ElementMapping(node,b_sub);
  }
```


Ejercicio con el modelo

- Construir una transformación que copie todos los elementos del modelo “Shapes” del dominio *source* al *target* al mismo tiempo que se reemplazan *Circles* por *Squares*; también los asociaciones(“Arrows”) deben ser hechas corresponder a las correspondientes del dominio *target*.