TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria dell'Impresa
Corso di Informatica
Service Oriented Software Engineering

UML to Odata Transformations

Prepared by: Federica Magliocca

Abstract

In un processo di sviluppo software **Model Driven Architecture (MDA)**, i modelli vengono ripetutamente trasformati in altri modelli per ottenere finalmente un insieme di modelli con dettagli sufficienti per implementare un sistema.
In questo caso il sistema preso in esame è il sistema **OData**, si tratta di un protocollo per la pubblicazione e il consumo di servizi online basati su dati interrogabili e interoperabili.
Basandomi sull'aritcolo *"Model-driven Development of OData Services: An Application to Relational Databases"* di H. Ed-douibi, J.L.C. Izquierdo e J. Cabot, ho implementato una trasformazione da un modello UML di origine a un modello OData di destinazione.
Il progetto è stato realizzato facendo uso della piattaforma **Eclipse Modeling Project (EMP)** per la specifica delle trasformazioni, definite attraverso il linguaggio **Query/View/Transformation (QVT)**.
La trasformazione da UML a OData è stata successivamente verificata attraverso diversi modelli UML di input, scelti in modo tale da coprire tutti i casi possibili.

Contents

1 Model Driven Engineering MDE

Model Driven Engineering (MDE) è la pratica per la realizzazione di sistemi software attraverso l'uso di modelli per analizzare, simulare e ragionare sulle proprietà del sistema stesso.

1.1 Model

Un modello è un'astrazione che rappresenta una vista parziale e semplificata di un sistema software; quindi la creazione di più modelli è solitamente necessaria per rappresentare e comprendere meglio il sistema.

I modelli rendono la pianificazione del progetto più efficace ed efficiente fornendo al contempo una visione più appropriata del sistema da sviluppare.

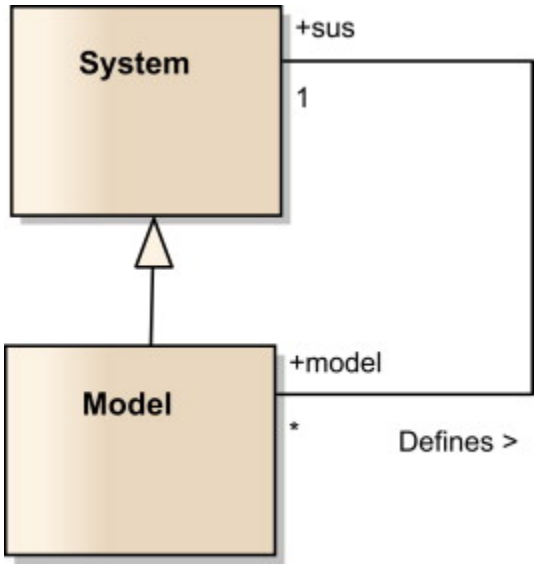

Figure 1.1: Relazione tra modello e sistema

1.2 Metamodel and Meta-metamodel

Un metamodello è un modello che definisce il linguaggio per esprimere un modello, in altre parole è il modello che fornisce le primitive utilizzate per costruire, interpretare e verificare il modello.

Un modello è conforme a un metamodello, nel senso che il modello dovrebbe soddisfare le regole definite a livello del suo metamodello.

Se un modello è costruito attraverso il suo metamodello, il metamodello deve essere costruito a sua volta con un meta-metamodello, il quale dovrebbe avere un meta-meta-metamodello che lo definisce e così via. La soluzione comune per superare questo problema è utilizzare un linguaggio che, a un livello particolare di questa gerarchia, si descriva nella propria lingua.

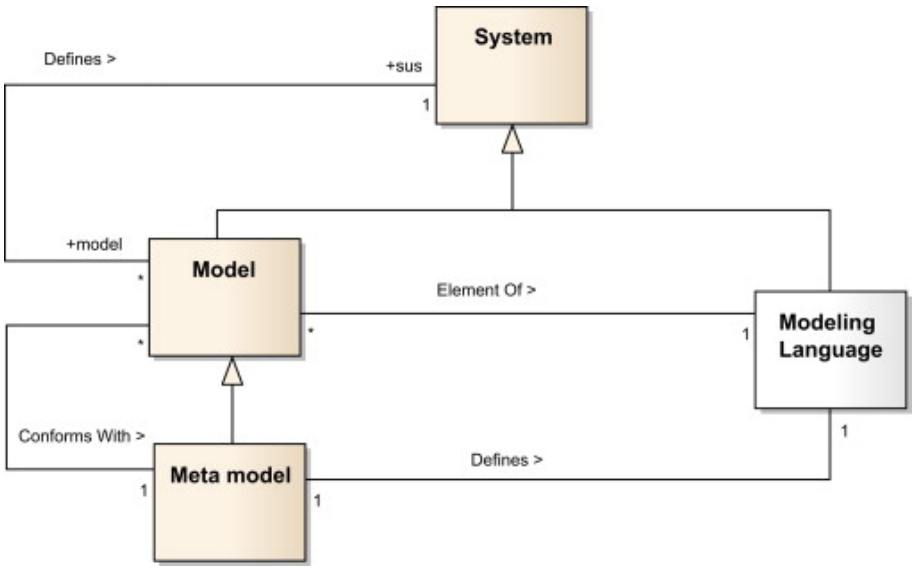

Figure 1.2: Relazione tra metamodello e modello

1.3 Model Driven Architecture MDA

Model Driven Architecture (MDA) è un'approccio Model Driven focalizzato principalmente sulla definizione dei modelli e delle loro trasformazioni.

L'approccio MDA permette di:

- specificare un sistema indipendentemente dalla piattaforma che lo supporta;
- specificare le piattaforme;
- scegliere una piattaforma particolare per il sistema;
- trasformare le specifiche del sistema in una per una piattaforma particolare.

MDA supporta la definizione di modelli a diversi livelli di astrazione, vale a dire **Computational Independent Models (CIM)**, **Platform Independent Models (PIM)** e **Platform Specific Models (PSM)**.

- Un CIM è una vista di un sistema che si focalizza sull'ambiente e sui requisiti del sistema, senza mostrare i dettagli della struttura. I requisiti CIM dovrebbero essere riconducibili ai costrutti PIM e PSM che li implementano e viceversa.
- Un PIM è una vista di un sistema che si concentra sul funzionamento di un sistema nascondendo i dettagli necessari per una particolare piattaforma. Questo lo rende adatto all'uso con un certo numero di piattaforme diverse.
- Un PSM combina le specifiche nel PIM con i dettagli che specificano come quel sistema utilizza un particolare tipo di piattaforma.

Inoltre MDA fornisce le specifiche per la trasformazione di un PIM in un PSM per una particolare piattaforma.

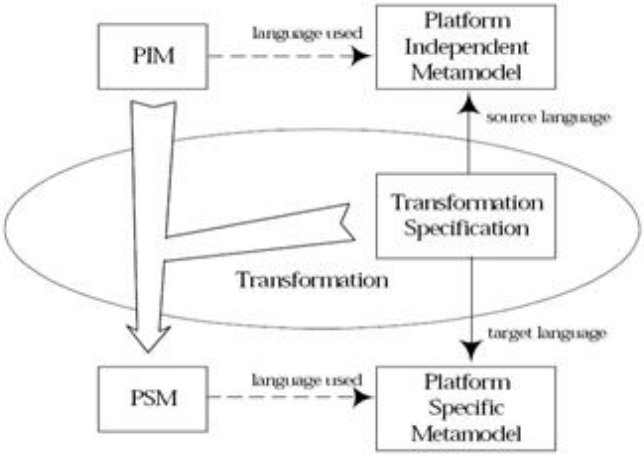

Figure 1.3: Trasformazione da PIM a PSM

Per consentire tali trasformazioni, l'MDA richiede che i modelli siano espressi in un linguaggio basato su MOF.

1.3.1 Meta Object Facility MOF

MOF è un meta-metamodello unico perché è istanziato dal proprio modello, cioè il MOF è il metamodello del MOF.

MOF fornisce un insieme di concetti per definire i metamodelli, in particolare:

- Diagrammi di classe per definire la sintassi astratta
- OCL per definire la semantica di un linguaggio di modellazione

MOF è progettato con un'architettura a quattro livelli:

- **M3** è un meta-metamodello auto-descrittivo di MOF;
- **M2** è un metamodello costruito secondo lo standard MOF (ad esempio, UML 2.0);
- **M1** è un modello utente;
- **M0** è un oggetto di realtà, soggetto a modellazione;

UML è un esempio di linguaggio di modellazione basato su MOF. Quindi un modello UML può essere serializzato in XMI.

1.3.2 XML Metadata Interchange XMI

Un modo standard di mappare oggetti in XML.
Definisce due set di regole:

- Un set per la serializzazione di oggetti in documenti XMI
- Un set per la generazione di schemi XML da modelli

1.4 QVT

Il linguaggio utilizzato per trasformare i modelli è QVT, l'abbreviazione di **Query, View, Transformation**. Il nome suggerisce una struttura in tre parti.

La prima parte è denominata **Query** perché le query possono essere applicate a un modello di origine, un'istanza del metamodello di origine; **View** è una descrizione di come dovrebbe apparire il modello di destinazione; e infine **Transformation** è la parte in cui i risultati delle query vengono proiettati sulla vista, e creano così il modello di destinazione.

QVT è composto da tre diversi linguaggi specifici del dominio: Relations, Core e Operational Mappings.

1.4.1 Relations

Nella variante Relations una trasformazione tra modelli consiste in un insieme di relazioni che dovrebbero valere tra modello sorgente e destinazione. Tale relazione è definita da due o più domini e una coppia di pre-condizioni(**when-condition**) e post-condizioni(**where-condition**).
Un dominio è un elemento di un tipo specifico che può essere trovato nel modello.
Una trasformazione può essere **"checkonly"** o **"enforced"**. Il primo controlla solo la coerenza mentre il secondo impone la coerenza modificando il modello di destinazione.

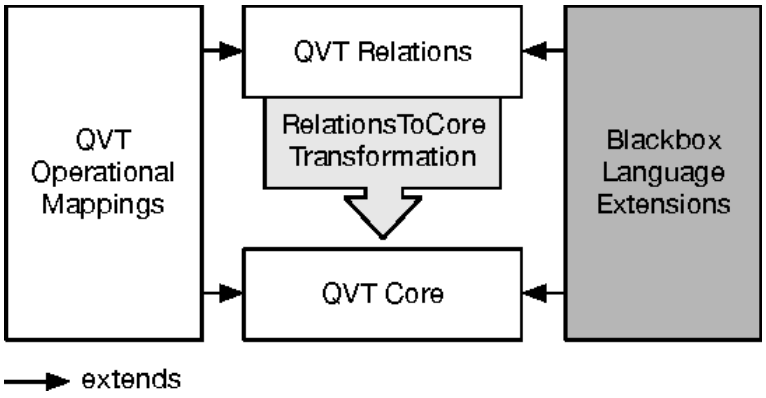

Figure 1.4: linguaggio QVT

1.4.2 Core

Core è un piccolo linguaggio dichiarativo potente quanto il linguaggio Relations. Ma a differenza di Relations, Core non utilizza modelli. A causa della sua relativa semplicità, la sua semantica può essere definita più semplicemente.
Una trasformazione nella variante Relations può essere trasformata in una nella variante Core.

1.4.3 Operational Mappings

Operational Mappings, spesso chiamato QVTo, è un linguaggio imperativo che estende Relations e Core. La sintassi è simile a quella di altri linguaggi imperativi, come Java. A differenza delle varianti dichiarative, le mappature operative sono unidirezionali: c'è un modello di origine e uno o più modelli di destinazione.

1.4.4 Blackbox

Una Blackbox consente di codificare algoritmi complessi in qualsiasi linguaggio di programmazione supportato. Questo è molto utile perché alcuni algoritmi sono difficili da implementare in OCL, o non possono essere espressi affatto.

2 UML to OData

Open Data Protocol (OData) è un protocollo per favorire la pubblicazione e il consumo di servizi online basati su dati interrogabili e interoperabili.
Grazie alla sua maturità e facilità d'uso per utenti e applicazioni, OData è diventata una scelta molto comune di pubblicare insiemi di dati online. Tuttavia, la creazione di servizi OData è un compito noioso e dispendioso in termini di tempo, dal momento che i fornitori di dati dovrebbero rappresentare i loro modelli di dati in formato OData, implementare la logica di business per trasformare le richieste OData in istruzioni SQL e de/serializzare i messaggi scambiati conformi al protocollo OData.

2.1 MDA for OData Services

Nell'articolo *"Model-driven Development of OData Services: An Application to Relational Databases"* di H. Ed-douibi, J.L.C. Izquierdo e J. Cabot, viene fornito un metodo basato su modelli per automatizzare la generazione di servizi OData pronti per la distribuzione. In particolare, viene proposto un approccio MDA in cui i modelli OData vengono derivati da un modello UML di input.
Quindi viene utilizzato il modello UML come Platform-Independent Model (PIM). e il metamodello OData come Platform Specific Model (PSM).

2.2 The OData metamodel

Gli autori forniscono un metamodello OData per le trasformazioni. Tale metamodello è allineato con la specifica OData CSDL che definisce i concetti principali che devono essere esposti da un qualsiasi servizio OData, facilitando così in seguito la generazione di documenti metadata OData.
La parte in alto del metamodello comprende l'elemento **ODService**, che include una

serie di schemi. Uno o più schemi definiscono il modello dati di un servizio OData.
Uno schema è rappresentato dall'elemento **ODSchema** che include lo spazio dei nomi dello schema, il namespace e i riferimenti alle strutture dati definite dallo schema come enumerazioni (enumTypes), tipi complessi (complexTypes) e tipi di entità (entityTypes).
I tipi di entità e i tipi complessi sono rappresentati da **ODEntityType** e **ODComplexType**, rispettivamente. Tutti e due gli elementi sono sottotipi di ODStructuredType, elemento astratto che rappresenta un tipo strutturato.
Un tipo strutturato è composto da proprietà strutturali e proprietà di navigazione e può definire un tipo di base (baseType). L'elemento **ODProperty** definisce un attributo del tipo strutturato, mentre l'elemento **ODNavigationProperty** definisce un associazione tra due tipi di entità.
L'elemento ODSchema include inoltre un contenitore di entità (entityContainer) che definisce gli insiemi di entità e singleton interrogabili e aggiornabili dal servizio.
Tutte le strutture dati nel metamodello sono sottotipi dell'elemento **ODType** che descrive la struttura di un tipo di dati astratto.

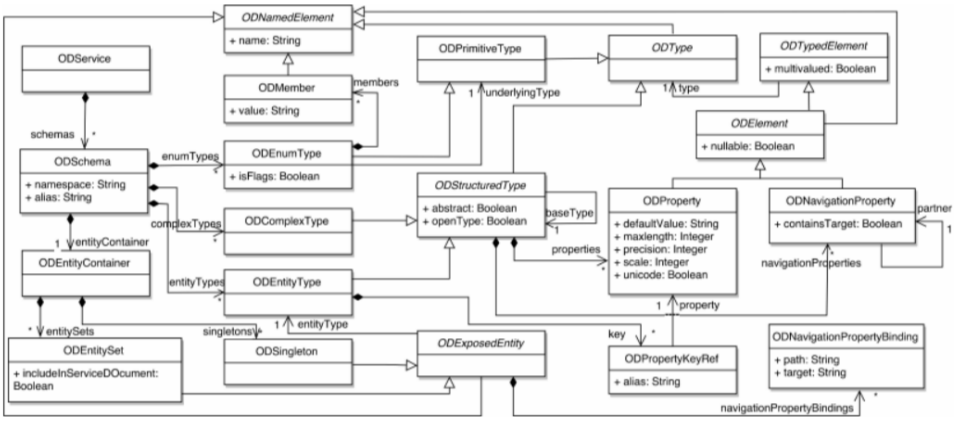

Figure 2.1: metamodello OData

2.3 Mapping UML to OData Models

I modelli OData possono essere derivati automaticamente da modelli UML mediante una trasformazione da modello a modello.
La tabella in Figura 3.3 mostra le principali regole di trasformazione dagli elementi del metamodello UML agli elementi del metamodello OData descritte nell'articolo.
Le istanze degli elementi **ODEntityType**, **ODComplexType** e **ODEnumType**, una volta creati, vengono aggiunti all'elemento **ODSchema**, e allo stesso modo le istanze di **ODEntitySet** vengono aggiunte all'elemento **ODEntityContainer**.
Inoltre, ogni istanza dell'elemento **ODProperty** e **ODNavigationProperty**, viene aggiunta al suo corrispondente **ODEntityType** e **ODComplexType**, così come ogni istanza di **ODNavigationPropertyBinding** viene aggiunta al suo **ODEntitySet** corrispondente.

Ref.	Source elements	Conditions	Target elements	Initialization details
1	c: Class	-	et: ODEntityType es: ODEntitySet	- et.name = c.name - **if** c.abstract = true **then** et.abstract ← true - **if** c.generalizations contains a class cc **then** et.baseType ← t where t is the corresponding EntityType of cc - et.properties ← *(cf. rules to transform attributes, rows 3 and 4)* - et.navigationProperties ← *(cf. rule to transform navigable association ends, i.e., row 5)* - es.name ← the plural form of c.name - es.entityType ← et - es.navigationPropertyBindings ← *(cf. rule to transform navigable association ends, i.e., row 5)*
2	dt: DataType	-	ct: ODComplexType	- ct.name ← dt.name - **if** ct.abstract = true **then** dt.abstract ← true - **if** dt.generalizations contains a data type dd **then** ct.baseType ← t where t is the corresponding ODComplexType of dd - ct.properties ← *(cf. rules to transform attributes, i.e., rows 3 and 4)*
3	p: Property	p is a class attribute or a data type attribute	op: ODProperty	- op.name ← p.name - op.type ← t where t is the corresponding type of the attribute - **if** p is multivalued **then** op.multivalued ← true
4		p is a class attribute marked as ID	pk: ODPropertyKeyRef	- pk.property = op
5		p is a navigable association end	np: ODNavigationProperty npb: ODNavigationPropertyBinding	- np.name ← p.name - np.type ← t where t is the corresponding entity type of p.type - **if** p.aggregation = Composite **then** np.containsTarget ← true - **if** p is multivalued **then** np.multivalued ← true - npb.path ← p.name - npb.target ← t.name where t is the corresponding entity set of p.type
6	e: Enumeration	-	oe: ODEnumType	- oe.name ← e.name - oe.members ← *(cf. rule to transform literals, i.e., row 7)*
7	el: EnumerationLiteral	-	om: ODMember	- om.name ← el.name

Figure 2.2: Regole di trasformazione

3 Transformation UML to OData

Per realizzare il progetto ho utilizzato la piattaforma **Eclipse Modeling Project (EMP)**, un framework di modellazione basato su Eclipse per la creazione di strumenti e altre applicazioni basate su un modello di dati strutturati.

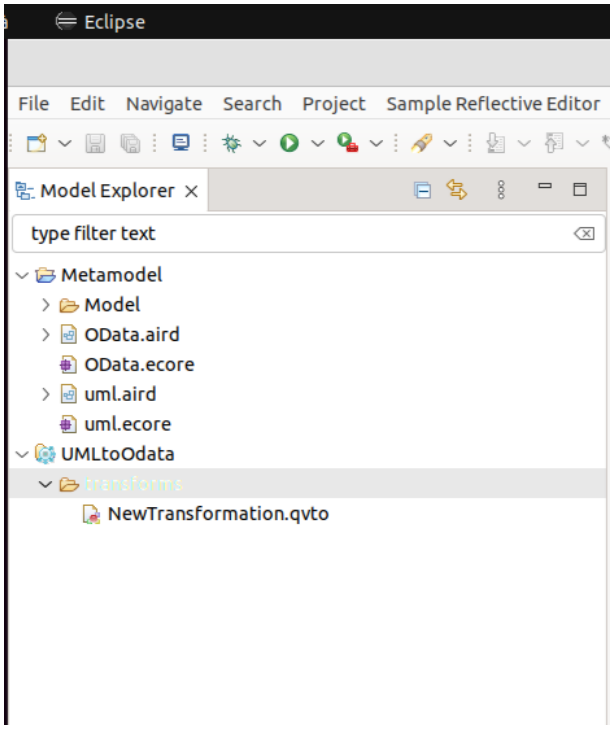

Figure 3.1: Progetto su Eclipse

Il progetto è composto da due cartelle:

- La cartella **Metamodel** che contiene il metamodello UML e il metamodello OData descritti attraverso il metamodello Ecore, e una cartella **Model** all'interno della quale vi sono tutti i modelli UML di esempio utilizzati per verificare la correttezza della trasformazione;

- La cartella **UMLtoOData** contenente le specifiche della trasformazione in QVT da UML a OData.

3.1 Metamodels

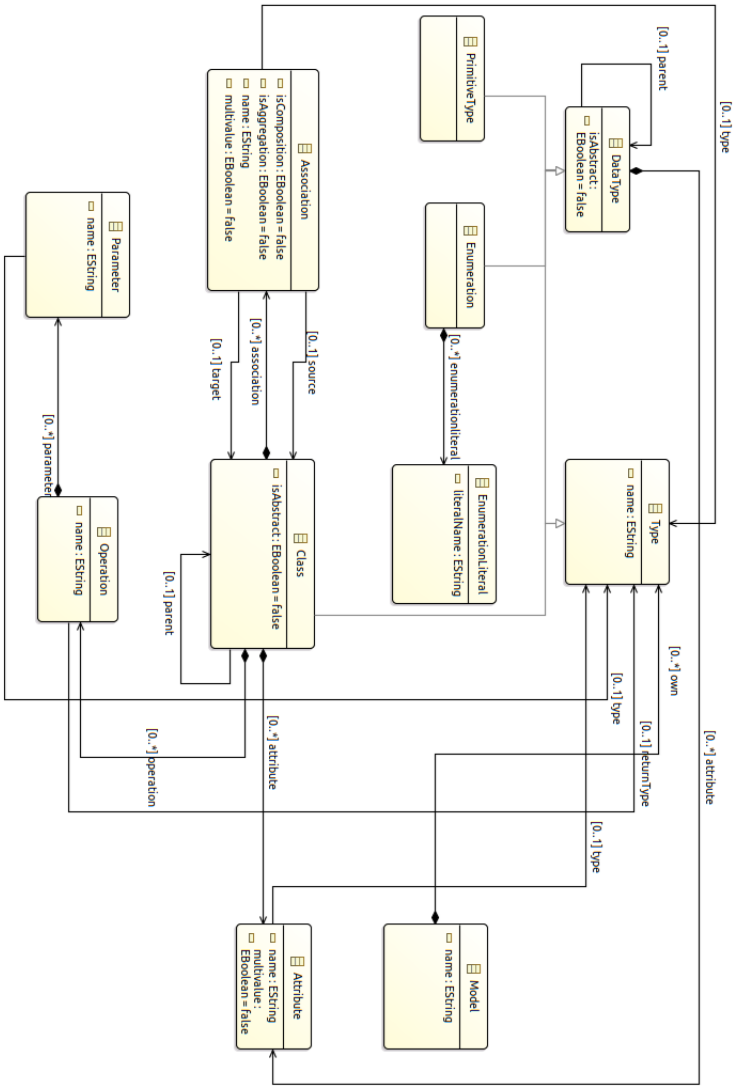

Figure 3.2: Metamodello UML in Ecore

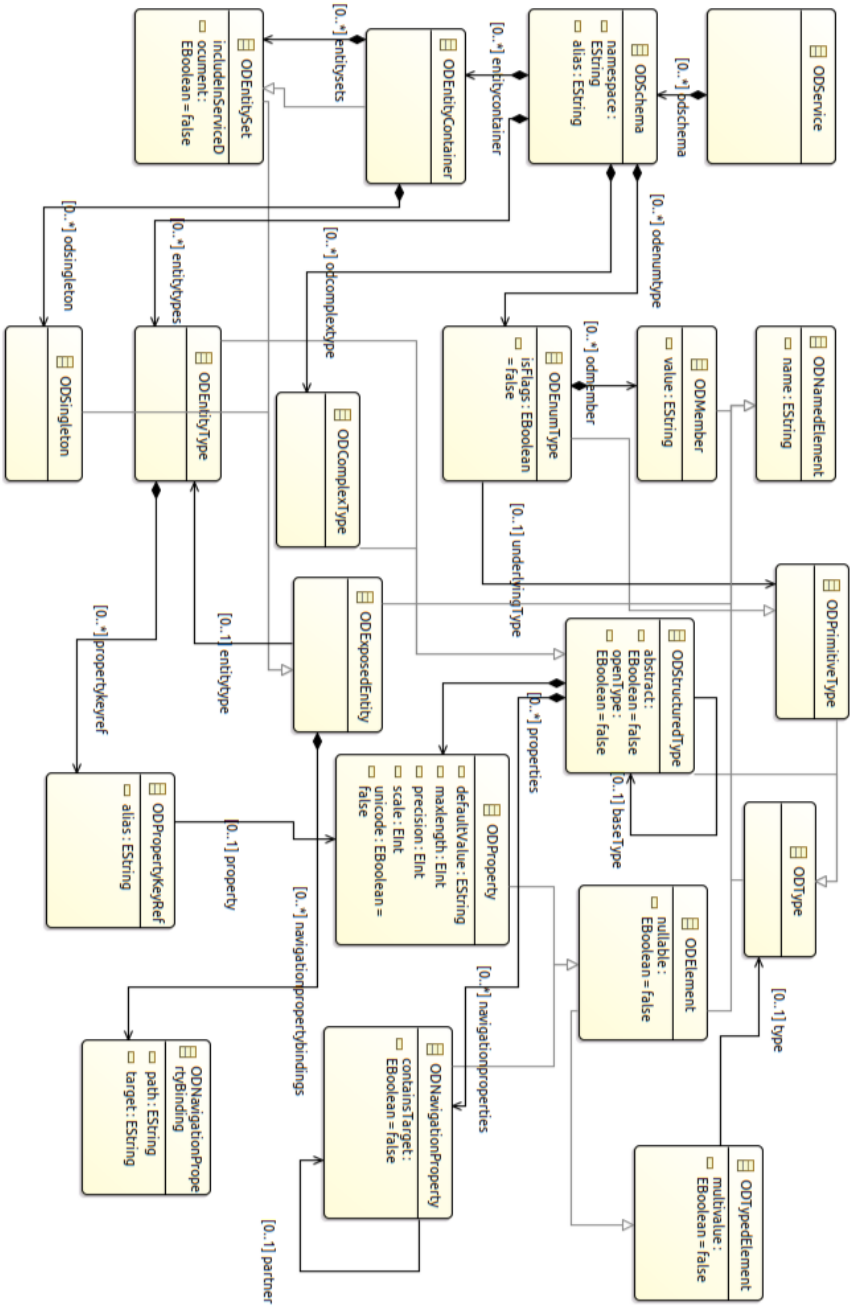

Figure 3.3: Metamodello OData in Ecore

3.2 Transformation

Per realizzare la trasformazione da UML a OData è necessario importare i corrispondenti metamodelli creati nella cartella Metamodel.
La trasformazione **NewTransformation()** prende in input un modello UML di origine e resituisce il modello OData di destinazione. All'interno di essa sono definite tutte la mappature dagli oggetti del metamodello UML a quelli del metamodello OData.
La funzione **main()** utilizza la funzione rootObjects() per ottenere l'istanza del modello e passarla in input a ModelToOdSchema().
ModelToOdSchema() trasforma un'istanza Model in un'istanza ODSchema assegnandovi un namespace, un alias, un insieme di oggetti ODEntityType richiamando ClassToODEntityType() e un ODEntityContainer attraverso la funzione ODEntityContainer().
ODEntityContainer() crea un oggetto ODEntityContainer con un nome e un insieme di oggetti ODEntitySet, richiamando la mappatura ClassToODEntitySet().

```
modeltype UML "strict" uses uml('http://umlmodel.com');
modeltype OData "strict" uses OData('http://odata.com');

transformation NewTransformation(in source: UML, out target: OData);

main() {
source.rootObjects()[Model] -> map ModelToOdSchema();
}

mapping Model::ModelToOdSchema() : ODSchema{
//s.schemaNamespace = "com.example" + m.name
result.namespace := "com.example.OData" + self.name;
//s.schemaAlias = m.name
result.alias := self.name;
result.entitytypes := self.own[Class].map ClassToODEntityType();
result.entitycontainer := self.map ODEntityContainer();
}

mapping Model::ODEntityContainer() : ODEntityContainer{
//s.entityContainerName = m.name + "Service"
result.name := self.name + "Service";
result.entitysets := self.own[Class].map ClassToODEntitySet();
}
```

La funzione **ClassToODEntityType()** trasforma un oggetto Class in un oggetto ODEntityType con un nome e un valore booleano abstract uguale a true se la classe è astratta. Inoltre se la classe ha una generalizzazione viene creato un tipo ODEntityType che sarà il baseType dell'oggetto ODEntityType risultante.
All'oggetto ODEntityType vengono poi associati gli attributi della classe trasformati in oggetti ODProperty, un ODPropertyKeyRef, se la classe ha un attributo chiave, e un

insieme di ODNavigationProperty che rappresentano le associazioni della classe.
La funzione **ClassToODEntitySet()** mappa un oggetto Class in un oggetto ODEntitySet assegnandovi un nome, l'ODEntityType precedentemente creato con la funzione ClassToODEntityType() e un insieme di ODNavigationPropertyBinding.

```
mapping Class::ClassToODEntityType() : ODEntityType {
//et.name = c.name
result.name := self.name ;
//if c.abstract = true then et.abstract ← true
if (self.isAbstract = true) result._abstract := true;
//if c.generalizations contains a class cc then et.baseType ← t
↪  where t is the corresponding EntityType of cc
if (self.parent <> null)
        result.baseType  := self.parent.map ClassToODEntityType();
//et.properties ← (cf. rules to transform attributes, rows 3 and 4)
//p is a class attribute or a data type attribute
result.properties := self.attribute.map AttributeToODProperty();
//p is a class attribute marked as ID
var IDAttributes : OrderedSet(Attribute);
self.attribute->forEach(i){if (i.name="ID") IDAttributes += i;};
result.propertykeyref := IDAttributes.map
↪  AttributeToODPropertyKeyRef();
//et.navigationProperties ← (cf. rule to transform navigable
↪  association ends, i.e., row 5)
//p is a navigable association end
result.navigationproperties:= self.association.map
↪  AssociationToODNavigationProperty();
}

mapping Class::ClassToODEntitySet() : ODEntitySet {
//es.name ← the plural form of c.name
result.name := self.name + "s";
//es.entityType ← et
result.entitytype := self.resolveone(ODEntityType);
//es.navigationPropertyBindings ← (cf. rule to transform navigable
↪  association ends, i.e., row 5)
result.navigationpropertybindings := self.association.map
↪  AssociationToODNavigationPropertyBinding();
}
```

La funzione **DataTypeToODComplexType()** trasforma un oggetto DataType in un oggetto ODComplexType.
Esattamente come per la funzione ClassToODEntityType(), se il datatype è astratto il valore abstract viene impostato al valore true e se esso ha una generalizzazione viene creato un tipo ODComplexType che sarà il tipo base dell'oggetto risultante.
Inoltre al tipo ODComplexType risultante viene associato un insieme di ODProperty attraverso la mappatura AttributeToODProperty().
Attraverso la funzione **EnumerationToODEnumType()** viene creato un oggetto ODEnumType da un oggetto Enumeration assegnandovi un nome e un insieme di ODMember attraverso la mappatura **EnumerationLiteralToODMember()** che trasforma gli EnumerationLiteral dell'enumerazione in ODMember.

```
mapping DataType::DataTypeToODComplexType() : ODComplexType{
//ct.name ← dt.name
result.name := self.name;
//if ct.abstract = true then dt.abstract ← true
if (self.isAbstract = true) result._abstract := true;
//if dt.generalizations contains a data type dd then ct.baseType ←
↪  t where t is the corresponding ODComplexType of dd
if (self.parent <> null) result.baseType := self.parent.map
↪  DataTypeToODComplexType();
//ct.properties ← (cf. rules to transform attributes, i.e., rows 3
↪  and 4)
//p is a class attribute or a data type attribute
result.properties := self.attribute.map AttributeToODProperty();

}

mapping Enumeration::EnumerationToODEnumType() : ODEnumType{
//oe.name ← e.name
result.name := self.name;
//oe.members ← (cf. rule to transform literals, i.e., row 7)
result.odmember := self.enumerationliteral.map
↪  EnumerationLiteralToODMember();
}

mapping EnumerationLiteral::EnumerationLiteralToODMember() :
↪  ODMember{
//om.name ← el.name
result.name := self.literalName;
}
```

La funzione **AttributeToODProperty()** trasforma un attributo in un oggetto ODProperty, utilizzando una specifica mappatura a seconda che si tratti dell'attributo di una classe, di un datatype o di un'enumerazione.
La funzione **AttributeToODPropertyKeyRef()** trasforma un attributo chiave di una classe in un oggetto ODPropertyKeyRef.

```
mapping Attribute::AttributeToODProperty() : ODProperty{
//op.name ← p.name
result.name := self.name;
//op.type ← t where t is the corresponding type of the attribute
if(self.type.metaClassName() = "DataType") result.type :=
↪  self.type.oclAsType(DataType).map DataTypeToODComplexType();
if(self.type.metaClassName() = "PrimitiveType") result.type :=
↪  self.type.map TypeToODPrimitiveType();
if(self.type.metaClassName() = "Enumeration") result.type :=
↪  self.type.oclAsType(Enumeration).map EnumerationToODEnumType();
//if p is multivalued then op.multivalued ← true
if (self.multivalue) result.multivalue := true;
}

mapping Attribute::AttributeToODPropertyKeyRef() : ODPropertyKeyRef {
//pk.property = op
result._property := self.resolveone(ODProperty);
}
```

AssociationToODNavigationProperty() mappa un'istanza Association in un'istanza ODNavigationProperty alla quale assegna un tipo ODEntityType e un valore booleano (containsTarget) uguale a true se l'associazione è una composizione.

```
mapping Association::AssociationToODNavigationProperty() :
↪  ODNavigationProperty {
//np.name ← p.name
result.name := self.name;
//np.type ← t where t is the corresponding entity type of p.type
result.type := self.type.map TypeToODEntityType();
//if p.aggregation = Composite then np.containsTarget ← true
if (self.isComposition) result.containsTarget := true;
//if p is multivalued is then np.multivalued ← true
if (self.multivalue) result.multivalue := true;
}
```

Allo stesso modo **AssociationToODNavigationPropertyBinding()** trasforma un oggetto Association in un'oggetto ODNavigationPropertyBinding, dove il path dell'istanza risultante sarà il nome dell'associazione e il target il nome dell'ODEntitySet del tipo dell'associazione.

```
mapping Association::AssociationToODNavigationPropertyBinding() :
 ↪  ODNavigationPropertyBinding {
//npb.path ← p.name
result.path := self.name;
//npb.target ← t.name where t is the corresponding entity set of
 ↪  p.type
result.target := self.type.map TypeToODEntitySet().name;
}
```

Le funzioni **TypeToODPrimitiveType()**, **TypeToODEntityType()** e **TypeToODEntitySet()** sono state implementate per trasformare i tipi degli attributi e delle associazioni.

```
mapping Type::TypeToODPrimitiveType() : ODPrimitiveType{
result.name := self.name;
}

mapping Type::TypeToODEntityType() : ODEntityType{
result.name := self.name;
}

mapping Type::TypeToODEntitySet() : ODEntitySet{
result.name := self.name;
result.entitytype := self.resolveone(ODEntityType);
}


}
```

4 Testing

Per verificare la correttezza della trasformazione **UMLToOData**, quest'ultima è stata eseguita con diversi modelli UML di input, definiti in modo tale da coprire tutti i casi possibili.
Di seguito sono illustrati i modelli UML utilizzati e i rispettivi risultati.

4.1 Product-Supplier Model

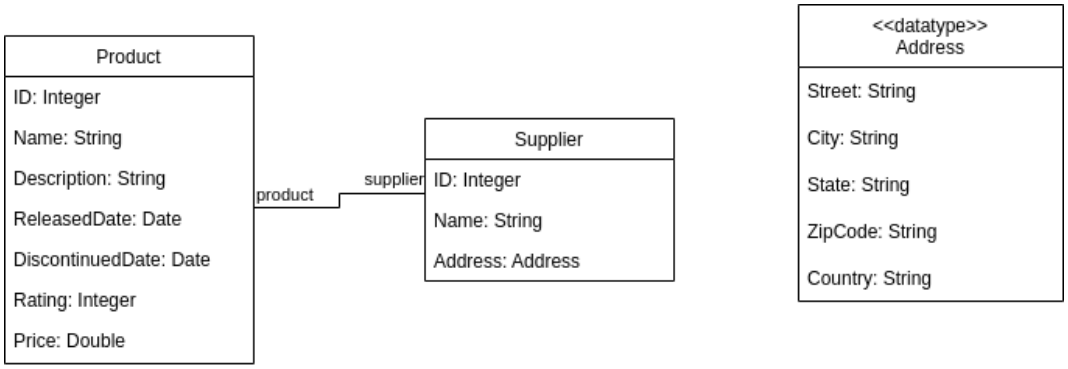

Figure 4.1: Modello UML Product-Supplier

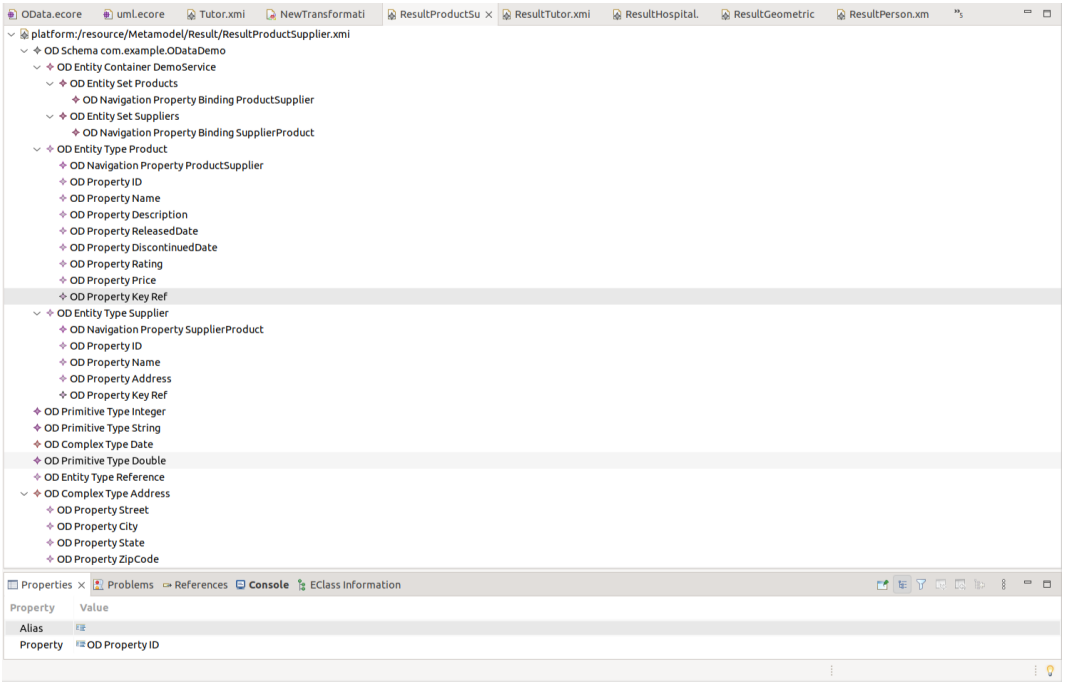

Figure 4.2: Modello OData ottenuto in EMF dal modello Product-Supplier

4.2 Tutor Model

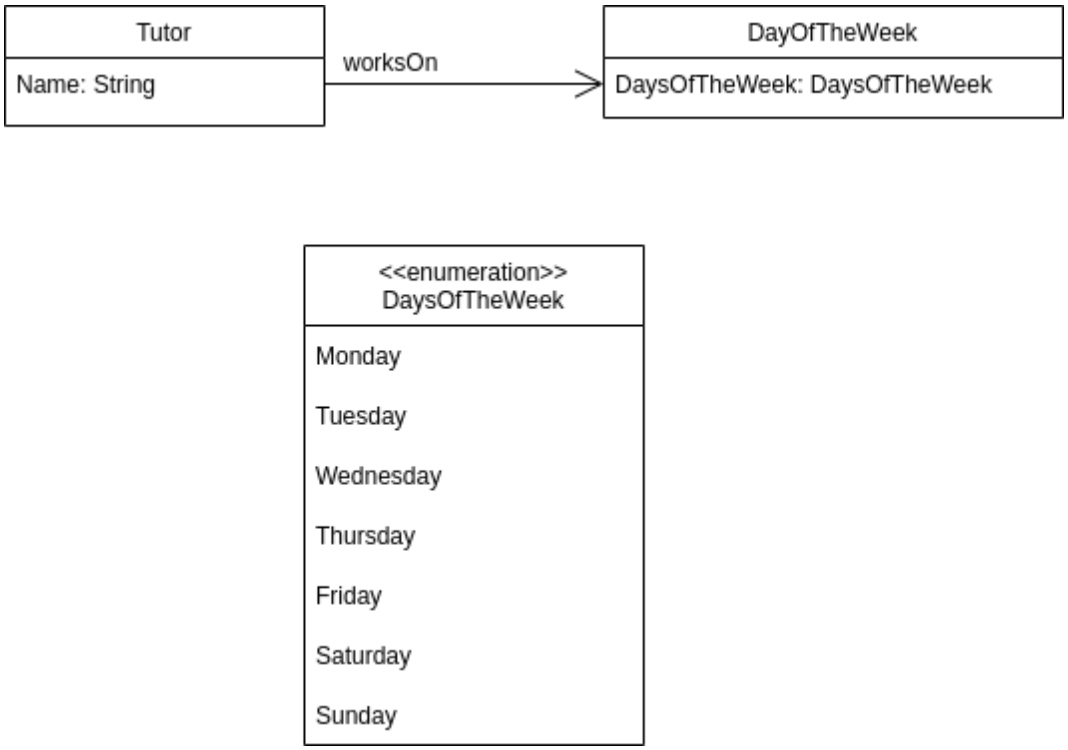

Figure 4.3: Modello UML Tutor

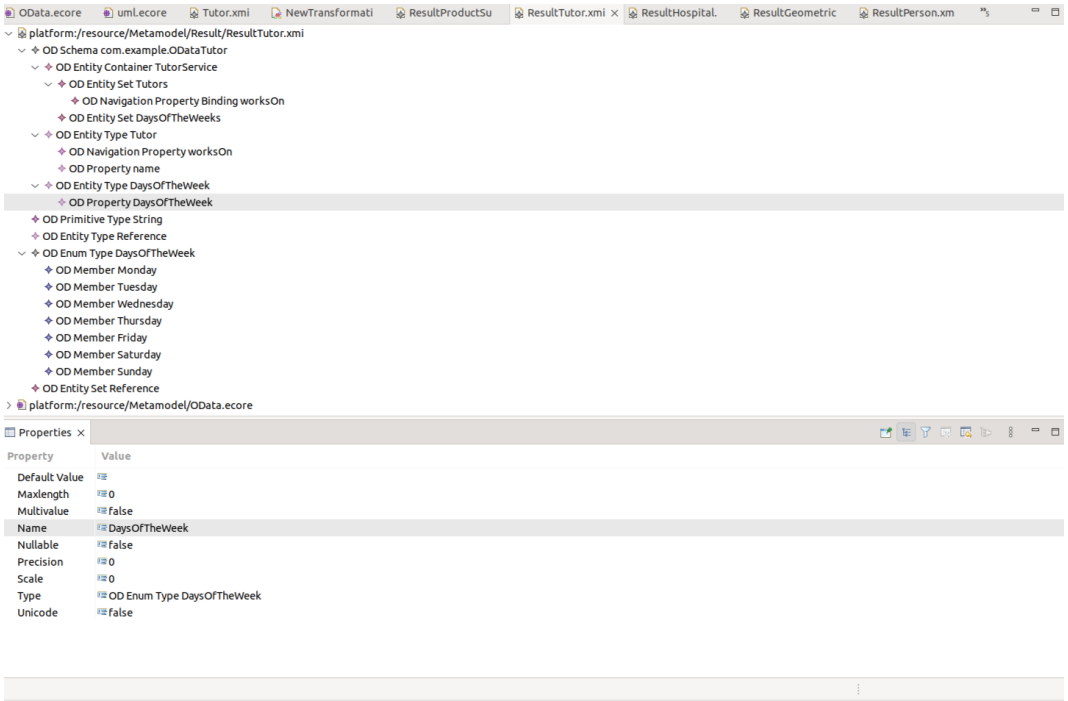

Figure 4.4: Modello OData ottenuto in EMF dal modello Tutor

4.3 Geometric Shapes Model

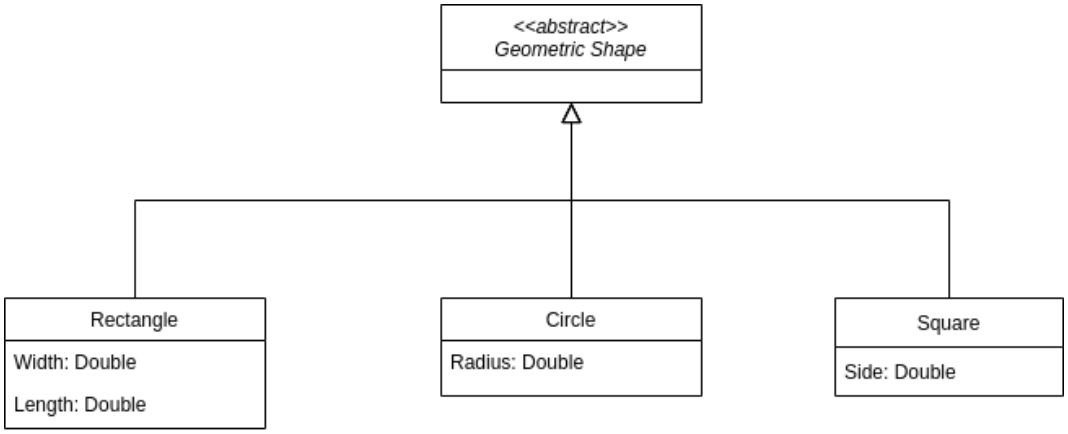

Figure 4.5: Modello UML Geometric Shapes

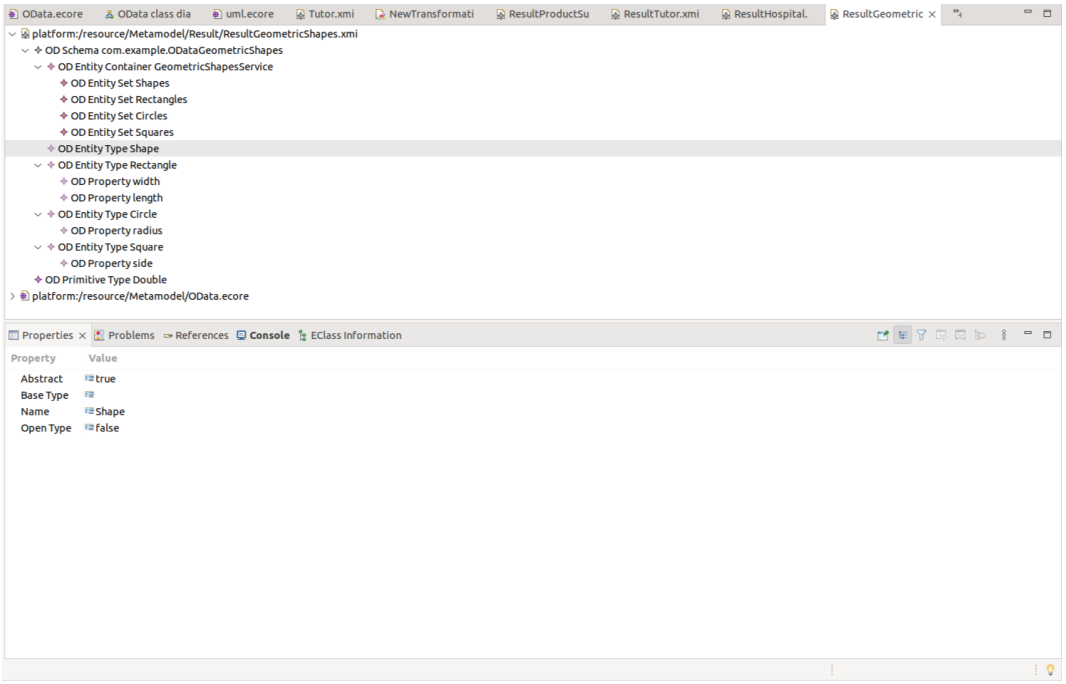

Figure 4.6: Modello OData ottenuto in EMF dal modello GeometricShapes

4.4 Hospital Model

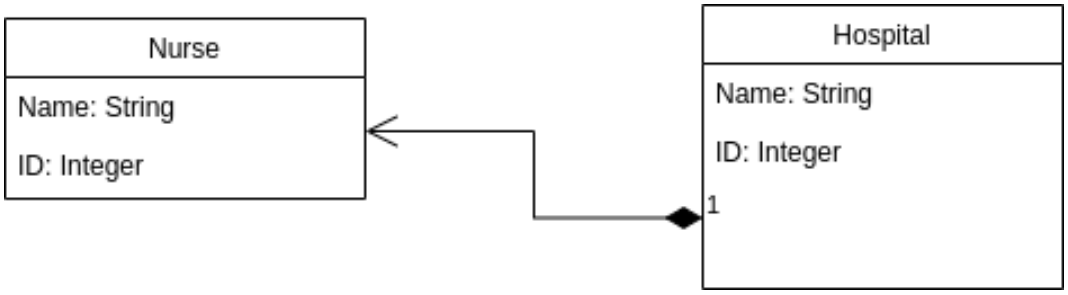

Figure 4.7: Modello UML Hospital

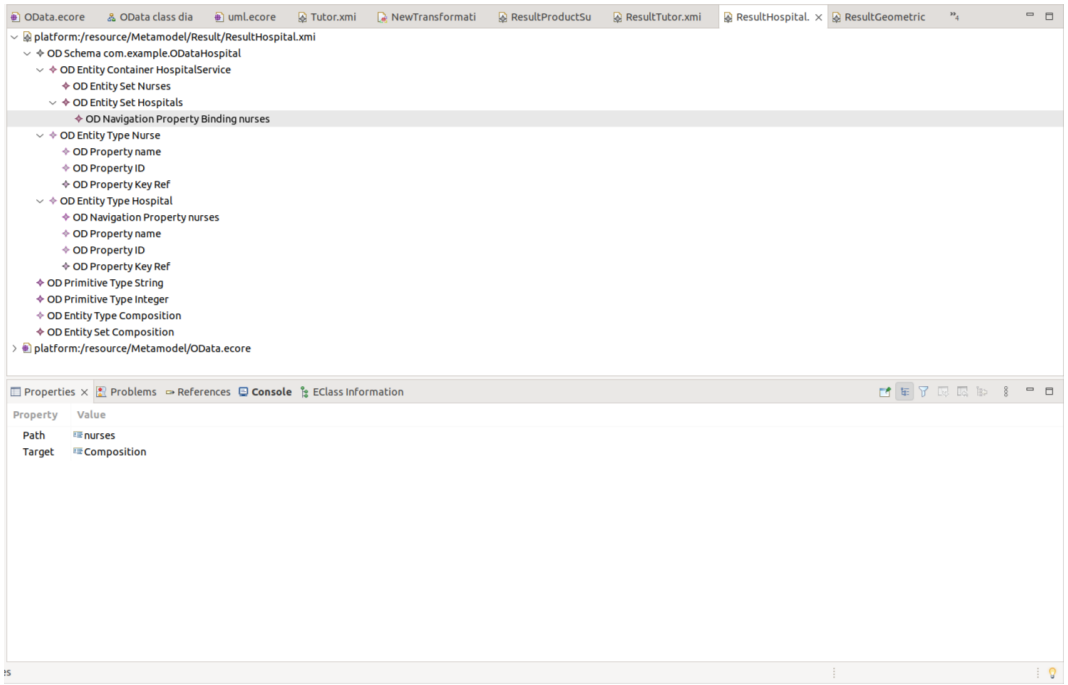

Figure 4.8: Modello OData ottenuto in EMF dal modello Hospital

4.5 Person Model

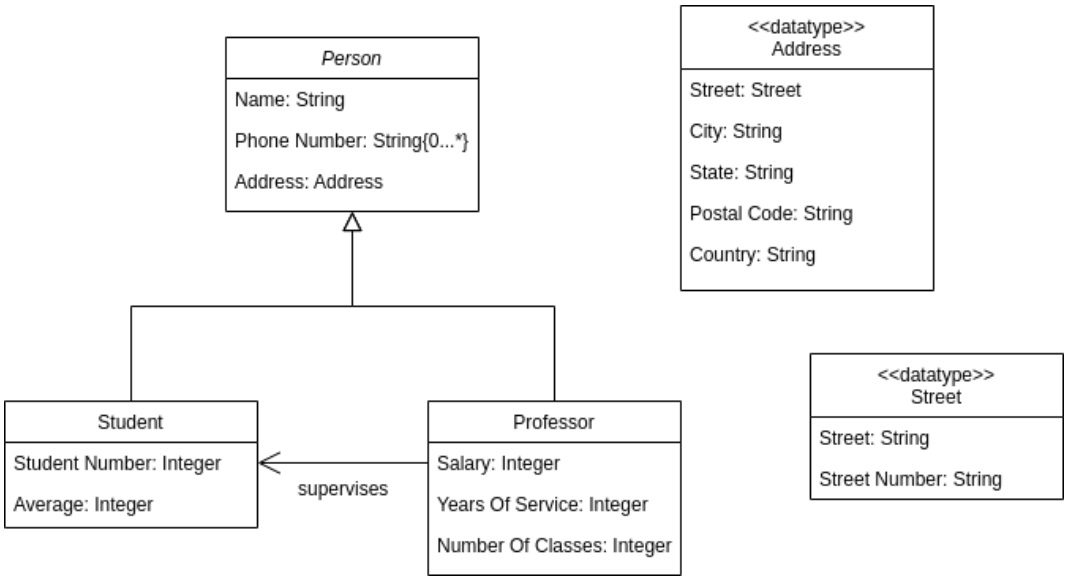

Figure 4.9: Modello UML Person

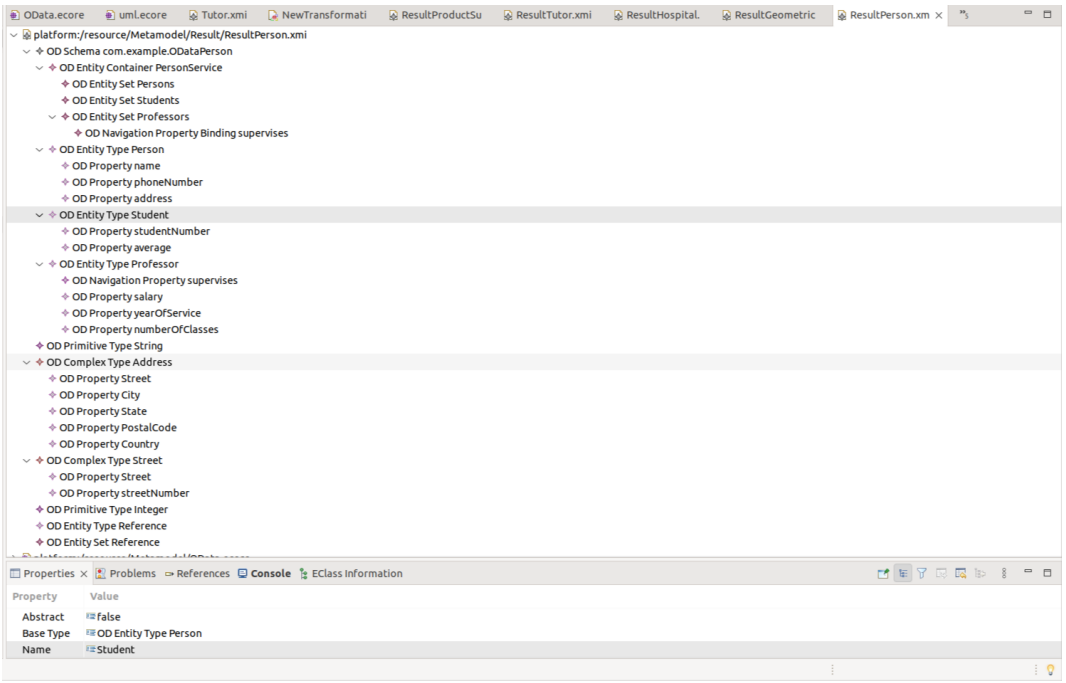

Figure 4.10: Modello OData ottenuto in EMF dal modello Person

5 Instructions for use

Aprire il progetto in Eclipse Modeling Project e nella cartella UMLToOData fare click con il tasto sinistro sul file NewTransformation.qvto, scegliere Run As e di seguito Run Configurations.
Nella finestra che si apre inserire i parametri della trasformazione:

- IN source: selezionare il path del file XMI che si vuole eseguire tra quelli presenti nella cartella *Metamodel→Model*;

- OUT target: selezionare il path del file XMI che si vuole generare.

Per ogni evenienza nella cartella *Metamodel→Result* sono contenuti i risultati della trasformazione dei modelli UML presenti nella cartella *Metamodel→Model*.

Se si vuole creare un nuovo modello UML da dare in input alla trasformazione aprire il file *Metamodel→uml.ecore* e fare click con il tasto sinistro sulla EClass Model e scegliere Create Dynamic Istance.
Questo genererà un file XMI all'interno del quale sarà contenuta un'istanza Model. Cliccare con il tasto sinistro su Model e cliccare su NewChild per creare le classi, le associazioni e i tipi del modello.