# Trasformazione di modelli UML2ODATA con approccio Model-Driven-Engineering

Federico Ranaldi

February 1, 2024

## 1 Introduzione

In questo lavoro, viene descritto il processo di trasformazione automatica tra due Entity Data Models (EDM), con particolare attenzione al passaggio da modelli UML (Unified Modeling Language) a modelli OData (Open Data Protocol). OData si presenta come un linguaggio e un protocollo per la rappresentazione di dati sul web, offrendo un'alternativa al Web semantico implementato con RDF (Resource Description Framework). La trasformazione è eseguita utilizzando il linguaggio QVT Operational (QVTo), un framework utile per la trasformazione model-to-model all'interno dell'Eclipse Modeling Framework (EMF).

Il processo inizia con la definizione delle corrispondenze tra gli elementi dei metamodello UML e OData, identificando come classi, associazioni e attributi in UML possano essere rappresentati come entità, relazioni e proprietà in OData. QVTo facilita la mappatura di questi elementi grazie alla sua capacità di manipolare modelli astratti e concretizzare la trasformazione in modo efficace e accurato.

La trasformazione automatica sfrutta le potenzialità di QVTo per interpretare i modelli UML e generare equivalenti strutture OData, garantendo così che la rappresentazione dei dati rimanga coerente e fedele al modello originale. Questo approccio si rivela particolarmente utile per applicazioni che richiedono una rappresentazione dei dati flessibile e adattabile ai servizi web, massimizzando l'interoperabilità e l'efficienza nella gestione dei dati.

Nonostante la semplicità di QVTo, nel processo di trasformazione tra due modelli può risultare molto complesso capire quali sono le coppie di elementi appartenenti ai due modelli usati per rappresentare entità semanticamente e concettualmente simili. In casi peggiori come quello che verrà trattato in questo lavoro, ci si può trovare davanti a EDMs che implementano paradigmi di rappresentazione dei dati differenti.Non riuscendo a trovare tutte le possibili corrispondenze, bisogna ideare un modo per far corrispondere una determinata parte del primo linguaggio di rappresetazione a una determinata parte del secondo evitando che in fase di trasformazione vengano persi dei dati che non possono essere mappati.

Nel corso della trattazione si farà riferimento al lavoro [EdIC18] che propone una serie di specifiche per la trasformazione di servizi dal loro formato originale (in particolare si considerano dati di partenza rapprensentati con il paradigma Entity-Relationship) a dati elaborabili con il protocollo OData.

Verrà tratto spunto dalla parte dell'articolo in cui si descrive la trasformazione da UML a OData proponendo delle specifiche differenti ma che comunque portano ad una soluzione efficiente e accettabile escludendo alcuni dettagli.

L'efficienza della trasformazione sarà infine mostrata con dei semplici modelli di esempio UML che verranno mappati attraverso essa nei rispettivi modelli OData.

## 2 Metamodelli

### 2.1 Estremi della trasformazione

OData è un protocollo per la creazione di servizi Web, ma in questo lavoro faremo riferimento esclusivamente al linguaggio per la rappresentazione dei dati che esseo prevede, mettendolo sullo stesso piano di UML. In un contesto di Model-Driven Engineering (MDE), la trasformazione trattata ha come sorgente metamodello UML e come destinazione il metamodello OData. Un metamodello non è altro che un

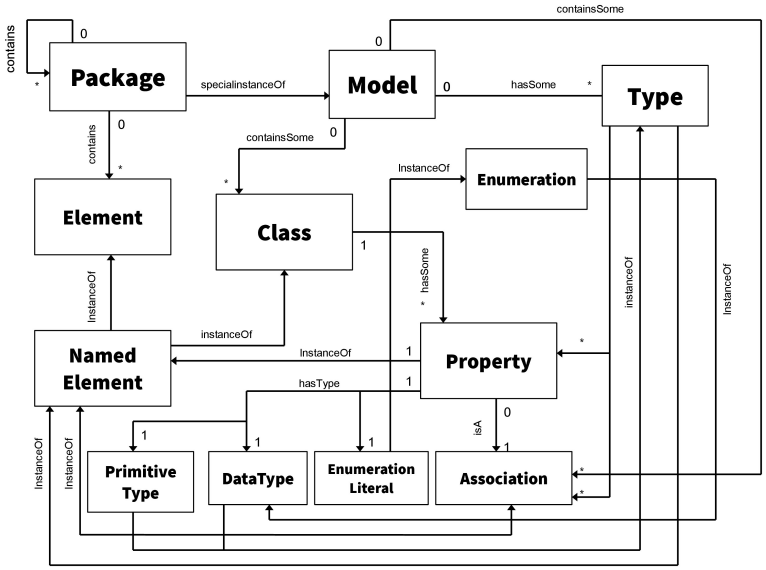
linguaggio che definisce come rappresentare vari tipi di entità e collegamenti all'interno di un modello. Pertanto, un modello UML rappresenta un dominio specifico utilizzando il linguaggio e le strutture fornite dal metamodello UML. Analogamente, un modello OData è strutturato secondo le specifiche del metamodello OData. La trasformazione descritta in seguito illustra come i dati e le strutture in un modello UML possono essere reinterpretati e mappati in maniera coerente in un modello OData, seguendo le regole e le convenzioni dei rispettivi metamodelli. Dal momento che vogliamo ottenere in maniera automatica un modello OData a partire da un modello UML è necessario capire quale siano i componenti e le relazioni dei rispettivi metamodelli. In questa sezione vengono analizzate in maniera approfondita delle porzioni dei metamodelli UML e OData. Ci si concentrerà su delle porzioni perché i due metamodelli sono molto complessi (in accordo al loro ruolo ; cioè quello di fornire specifiche per la creazione di un modello) e una loro descrizione esaustiva richiederebbe troppo tempo. Lo scopo di questo lavoro è quello di mostrare come può essere effettuata una trasformazione tra due metamodelli piuttosto differenti.

## 2.2 UML

Dal momento che Eclipse Modeling Framework (EMF) include solo la seconda versione del metamodello UML, ovvero UML 2.0, la sorgente della trasformazione sarà proprio quest'ultimo.

Introdotto nel 2005, UML 2.0 ha apportato notevoli cambiamenti rispetto alla sua versione precedente, rendendo il sistema di modellazione più complesso ma allo stesso tempo fornendo maggior potenza e flessibilità nella rappresentazione dei dati. UML 2.0 introduce miglioramenti significativi in termini di dettaglio e profondità della modellazione (come prototipi,profili e interfacce), permettendo una rappresentazione più accurata e articolata dei modelli. Tuttavia, per lo scopo di questo lavoro, non verranno trattate le specifiche novità introdotte con UML 2.0 in dettaglio, poiché gli esempi di modelli UML su cui testare la trasformazione sono relativamente semplici. L'accento verrà infatti posto sulle scelte adottate nel processo di trasformazione stesso, mantenendo la complessità a un livello gestibile. In link è possibile visualizzare la struttura arborea del metamodello UML che mette in relazione i vari componenti di UML2 includendo una descrizione di questi, i componenti da cui derivano e i componenti che estendono (rispettivamente superclassi e sottoclassi secondo un'ottica Object Oriented) inclusi i relativi metodi e attributi (anche quelli ereditati). Non essendo reperibile sul web una rappresentazione visuale in forma di diagramma che sia esaustiva per i nostri scopi in 1 è mostrata una pseudo-rappresentazione che nasconde i dettagli che non interessanti ai fini della comprensione di quanto verrà mostrato nelle prossime sezioni.

Figure 1: "Pseudo-Representation" of UML2 Metamodel portion.



Senza entrare troppo nel merito del metamodello di UML2 che è molto più complesso di quello di mostrato in figura , in seguito alcuni chiarimenti discorsivi sulle classi del metamodello e le relazioni tra esse.

Si osserva metamodello UML2, l'elemento principale è rappresentato dalla classe "Package", un contenitore flessibile in grado di includere vari tipi di elementi definiti come *Packageable Elements*. Questi elementi possono essere ulteriori Packages, istanze della classe "Model", e istanze della classe "NamedElement". La classe "NamedElement" è un caso particolare di "Element", caratterizzata dalla presenza di un nome che identifica l'entità rappresentata. Grazie al principio di ereditarietà, molte altre classi nel metamodello UML2 sono istanze di "NamedElement".

Concentrandoci sulla classe "Model", si noota che essa funge da contenitore per elementi utili alla rappresentazione di un dominio specifico (che come si può immaaginare saranno entità e relazioni tra esse). Un istanza di "Model" contiene quindi istanze della classe "Class", utilizzate per rappresentare entità (o classi) all'interno del dominio modellato. Come anticipato, ogni istanza di "Class" è anch'essa un'istanza di "NamedElement".

La classe "Class" è caratterizzata da una o più istanze della classe "Property", che definiscono le caratteristiche o gli attributi della classe. Alle istanze di "Property" sono attribuiti vari tipi tramite la relazione "*hasType*", che le associa a "PrimitiveType", "DataType" ed "EnumerationLiteral", tutti derivati dalla classe "Type". "PrimitiveType" include tipi primitivi (come Integer, Boolean, String ...) mentre "DataType" rappresenta tipi di dati complessi che possono essere definiti da chi progetta il modello.Quest'ultimo può essere rappresentato da un insieme di proprietà che hanno tipo primitivo oppure (ricorsivamente) anche tipo complesso (quindi istanza di "DataType"). "EnumerationLiteral" è invece un caso speciale di "Enumeration" (anch'essa istanza di Type) che rappresenta un insieme di valori attribuiti a delle variabili fissate. Il concetto di *Literal* richiama a quello di proprietà ma è meno complesso.

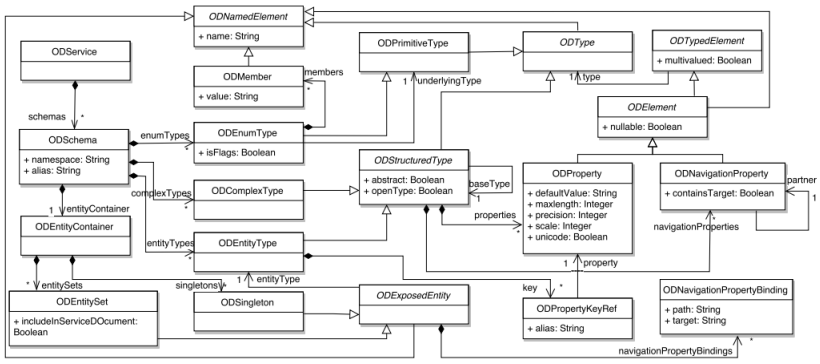
Le istanze di "Class" possono anche essere collegate tra di loro mediante relazioni. Queste relazioni sono definite come istanze di "Association", che all'interno di "Model" sono elementi a sé stanti e come già detto servono a creare relazioni tra diverse classi (istanze di "Class"). In UML2, una relazione tra due classi è rappresentata tramite proprietà (istanze di "Property") che fanno riferimento a un'associazione (istanza di "Association") definita indipendentemente dalle classi stesse. Per chiarire quest'utlimo aspetto cruciale poi per parte della trasformazione , si ha che le associazioni(istanze di "Association") in UML2 possono essere definite senza essere usate e per usarle servono delle proprietà(istanze di "Property") che facciano riferimento ad esse.

Comprendere la struttura e le interrelazioni tra questi elementi è fondamentale per effettuare la trasformazione di modelli UML2 in altri formati, poiché offre una base per mappare correttamente le caratteristiche e le relazioni del modello originale.

## 2.3 OData

Prendendo spunto da [EdIC18] una porzione del metamodello di OData è quella rappresentata in 2.

Figure 2: Representation of OData Metamodel portion.




Volendo fornire una riesamina più discorsiva di questo metamodello si ha che l'elemento centrale è l'ODService, che funge da contenitore principale e include un insieme di schemi. Questi schemi,

rappresentati dalla classe ODSchema, sono cruciali per definire il modello di dati del servizio OData. Ogni schema in OData ha un namespace e un alias e può includere diversi tipi di strutture dati come enumerazioni, tipi complessi e tipi di entità.

La classe "ODType", che è astratta, agisce come una base comune per le diverse strutture di dati. Ad esempio, le enumerazioni in OData sono rappresentate dalla classe "ODEnumType", che è una specializzazione di "ODPrimitiveType". La classe "ODEnumType" include dettagli come il nome, le opzioni di multiselezione, il tipo primitivo sottostante e un elenco di membri. Ogni membro, definito dalla classe "ODMember" rappresenta un *Literal* (come si vedrà nella trasformazione) il cui valore è dato da una stringa. Quindi sipuò già anticipare che gli "ODEnumType" corrispondono concettualmente agli "EnumeratioLiteral" di UML2. Per quanto riguarda la rappresentazione delle entità, si istaziano due classi principali: "ODEntityType" per le entità con chiave e "ODComplexType" per i tipi complessi senza chiave. Volendo fare un parallelismo con UML2 si può pensare che questi corrispondano rispettivamente a "Class" e "DataType".Sia "ODEntityType" che "ODComplexType" derivano da "ODStructuredType", una classe che rappresenta un elemento strutturato. Tale elemento, essendo "ODStructuredType" essendo sottoclasse di "ODType" che a sua volta è sottoclasse di "ODNamedElement" include un nome come attributo ereditato da quest'utlima classe. Inoltre ODStructuredType è caratterizzato da due attributi che indicano se il tipo è astratto ("*abstract:boolean*") e se permette proprietà non dichiarate ("*abstract:boolean*") quindi ereditate. Gli "ODStructuredType" hanno proprietà che li definiscono (istanze di "ODProperty") proprietà di navigazione (istanze di "ODNavigationProperty") che li collegano con alte istanze della loro stessa classe. Sostanzialmente le "ODNavigationProperty" sono proprietà che consentono di mettere in relazione due istanze di "ODStructuredType" e sono anch'esse derivate dalla classe "ODNamedElement". In particolare, ODNavigationProperty aggiunge attributi per il contenimento e la proprietà di navigazione opposta. Le "ODProperty" per ereditarietà hanno come attributo principale il tipo di dato queste rappresentano che come si immagina può essere primitivo (istanza di "ODrimitiveType") o complesso (istanza di ODCOmplexType). Nell'istanziare un "ODStructuredType" si definisce per essero una o più istanze di "ODPropertyKeyRef", una classe speciale che consente di risalire alle istanze di "ODProperty" che per quel determinato ODStructuredType rappresentano un identificativo(chiave) . Infine, tipico dei servizi OData e utile al loro impiego, l'elemento "ODEntityContainer" definisce il contenitore di entità del servizio, specificando i set di entità ("EntitySet") e i singleton ("ODSingleton") che possono essere interrogati e aggiornati. "ODEntitySet" e "ODSingleton" sono sottoclassi di ODExposedEntity, che includono riferimenti al tipo di entità target e vincoli di proprietà di navigazione.

## 2.4 Strumenti usati per la trasformazione

Per realizzare un prototipo in grado di trasformare modelli da UML2 a OData, è stato utilizzato Eclipse Modeling Framework (EMF) come ambiente di sviluppo. EMF offre un'ampia gamma di funzionalità e packages (tra cui il metamodello UML2) che sono particolarmente utili nella costruzione di metamodelli e, di conseguenza, nella creazione di modelli basati su questi metamodelli. Uno dei passaggi chiave del lavoro è stato quello di rappresentare il metamodello OData nel formato ECORE (3), un formato standard utilizzato all'interno di EMF per definire metamodelli. Per la realizzazione del codice capace di eseguire la trasformazione tra i due modelli, abbiamo impiegato il linguaggio QVTo (Query,View,Transformation Operational), che è anch'esso parte integrante dell'ecosistema EMF. In seguito verranno brevemente descritti gli aspetti fondamentali del linguaggio QVTo per poi effettivamente mostrare la trasformazione UML2OData.

## 2.5 Breve panoramica sul linguaggio QVTo

QVTo, acronimo di Query/View/Transformation Operational, è un linguaggio di programmazione specializzato nella trasformazione di modelli in ambito MDE. Fa parte delle specifiche MOF 2.0 QVT (maggiori informazioni su link) stabilite dall'Object Management Group (OMG). Questo linguaggio è stato sviluppato per permettere la trasformazione efficiente e accurata di modelli da una forma a un'altra, una pratica comune nell'ingegneria del software moderna. Riassumentìdo revemente, il paradigma di QVTo si articola nei seguenti passaggi principali:

1. Definizione dei (Meta)modelli di Input e Output: Vengono definiti i metamodelli su cui dovranno basarsi i modelli di input (in questo lavoro UML2) e i modelli di output (in questo lavoro OData).

Nell'eseguire una trasormazione specifica di un modello A (in UML2) in un modello B (in OData) verrà dato in pasto al programma che effettua la trasformazione solo il primo dei due. Sarà poi la trasformazione a produrre il modello B in accordo alle regole definite.

2. Specifica delle Regole di Trasformazione: Si designano delle regole specifiche in QVTo che descrivono come gli elementi del modello di input devono essere trasformati negli elementi del modello di output. Come già accennato , dal momento che la trasformazione deve avvenire per qualsiasi modello che abbia i metamodelli di Input e Output quello che si fa è andare a studiare la struttura generale dei metamodelli stessi piuttosto che i modelli di I/O specifici. Inoltre ,quando si ha a che fare con modelli di input e output basati su metamodelli molto diversi, questa è la fase più tediosa.

3. Esecuzione della Trasformazione: Il motore di trasformazione QVTo elabora i modelli di input seguendo le regole definite, producendo i modelli di output (su questa parte c'è un piccolo dettaglio implementativo che verrà chiarito in seguito **??**).

Per ricapitolare , seppure in fase di esecuzione si considera un modello specifico da trasformare da formato UML a formato OData, le regole di trasformazione definte hanno una validità generale. Pertanto la trasformazione potrà essere applicata a qualsiasi modello. Verrà infatti mostrata, nella prossima sezione , come lo stesso programma di trasformazione viene eseguito su diversi modelli di esempio.

## 2.6 Trasformazione

In questa sottosezione viene trattata la parte più importante dell'intero lavoro, la trasformazione UML2OData. Verrà descritto ogni step della trasformazione riportando il codice in linguaggio QVTo.

Listing 1: Step 1.

```
modeltype UML2 'strict' uses 'http://www.eclipse.org/uml2/2.0.0/UML';
modeltype OData 'strict' uses OData('http://odatamodel.com');

transformation Transformation(in source: UML2, out target: OData);
```

In 1 vengono importati in ordine il metamodello UML2 e il metamodello OData (precedentemente creato in standard ECORE). Dopodiché si inizializza un oggetto di tipo "transormation" che sarà la funzione che esegue la trasformazione. Questa funzione ha come parametri modello di input e modello di output della trasformazione; quindi rispettivamente un modello basato sul metamodello UML2 e uno basato su OData.

Listing 2: Step 2.

```
main() {

source.rootObjects()[Model] -> map ModelToODSchema();

}
```

In 2, la funzione "main" è usata per definire il punto di partenza della trasformazione, cioè la classe "Model" di UML2, che si è scelto di mappare nella classe "ODSchema" di OData. Si osserva che il metodo *rootObjects()* consente di accedere all'elemento radice del modello a cui questo è applicato. Dunque l'elemento "Model" viene mappato nell'elemento "ODSchema" attraverso la funzione *ModelToODSchema()*.

Listing 3: Step 3.

```
mapping Model::ModelToODSchema() : ODSchema{

result.namespace := "com.example.OData." + self.name;

result.alias := self.name;
```

```
source.rootObjects()[Package] -> PackageToODEntityContainer();

result.odcomplextype := self.packagedElement[DataType].map DataTypeToODComplexType();

result.entitytypes := self.packagedElement[Class].map ClassToODEntityType();

result.odenumtype := self.packagedElement[Enumeration].map EnumerationToODEnumType();

self.packagedElement[Association].map AssociationToODNavigationPropertyBinding();

}
```

In 3 è mostrato il codice della funzione *ModelToODSchema()* che effettua un mapping semplice per le variabili di classe "namespace" e "alias" del modello di output (result.) assegnando a queste gli elementi corrispondenti del modello di input (self.) . Successivamente alla classe "ODEntityContainer" viene mappata la classe "Package" del metamodello UML2 usando la funzione **??**. Seppure "Package" in UML2 rappresenta una classe concenttualmente molto più importante e complessa di ODEntityContainer in OData si è scelto comunque di fare questo mapping. Si vedrà poi nello sviluppo della funzione **??** che questa parte della trasformazione è servita solo per inizializzare un oggetto di tipo "ODEntityContainer" nel modello di output. Lla classe "ODComplexType" collegata alla classe "ODSchema" tramite la relazione "–odcomplextype–", viene fatta corrispondere la classe "DataType" del modello di input mediante la funzione *DataTypeToODComplexType()*. La classe "ODEntityType" collegata alla classe "ODSchema" tramite la relazione "–entitytypes–", viene fatta corrispondere alla classe "Class" di UML2 mediante la funzione *ClassToODEntityType()*. La classe "ODNavigationPropertyBinding" collegata (non direttamente) alla classe "ODSchema", viene fatta corrispondere alla classe "Association" di UML2 mediante la funzione *AssociationToODNavigationPropertyBinding()*.

Listing 4: Step 4.

```
mapping Package::PackageToODEntityContainer() : ODEntityContainer{

result.entitysets := self.packagedElement[Class].map ClassToODEntitySet();

}
```

In 4 è presente il codice della funzione utile al mapping di "Package" verso "ODEntityContainer". La classe "ODEntitySet" collegata alla classe "ODEntityContainer" tramite la relazione "–entitysets–", viene fatta corrispondere alla classe "Class" di UML2 mediante la funzione *ClassToODEntitySet()*.

Listing 5: Step 5.

```
mapping Class::ClassToODEntityType() : ODEntityType {

// IMPORTANT INFO: ODEntityType is a ODStructuredType and thus it inherits
    generalization's attributes
// IMPORTANT INFO: ODEntityType is a ODType and thus it inherits generalization's
    attributes

result.name := self.name ;

var AllAttributes : OrderedSet(Property); //all properties (included IDs)

self.ownedAttribute->forEach(i){if (i.association = null) AllAttributes+=i;};

result.properties := AllAttributes.map PropertyToODProperty();

var IDAttributes : OrderedSet(Property);        //properties used as ID

self.ownedAttribute->forEach(i){if (i.isID) IDAttributes+=i;};

result.propertykeyref := IDAttributes.map PropertyToODPropertyKeyRef();

var associationAttributes : OrderedSet(Property);

self.ownedAttribute->forEach(i){if (i.association != null) associationAttributes+=i
    ;};
```

```
result.navigationproperties := associationAttributes.map
    PropertyToODNavigationProperty();

}
```

Come si vede in 5, "Class" e "ODEntityType" sembrano essere classi simili di metamodelli differenti. Infatti entrambi possiedono un nome (il cui valore viene mappato) e una serie di attributi (o proprietà) che le qualificano. Quindi si assegnano le proprietà del modello di input ottenute tramite la funzione *ownedAttribute()* al modello di output facendo le dovute distinzioni tra tipologie di proprietà e attenzione ad alcuni dettagli. Infatti mentre in UML2 tutte le proprietà sono casi particolari della classe "Property" , in OData si distingue tra "ODPropery" ed "ODNavigationProperty". Nel metamodello OData infatti , queste due sono classi a sé stanti che hanno due relazioni differenti con la classe "ODStructuredType" e sono inoltre derivate dalla classe "ODElement". Inoltre secondo le specifiche del metamodello OData per quelle proprietà il cui valore rappresenta un identificativo/chiave bisogna istanziare un oggetto supplementare della classe "ODPropertyKeyRef" collegato alla rispettiva "ODProperty". Dunque in 5 , vengono gestite tutte le proprietà della classe del modello UML2 di partenza e vengono istanza le "ODProperty" , "ODPropertyKeyRef" e "ODNavigationProperty". Scorrendo tra le proprietà della classe "Property" , si selezionano quelle proprietà escluse quelle che rappresentano chiavi esterne (che in OData sono le "ODNavigationProperty") e vengono create delle rispettive istanze di "ODproperty" gestite con la funzione *PropertyToODProperty()*. Per quelle particolari proprietà che rappresentano chiavi primarie (individuate grazie all'attributo booleano *isID* di "Property"), si va a instanziare un oggetto "ODPropertyKeyRef" i cui attributi sono gestiti con la funzione *PropertyToODPropertyKeyRef()*. Infine per mappare le "Property" che rappresentano chiavi esterne nelle "ODNvigationProperty" si fa riferimento alle istanze della classe "Association" di UML2. Come già accennato la classe "Association" che non è direttamente collegata a "Class" serve a definire un'associazione che sussiste tra due classi. Quindi in UML2 quando si vuole creare una "Property" che sia una chiave esterna bisogna collegarla alla relativa istanza di "Association". Questo collegamento è reso possibile grazie all'attributo *association* di "Property". Ciò spiega perché nella ricerca di tutte le altre tipologie di proprietà (escluse le chiavi esterne) si va a guardare la variabile *association* il cui valore deve essere uguale a Null. Ad occuparsi, una volta istanziate , delle "ODNavigationProperties" è la funzione *PropertyToODNavigationProperty()*. Si noti che questa è una delle parti più critiche della trasformazione perché prova a gestire le diversità più significative che ci sono tra i due metamodelli.

Listing 6: Step 6.

```
mapping Property::PropertyToODProperty() : ODProperty {

result.name := self.name;

var TypeName := self.type.name.toString();

if(TypeName != "String" and TypeName != "Integer" and TypeName != "Boolean" and
    TypeName != "Real"){

 result.type := self.type.resolveone(ODComplexType);

 }else{

    result.type := result.type := self.type.map TypeToODPrimitiveType();
 };

}

mapping Type::TypeToODPrimitiveType() : ODPrimitiveType{

        result.name := self.name;

}
```

In 6 è riportato il codice implementato per effettuare il mapping da "Property" a "ODProperty". Nel mappare il tipo di dato di "Property" che si ricava dall'attributo "Type", si è scelto di distinguere tra tipi primitivi e tipi complessi. Se il tipo della "Property" è un tipo primitivo (in particolare uno tra: Integer,String,Boolean,Real) la funzione *TypeToODPrimitiveType()* si occupa di mappare questo

in un "ODPrimitiveType". Se il tipo della "Property" è un "DataType" questo viene mappato in un oggetto della classe "ODComplexType" mediante la funzione *resolveone(ODComplexType)*. La funzione (SpecificType) si occupa di ricercare nella memoria del programma in esecuzione un oggetto che abbia come tipo specifico quello indicato come paramentro.

Listing 7: Step 7.

```
mapping Property::PropertyToODPropertyKeyRef() : ODPropertyKeyRef {
        result._property := self.resolveone(ODProperty);
        result.alias := self.name;
}
```

Osservando il metamodello OData , la classe "ODPropertyKeyRef" è collegata a "ODProperty" tramite la relazione "–property–". In 7 viene creata a partire da una "ODProperty" che rappresenta una chiave primaria la relativa istanza della classe "ODPropertyKeyRef" nel modello di output.

Listing 8: Step 8.

```
mapping Property::PropertyToODNavigationProperty() : ODNavigationProperty {

        result.name := self.name;

        if (self.aggregation != null) result.containsTarget := true;

        var TypeName := self.type.name.toString();

        result.type := self.type.map TypeToODPrimitiveType();

        var partnerProperty : Property;

        self.association.memberEnd->forEach(i) { if (i.namespace.name != self.name ){
            partnerProperty := i ; break;}};

        result._partner:=partnerProperty.resolveone(ODNavigationProperty);

        if(result._partner!=null){

                result._partner._partner := self.resolveone(ODNavigationProperty);

                }

}
```

In 8 vengono mappate le istanze di "Property" che rappresentano associazioni tra classi in istanze di "ODNavigationProperty". A partire da una "Property" di associazione possiamo risalire all'associazione stessa (istanza di "Association" attraverso il valore dell'attributo *association* di "Property". Una volta ottenuta l'istanza di "Association" si utilizzano i valori assunti dal suo attributo *MemberEnd*. Il valore di questo attributo è assunto da una coppia di namespace. I namespace(includeranno le classi) delle proprietà che rappresentano le estremità dell'associazione. Dunque elaborando il contenuto di *MemberEnd* si può risalire alle classi e quindi alle proprietà che coinvolge l'associazione. Attraverso la relazione "–partner–" si va a collegare una "ODNavigationProperty" di una classe alla relativa "ODNavigationProperty" della classe associata.

Listing 9: Step 9.

```
mapping DataType::DataTypeToODComplexType() : ODComplexType{

result.name := self.name;
result.baseType := self.resolveone(ODComplexType);
var ComplexAttributes : OrderedSet(Property);

result.properties := self.ownedAttribute.map PropertyToODProperty();

}
```

Essendo un "DataType" un particolare tipo di dato costituito da un insieme di proprietà a loro volta di vario tipo si può mappare questo in "ODComplexType" (illustrato in 10. Per fare ciò bisogna

prima individuare tutti gli attributi dell'istanza di "DataType" e mapparli in "ODProperty" mediante la già trattata *PropertyToODProperty()*. Le istanze di "ODProperty" contenute in un "ODComplexType" saranno collegate dalla relazione "–properties–". Quest'ultima relazione è accessbile da "ODComplexType" perché ereditata dalla classe madre "ODStructuredType".

Listing 10: Step 9.

```
mapping Enumeration::EnumerationToODEnumType() : ODEnumType{

        result.odmember := self.ownedLiteral.map EnumerationLiteralToODMember
            ();

}
mapping  EnumerationLiteral::EnumerationLiteralToODMember() : ODMember{

        result.name := self.name;

}
```

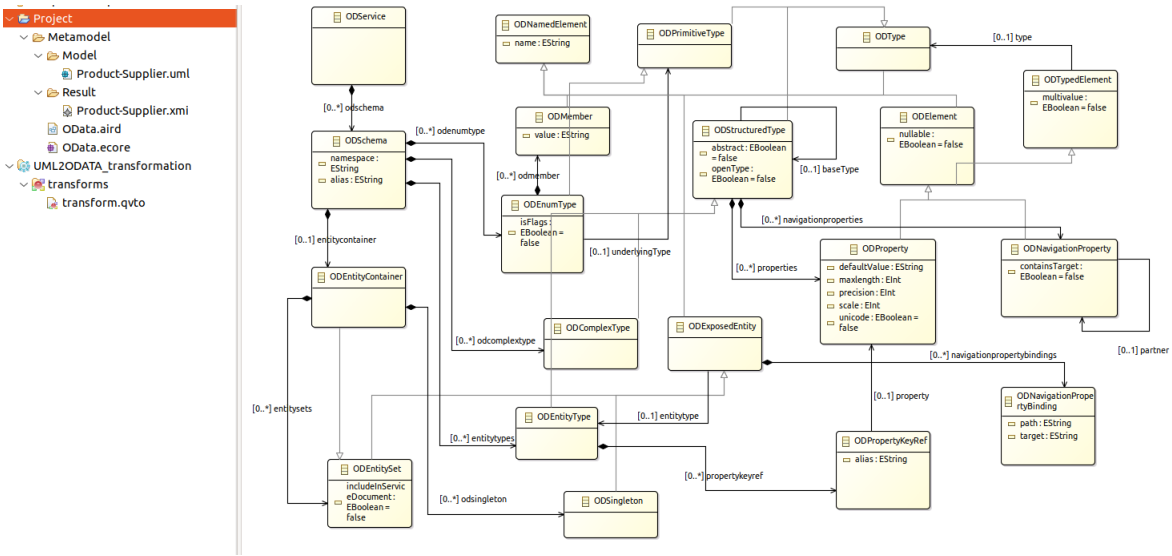
Il tipo "Enumeration" di UML2 viene mappato nel tipo "ODEnumType" di OData. Il contenuto di un "Enumeration" è rappresentato da un insieme di oggetti di tipo "EnumerationLiteral" e ottenibile mediante la funzion *ownedLiteral*. Dunque in questa parte della trasformazione vengono assegnati alla specifica istanza creata di "ODEnumType" i vari literals che verranno mappati come istanze di "ODMember".

Nel seguente link è presente l'intero progetto.

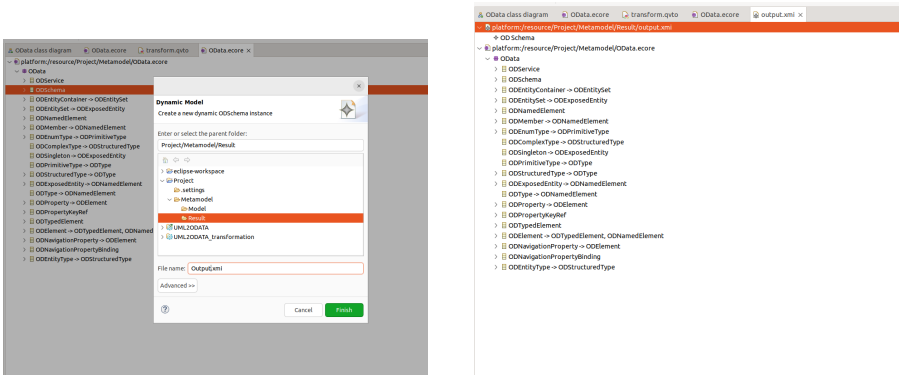
# 3 Istruzioni ed esempi di applicazione.

All'interno dell'ambiente EMF sono stati creati 2 progetti connessi tra di loro. Il primo progetto contiene il metamodello OData in formato ECORE ed i modelli input e output di esempio. Il secondo progetto invece è un progetto di tipo "QVT Operational Project" e contiene il codice della trasformazione trattato nella sezione precedente e il riferimento al metamodello OData presente nel primo progetto. Questo riferimento è inserito nella lista "-¿ Metamodel Mappings" accessibile tramite le "QVT settings". Si osservi la figura 3 per avere chiara la struttura dei due progetti.

Figure 3: OData Metamodel Portion in EMF




Per avviare il processo di trasformazione servono due modelli I/O rispettivamente UML2 e OData. Il modello UML2 è creato a parte con un edito per modelli (ad esempio UML Editor già presente in




(a) Creazione Istanza Dinamica

(b) Istanza Dinamica

Figure 4

EMF) , mentre per il modello di output bisogna creare un'istanza dinamica a partire dal metamodello OData. Quest'ultima viene creata affinche il modello di output (che sarà in formato OData) possa fare riferimento al metamodello OData (importato come package). Si osservino 4a per la creazione dell'istanza dinamica e 4b per il modello di output vuoto ma che contiene il metamodello OData come package.

A seguire sono mostrati una serie di modelli di esempio UML2 a cui verrà applicata la trasformazione.

## 3.1 Product-Supplier Model

Figure 5: Product-Supplier.UML

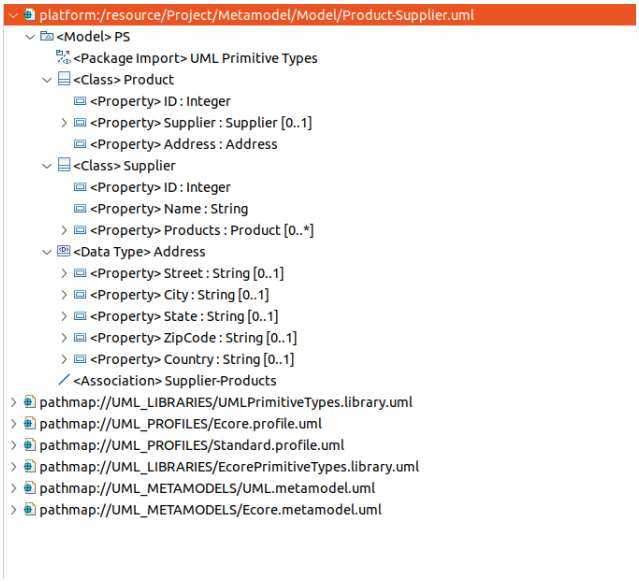
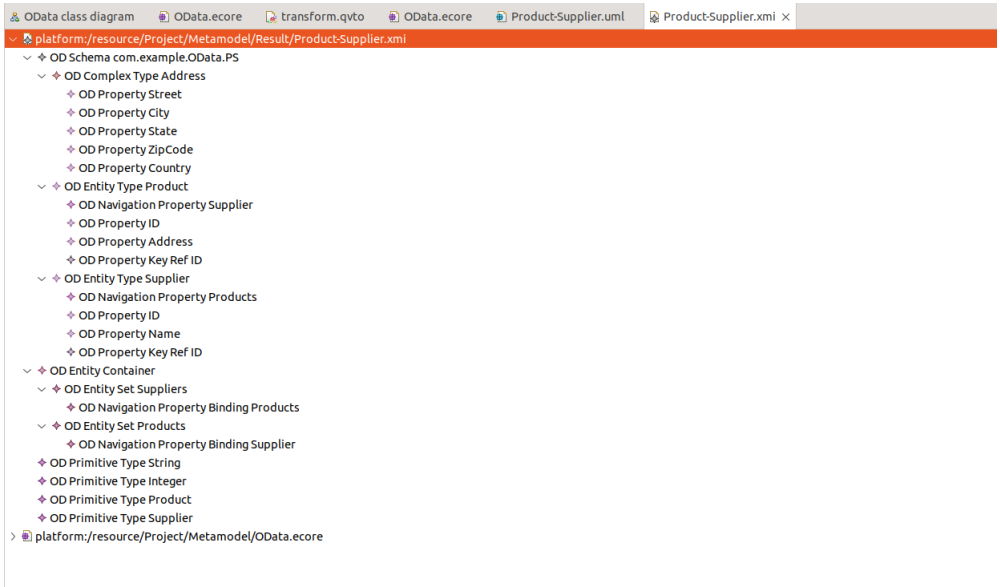



Figure 6: Product-Supplier.OData



## 3.2 Student-Report Model

Figure 7: Student-Report.UML

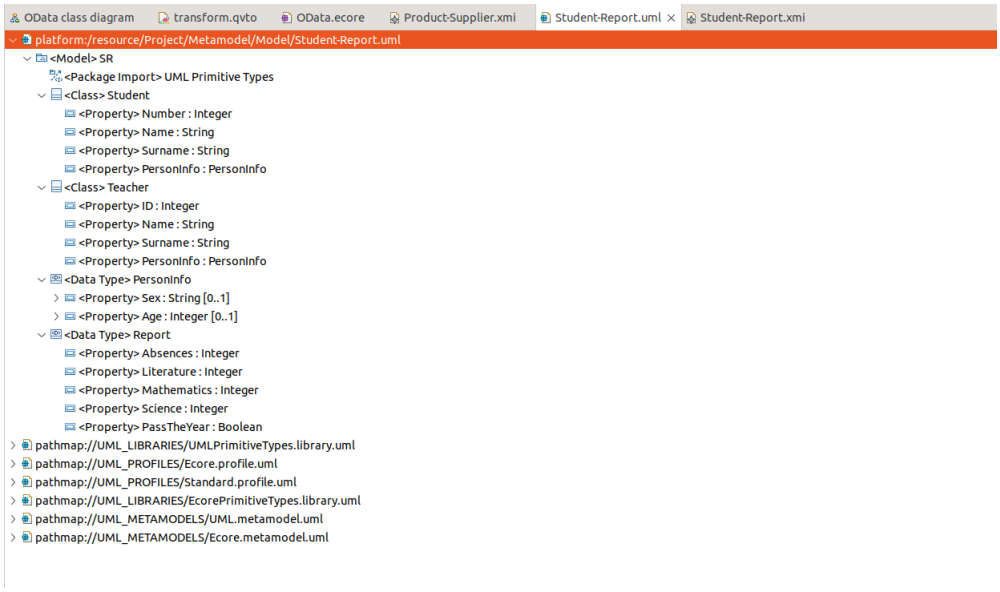
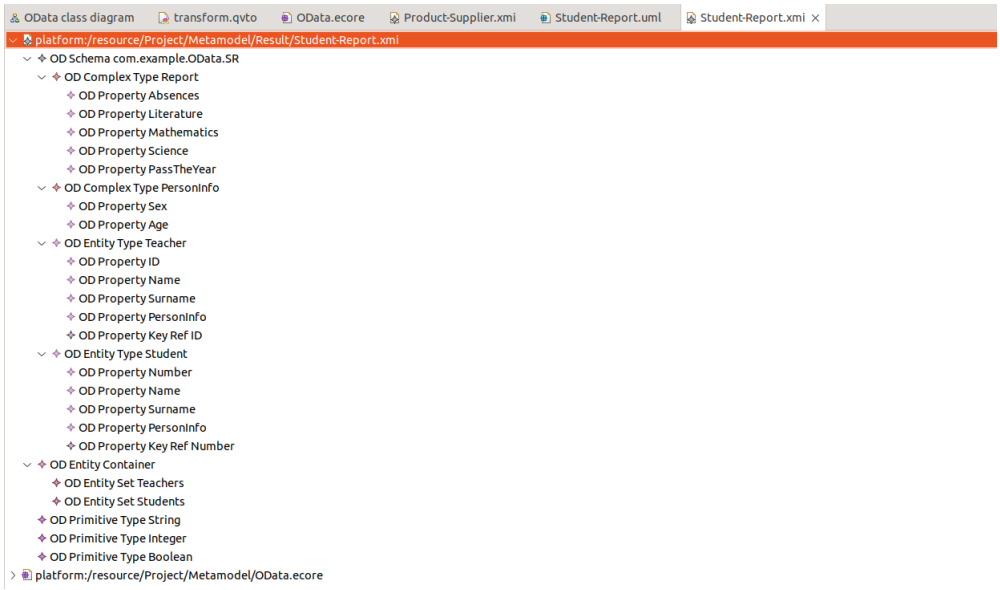



Figure 8: Student-Report.OData



## 3.3 Restaurant Model

Figure 9: Restaurant.UML

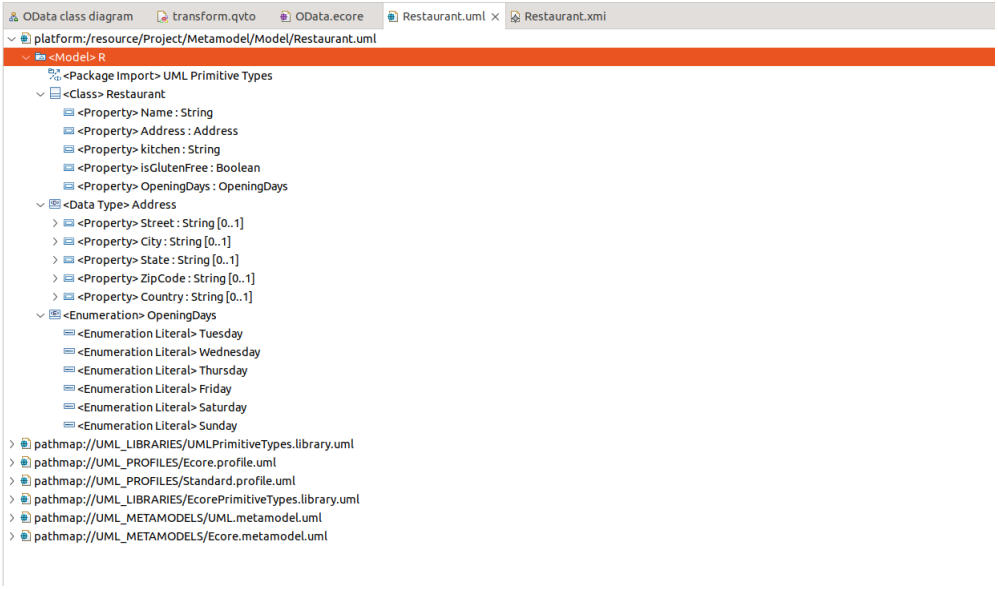
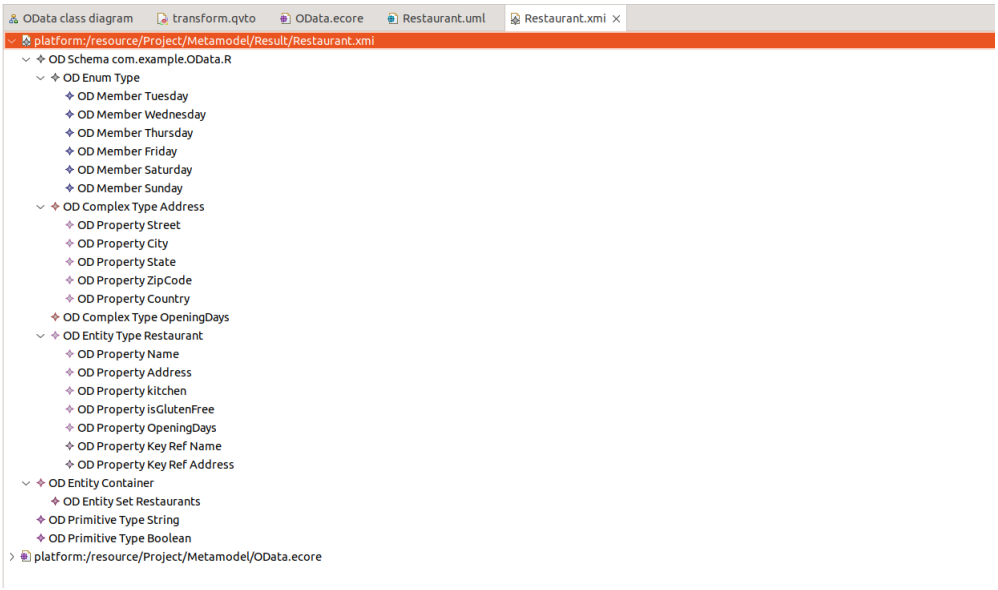



Figure 10: Restaurant.OData



## 3.4 Coach-Player Model

Figure 11: Coach-Player.UML

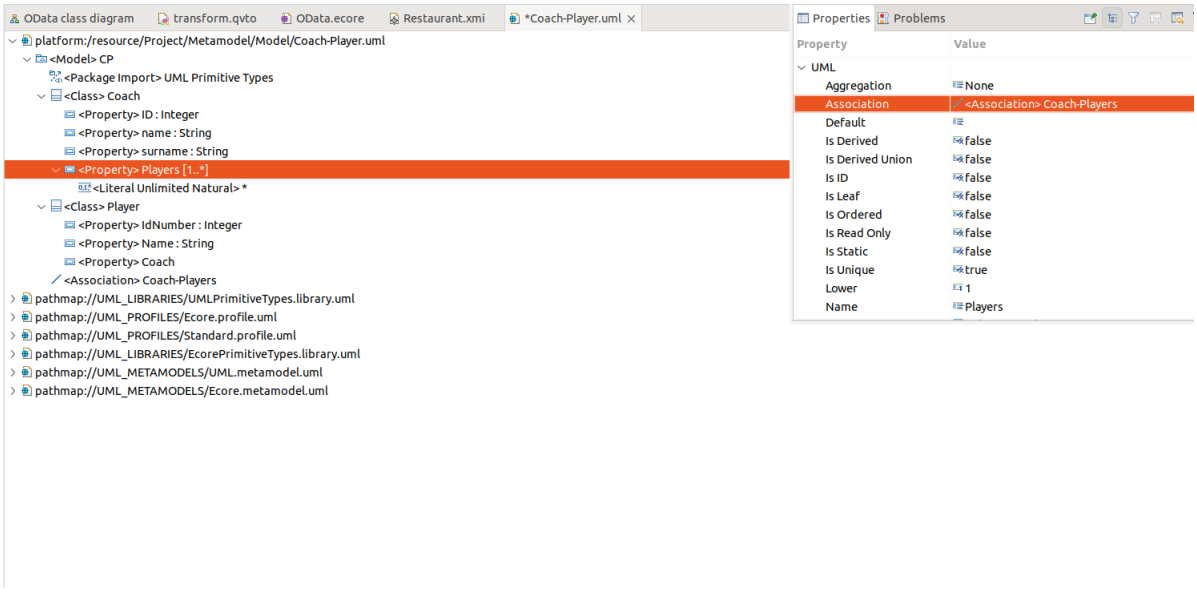
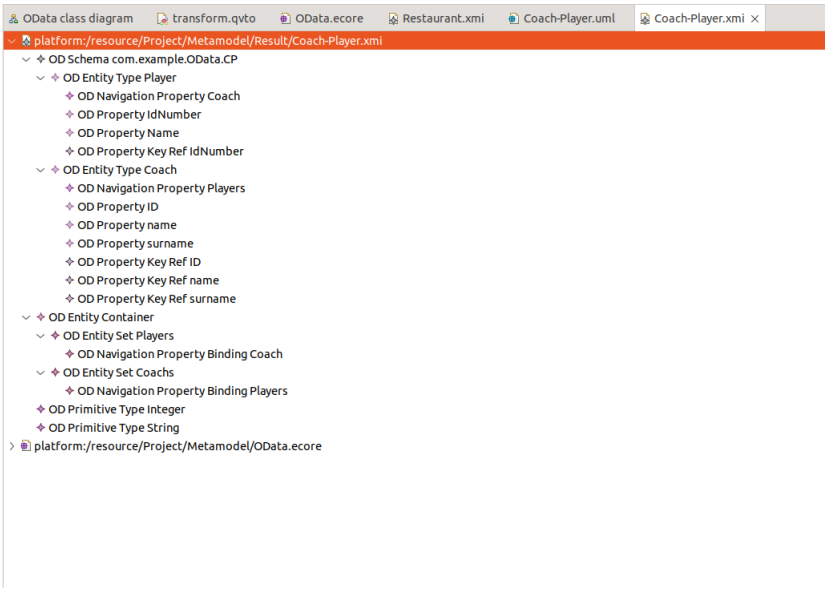



Figure 12: Coach-Player.OData



## 3.5 Car-Rental Model

Figure 13: Car-Rental.UML

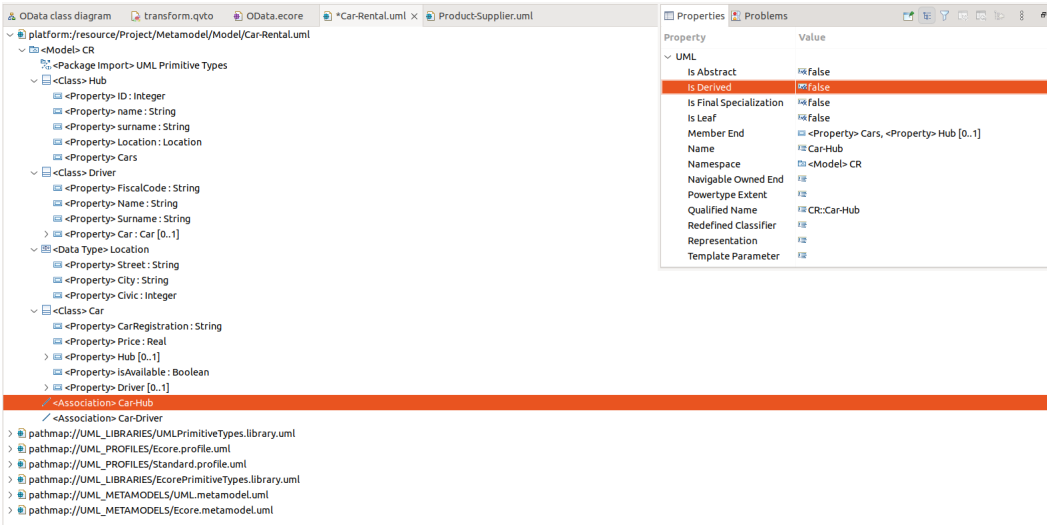
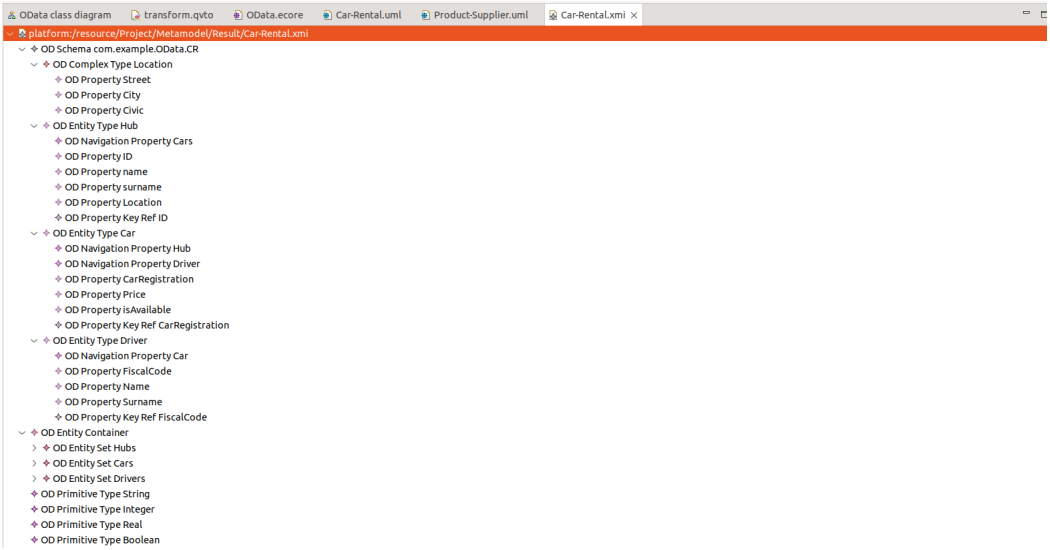



Figure 14: Car-Rental.OData



# References

[EdIC18] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. odel-driven development of odata services: An application to relational databases. pages 1–12, 2018.