Latvijas Universitāte
Fizikas un matemātikas fakultāte
Datorikas nodaļa

EMF un EMOF metamodelēšanas standarti, to integrācija valodā MOLA

Maģistra darbs

Autors
Pāvels Bubens

Vadītājs
Edgars Celms
Mg. sc. comp.

Rīga, 2006.

Anotācija

Darbā ir izpētīta atbilstība starp metamodelēšanas līdzekļiem *Eclipse EMF* un *OMG EMOF*, kā arī *EMOF* modifikāciju, kura tiek izmantota *MOLĀ*. Uz šīs izpētes pamata ir izstrādāti algoritmi, kā modeļu datus pārveidot atbilstoši metamodeļiem.

Darbā ir izstrādāts *Eclipse* ietvara spraudnis (*Eclipse framework plugin*) ar kura palīdzību *MOLAs* modeļa dati var tikt pārnesti uz *EMF* modeli. Spraudņa arhitektūra tiek aprakstīta izmantojot klašu diagrammas.

Abstract

Master’s paper describes differences between *Eclipse EMF* and *OMG EMOF* modelling standards. *MOLA*, another metamodelling standard that is very similar to *OMG EMOF*, also has been described. There has been developed the transformation algorithm that can transfer data from *MOLA* to *EMF*.

There have been created *Eclipse framework plug-in* that realize above transformation algorithm. *Eclipse framework plug-in* architecture is described using class diagrams.

Аннотация

В магистрской работе была иследована разница между средствами метамоделирования *Eclipse EMF* и *OMG EMOF*, а также *EMOF* модификация, которая используется в *MOLA*. На базе етих иследований был выработан алгоритм, который может переносить данные из одной модели в другую, согласно указанным метамоделям.

В результате работы был создан *Eclipse framework plugin,* с помощью которого данные *MOLA* модели переносит в *EMF* модель, согласно указанным метамоделям. Архитектура *Eclipse framework plugin* описана используя диаграмму класов.

Autoreferāts

Darba autors ir izstrādājis algoritmus ar kuru palīdzību modeļa dati no *MOLA* var tik pārnesti *EMF* modelī. Ar klašu diagrammu palīdzību tika parādīta atbilstība un attēlojums starp *MOLA* modelēšanas standarta koncepcijām un *EMF* modelēšanas koncepcijām.

Darba autors ir izstrādājis *Eclipse* ietvara spraudni (*Eclipse framework plugin*) kas realizē dotos transformācijas algoritmus.

Eclipse ietvara spraudnim ir izveidots intuitīva lietotāja saskarne, ar kuras palīdzību var ērti izvēlēties transformējamo (ieejas) *MOLA* modeli un nodefinēt *EMF* modeļa saglabāšanas parametrus.

Var teikt, ka dotais *Eclipse* ietvaru spraudnis paplašina *Eclipse* platformu ar jauno funkcionalitāti – *MOLA* modeļa transformācija uz *EMF* modeli. Tādā veidā var panākt vienkāršu *MOLA* valodas integrāciju ar citiem *EMF* balstītiem modelēšanas rīkiem, piemēram *RSA*.

Satura rādītājs

Ievads....7
1. Meta Modelēšanas valodas....8
1.1. Eclipse Modeling Framework (EMF)....8
1.1.1. Eclipse platformas apskats....8
1.1.2. Eclipse Modelēšanas Projekts (*Eclipse Modeling Project*)....9
1.1.2.1. Infrastruktūras projekti....9
1.1.2.2. Industrijas standarti....10
1.1.2.3. Tehnoloģijas un Pētījumu projekti....11
1.1.3. EMF Ecore....12
1.2. MOLA(MOdel transformation LAnguage)....17
1.3. OMG EMOF meta modelis....21
1.3.1. Essential MOF (EMOF) modelis....21
1.3.2. MOF Meta līmeni (*Metalevel*)....26
1.4. Meta modelēšanas standartu salīdzinājums....28
1.4.1. EPackage....29
1.4.2. EClass....30
1.4.3. EDataType....31
1.4.4. EAttribute....32
1.4.5. EReference....34
2. Transformāciju algoritma formāls apraksts....36
2.1. Izveidot klasifikatorus....38
2.2. Izveidot atribūtus....42
2.3. Izveidot asociācijas....47
2.4. Saglabāt XMI resursu....52
3. Praktiska realizācija....55
3.1. Lietotāju saskarne....56
3.2. Eclipse spraudņa konfigurācija....60
3.2.1. Overview skats....60
3.2.2. Dependencies skats....60
3.2.3. Extensions skats....61
3.2.4. Plugin.xml datne saturs....62
3.3. Eclipse Platformas arhitektūra....63
3.4. EMF Persistence API....65
4. Demonstrācijas piemērs....68
4.1. Ieejas EMF modeļu nodefinēšana ar RSA rīku....68
4.2. Modeļu transformācija uz MOLA modeli....69
4.3. Modeļu transformācija ar MOLA rīku....71
4.4. MOLA modeļa transformācija atpakaļ uz EMF modeli....75
4.5. Izejas modeļa klašu diagramma....78
Nobeigums....79
Literatūras saraksts....80

Ievads

Modeļu balstītā izstrādāšanā (*MDD - Model Driven Development*) modelis ir galvenais artefakts programmatūras izstrādes dzīves ciklā. *MDD* piedāvā modeļu transformāciju izpildīšanu un kodu ģenerēšanas automatizāciju.

Modelēšanas tehnoloģijas integrācija nav triviāls uzdevums, kaut gan eksistē standarta valodas un datu apmaiņas standarti, jo ir nepieciešams rīkoties ar dažādām valodām (standartizētām vai patentētām, vispārīgām vai domēnu specifiskām, un tā tālāk) un rīkiem.

Galvenais mērķis ir atļaut metodoloģijas ekspertiem izstrādāt modelēšanas rīkus un valodas, un tad viegli saintegrēt tos ar jau eksistējošiem rīkiem. Šajā sakarā, *Eclipse Modeling Framework* (*EMF*) ir diezgan daudzsološa tehnoloģija.

Pirmajā nodaļā ir aprakstīti *MOLA*, *OMG EMOF*, un *EMF* modelēšanas standarti. Tik salīdzināti augstāk minētie modelēšanas standarti.

Otrajā nodaļa ir dots uzdevuma algoritma formāls apraksts, ar uzsvaru uz datu pārveidošanu

Trešajā nodaļā ir aprakstīta praktiskā realizācija – aprakstīta lietotāja saskarne, komentāri par to kā ir programmēts un ir doti daži svarīgāko izmantoto tehnoloģiju apskati.

Ceturtajā daļa ir demonstrācijas piemērs. Demonstrācija piemēra galvenais mērķis ir pārliecināties, ka ar šo jauno rīku ir iespējama vienkārša sadarbība ar kādu citu *EMF* balstītu modelēšanas rīku – šai gadījumā ar *RSA* rīku.

1. Meta Modelēšanas valodas

1.1. *Eclipse Modeling Framework (EMF)*

1.1.1. *Eclipse* platformas apskats

Elipse tika attīstīta no tradicionāla integrēta izstrādāšanas rīka (*IDE*) uz varenu programmatūras ekosistēmu ar diezgan lielu atklātā koda līdzdalību. *Eclipse* pamatā ir modeļu balstīts meta datu vadības karkass *Elipse Modeling Framework* (*EMF*). Šis karkass atvieglo atšķirīgu rīku integrēšanu šajā ekosistēmā.

Eclipse priekštecis bija *Visual Age IDE*. 1990 gadā *IBM* nolēma atteikties no šī rīka tālākas attīstīšanas, jo, lai nodrošinātu jauno meta datu pieeju šajā vidē, bija nepieciešams rakstīt daudz jauna koda. Sākot no *Visual Age 3.0* (tika izlaista 1999) rīka iekšējie meta dati tika vadīti ar meta modeļiem. [2]

Eclipse platformas galvenās īpašības[8]:

- Platforma domāta dažādiem izstrādāšanas rīkiem un aplikāciju veidiem.
- Valodu neitrāla. Atbalsta iespēju darboties ar neierobežota satura tipiem. Piemēram, lai būtu iespēja izmantot kā izstrādāšanas rīku vairākām programmēšanas valodām: *HTML*, *Java*, *XML* un tā tālāk.
- Veicina konsekventu un sakarīgu rīku integrāciju gan lietotāja saskarnes līmenī, gan aplikācijas integrācijas un sadarbības līmenī. Lai būtu viegli pievienot jaunus rīkus jau eksistējošām konfigurācijām.
- Platforma domāta dažādām aplikācijām, ar grafisko saskarni un bez tās.
- Pievilkt rīku izstrādātāju sabiedrību. Popularizēt *Java* valodu kā galveno jauno rīku izstrādāšanas valodu.
- Platforma, kas varētu tikt darbināta uz vairākām operētāju sistēmām.

1.1.2. *Eclipse* Modelēšanas Projekts (*Eclipse Modeling Project*)

Eclipse Modelēšanas Projekta galvenais mērķis ir veicināt modeļu balstītas izstrādāšanas tehnoloģijas uz *Eclipse* platformas. Tas sastāv no vairākiem karkasiem, rīkiem, un standartu implementāciju[4].

1.1.2.1. Infrastruktūras projekti

Eclipse Modeling Framework (EMF)

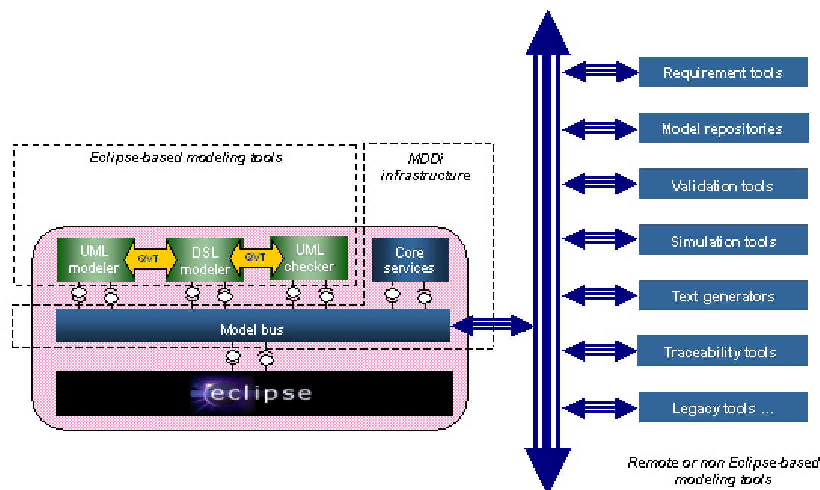
Modelēšanas karkass ar koda ģenerācijas iespēju. *EMF* galvenais mērķis - nodrošināt sadarbības iespējas pamatus dažādiem *EMF* balstītiem rīkiem un aplikācijām.

Graphical Modeling Framework (GMF)

Nodrošināt ģeneratīvo infrastruktūru, lai veidotu grafiskus redaktorus kas ir būvētas uz *EMF* un *GEF* bāzes.

Model Driven Development integration (MDDi)

Izveidot paplašināmus karkasu un paraugu rīkus rīku integrācijai *Eclipse* platformā. Šis karkass, formalizējot operācijas ar modeli, atvieglos rīku integrāciju, piemēram, modeļu transformācijas, modeļu validēšanu. Kā arī, tas piedāvā līdzekļus lai varētu apmainīties ar modeļu semantikām.



Attēls 1.1. *Model Driven Development* integration projekta arhitektūra[5].

1.1.2.2. Industrijas standarti

Unified Modeling Language (UML2)

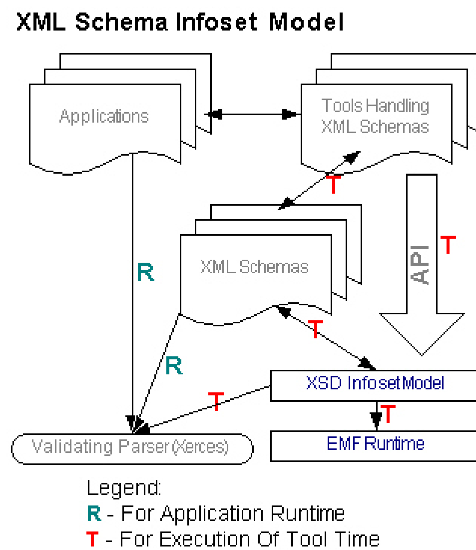
EMF balstīta *UML2* meta modeļu implementācija.

Object Constraint Language (OCL)

Definē *API OCL* izteiksmes sintaksei, lai veidotu vaicājumus un ierobežojumus.

XML Schema Infoset Model (XSD)

Rekomendācijas bibliotēka, kas piedāvā *API*, lai darboties ar *W3C XML Schema*.



Attēls 1.2. *XML* shēmas infoset modelis.

EMF Ontology Definition Metamodel (EODM)

RDF(S)/OWL Ontology Definition Metamodel (ODM) meta modeļu implementācija izmantojot *EMF* ar dažām papildu iespējām: *parsing*, secinājumi, modeļu transformācijas un rediģēšanas funkcijām.

1.1.2.3. Tehnoloģijas un Pētījumu projekti

Java Emitter Templates (JET)

Piedāvā kodu ģenerēšanas karkasu un līdzekļus *EMF* videi. *JSP* līdzīgas veidnes datnes (*template files*) varētu būt rediģēti un transformēti uz jebkādu koda (*source*) artefaktu. Piemēram, *Java*, *HTML*, vai *XML* datnēm.

Model Query (MQ)

Modeļa elementu meklēšana un iegūšana.

Model Transaction (MT)

Piedāvā modeļu vadības karkasu uz *EMF* bāzes, lai pārvaldīt *EMF* resursus.

Validation Framework (VF)

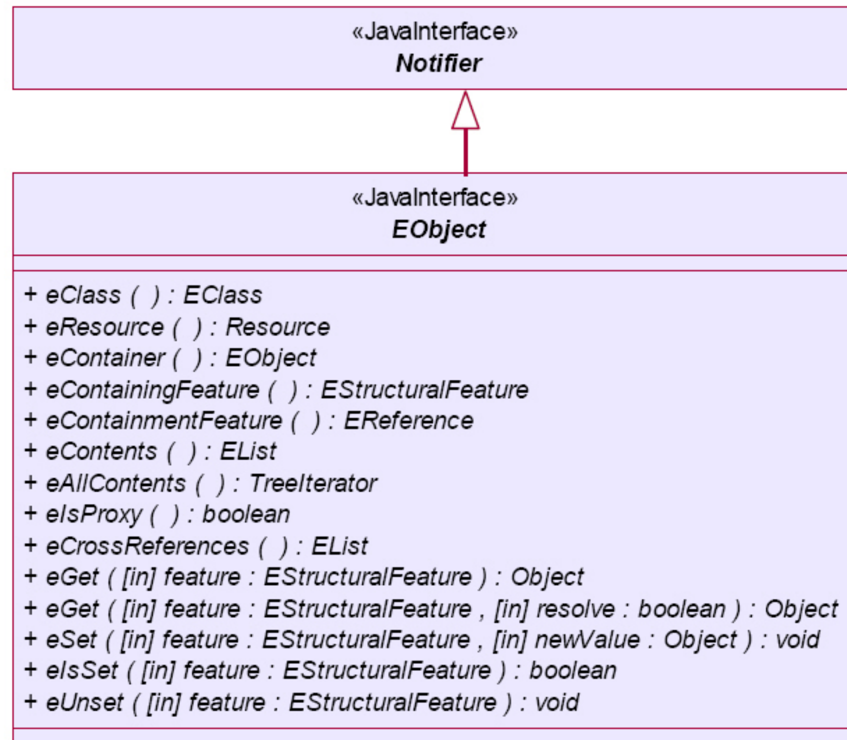
Piedāvā modeļu ierobežojumu definīcijas, navigācijai un novērtējumus modeļu validācijai.

Generative Modeling Tools (GMT)

Pētījumu orientēts projekts kas koncentrējas prototipu izstrādāšana *Model Driven Engineering (MDE)* sfērā.

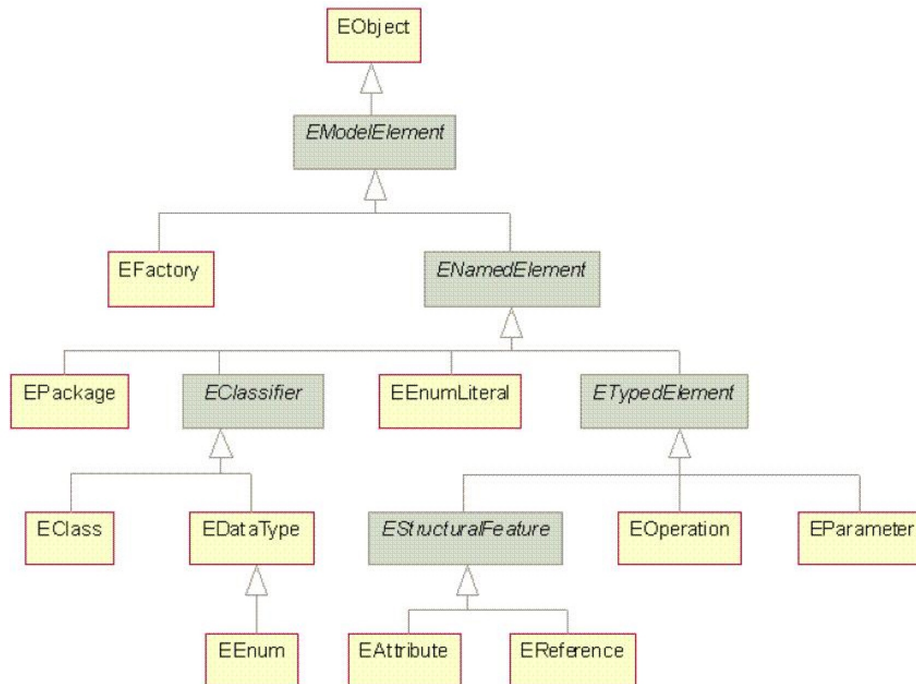
1.1.3. *EMF Ecore*

Ecore ir *EMF* meta modelis. Meta modelis tiek izmantots, lai veidotu modeļus *EMF* vidē. *Ecore* definē pati sevi.



Attēls 1.4. *EObject* klases definīcija [10].

EObject ir pamatobjekts *ECore*, līdzīgi ka *Object Java* valodā. Visi *EMF* objekti implementē *EObject* saskarni. Ar *eClass()* metodi vienmēr var uzzināt dotās klases meta klasi. Ar *eResource()* metodi var atrast to resursu, kur atrodas dotais objekts. Resurss ir glābšanas mehānisms dotam objektam.

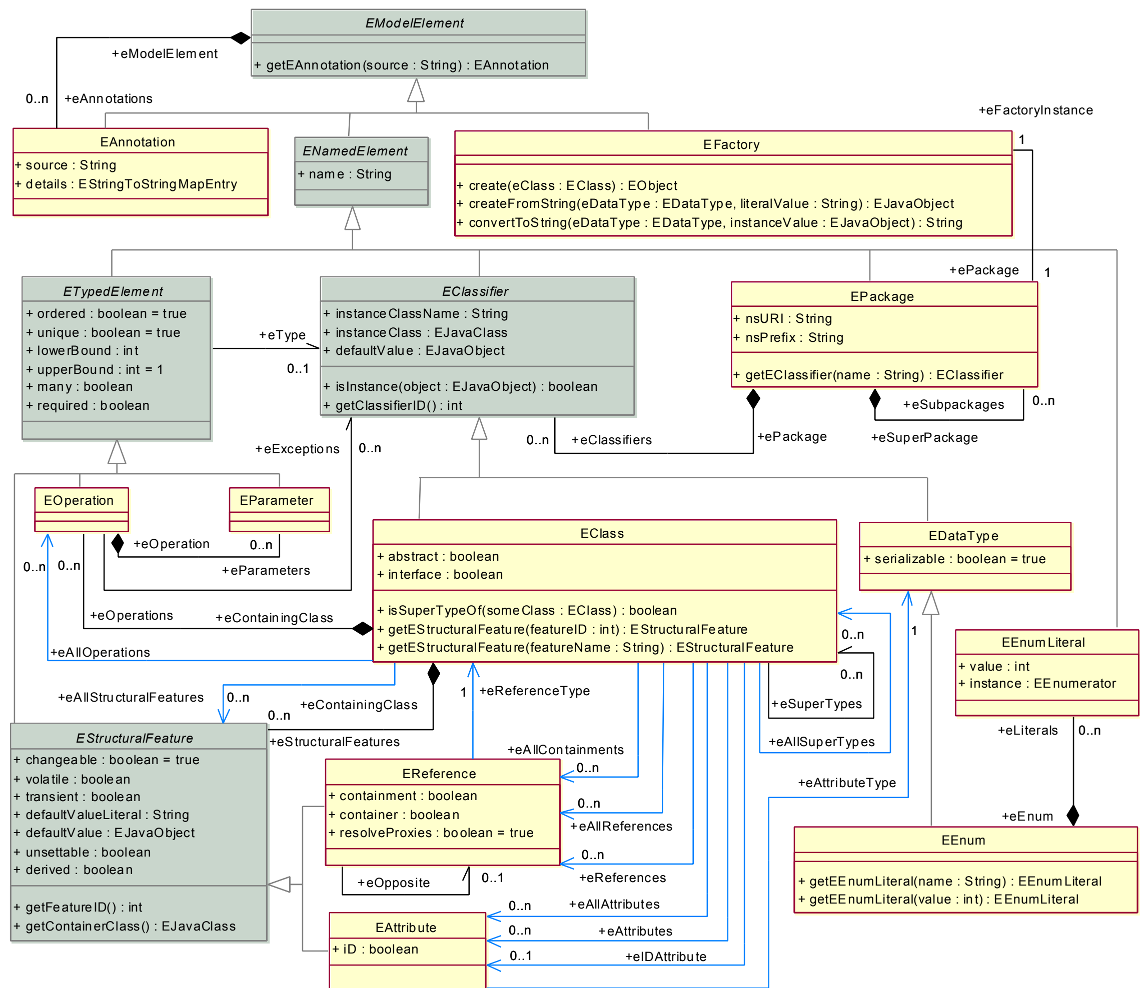


Attēls 1.5. *ECore* hierarhija [10]

EMF ir diezgan vienkārša fiksēta veidošanas struktūra:

- *EPackage* var saturēt apakšpaketes un klasifikatorus (*EClass* un *EDataType*).
- *EClasses* var saturēt *EAttributes*, *EReferences* un *EOperations*.
- *EOperation* var saturēt *EParameters* un tā tālāk.

Vienīgais izņēmums ir *EAnnotation*.



Attēls 1.6. *Ecore* implementācija [10]

ECore galvenās koncepcijas[11]:

`EPackage`. `EPackage` tiek izmantots, lai grupētu saistītās klases un datu tipus, kas atvieglotu modeļu uztveri un pārvaldību.

`EClassifier`. `EClassifier` ir tipu apraksts *ECore*. Jebkurš `EClassifier` ir `EClass` vai `EDataType`.

`EClass`. `EClass` ir viens no fundamentālākiem objektiem `ECore`. `EClass` var būt abstrakts vai konkrēts. Ir nodrošināta mantošanas koncepcija.

EDataType. EDataType ir ne *ECore* tipu apraksts. Tas var būt vai primitīvs tips, vai *Java Class*, kurš ir nodefinēts ārpus *ECore* modeļa, vai arī EEnum.

Tabula 1.1. *Java* valodas tipi iekš *ECore* [12]

Ecore Data Type	*Java Type*	*Serializable*
EBoolean *EByte*	*boolean* *byte*	jā jā
EChar	*char*	jā
EDouble *EFloat*	*double* *float*	jā jā
EInt	*int*	jā
ELong	*long*	jā
EShort	*short*	jā
EBooleanObject	*java.lang.Boolean*	jā
EByteType	*java.lang.Byte*	jā
ECharacterObject	*java.lang.Character*	jā
EDoubleObject	*java.lang.DoubleObject*	jā
EFloatObject	*java.lang.Float*	jā
EIntegerObject	*java.lang.Integer*	jā
ELongObject	*java.lang.Long*	jā
EShortObject	*java.lang.Short*	jā
EString	*java.lang.String*	jā
EJavaObject	*java.lang.Object*	nē
EJavaClass	*java.lang.Class*	jā

EEnum. EEnum tas ir tips, saraksts ar EEnumLiterals.

EStructuralFeature. EStructuralFeature apraksta elementu kas pieder dotajam klasei. Tas varētu būt vai EAttribute, vai EReference. Jebkuram EAtributes un EReference ir tips. Šim elementam arī var būt nodefinētas vairākas īpašības, piemēram, lowerBound, upperBound.

`EOperation.` `EOperation` ir elements kas apraksta kādu no metodēm no šīs klases.

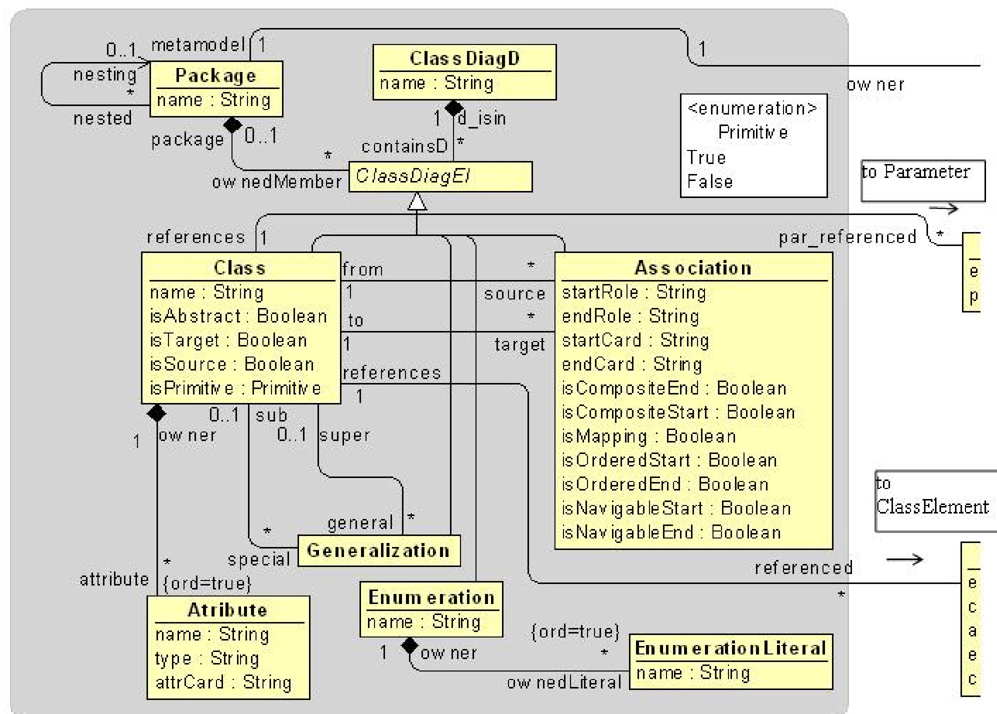
`EAnnotation`. Anotācijas izveido mehānismu kā pievienot papildu informāciju jebkuram objektam modelī. Piemēram, saistībā ar to, ka *MOF* ir daudz bagātāks metamodelis nekā *EMF* un lai nebūtu informācijas zudumi. `EAnnotation` ir viens atribūts “*source*”, kas atspoguļo `EAnotation` tipu. Piemēram, “*keyword*”.

`EStringToStringMapEntry.` `EAnnotation` izmanto `EStringToStringMapEntry` lai glabātu papildu informāciju par kādu *EMF* klasifikatoru. `EAnnotation` ir nodefinēts saraksts ar `EStringToStringMapEntry`. `EStringToStringMapEntry` ir divi atribūti atslēga un vērtība. Attiecīgi pēc tā atslēga var atrast doto `EStringToStringMapEntry` dotajā `EAnnotation` saraksta.

1.2. MOLA(MOdel transformation LAnguage)

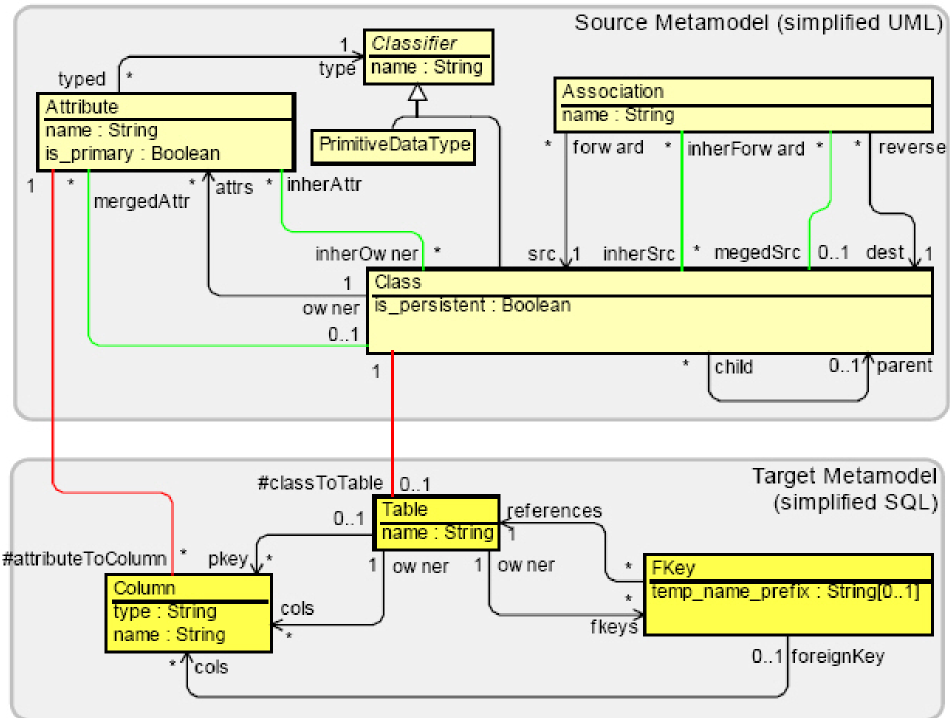
MOLA modeļu transformācijas valoda tika izstrādāta *Latvijas Universitātē*. Galvenais *MOLA* mērķis ir izveidot viegli lasāmu grafisku transformācijas valodu [14].

Līdzīgi, kā vairākām citām modeļu transformāciju valodām, *MOLA* tiek izmantots ieejas (*source*) un izejas (*target*) meta modelis. Valodā *MOLA* gan ieejas, gan izejas meta modelis tiek attēloti vienā klašu diagrammā. Strukturēšanai var tikt izmantotas paketes. *MOLA* valodā izmantotais meta modelis ir stipri līdzīgs *OMG EMOF*, tikai ar dažiem ierobežojumiem[14].



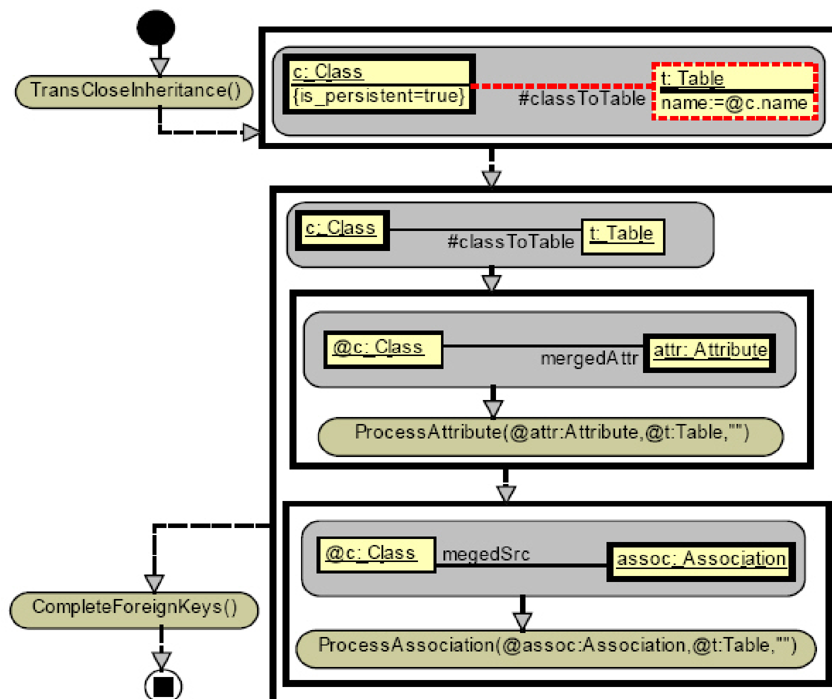
Attēls 1.7. *MOLA* meta modelis[15].

Speciālas attēlojuma (*mapping*) asociācijas tiek izmantotas, lai sasaistītu atbilstošas klases ieejas un izejas meta modeļos. Šajā ziņā, tas ir līdzīgi kā relācijas koncepcija citās transformāciju valodās – lai nostrukturizētu transformāciju un nodokumentētu transformācijas trasējamību. Ja ir nepieciešams glabāt starpposma datus, tad var tikt pievienoti īslaicīgas (*temporal*) klases un asociācijas.



Attēls 1.8. Kombinētie ieejas un izejas meta modeli iekš *MOLA* [14].

Pati par sevi transformācija tiek nodefinēta ar vienu vai vairākām *MOLA* diagrammām.

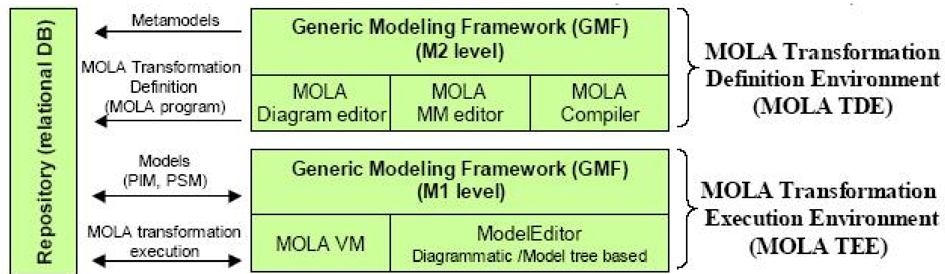


Attēls 1.9. *MOLA* transformācija programmas piemērs [14].

MOLA rīki [14]

MOLA rīkiem ir divas daļas:

- Transformāciju definēšanas vide (*Transformation Definition Environment*)
- Transformācijas izpildes vide (*Transformation Execution Environment*)



Attēls 1.10. *MOLA* rīku shēma [14].

Abas vides izmanto kopējo izpildes laika repositoriju un patreiz tā ir relāciju datubāze.

Lai vienkāršotu darbību ar *MOLA*, tad praksē tiek izmantots *MOLA* kodējuma meta modelis. Šajā ziņā, tas ir klašu diagramma plus datubāzes definīcija.

Attēls 1.11. *MOLA* kodējumu meta modelis.

1.3. OMG EMOF meta modelis

MOF (Meta Object Facility) ir adoptēts *OMG (Object Management Group)* standarts piedāvā meta modelēšanas karkasu modelēšanai, dažādus meta datu servisus, lai veicinātu modeļu izstrādāšanu un sadarbspēju. *MOF* tiek izmantots vairākās sfērās: modelēšanas un izstrādāšanas rīki, datu glabātuvju sistēmas, meta datu *repository* un tā tālāk.[16].

MOF ir pamatmehānisms modeļu vadītā arhitektūrā *(MDA)*. Galvenā *MOF* ideja – ka ir nepieciešama vairāk nekā viena modelēšana valoda. Lai būtu iespēja veidot modelēšanas valodu atšķirīgiem sistēmas aspektiem un abstrakcijas līmeņiem. Piemēram, dažādas modelēšanas valodas prasa dažādas modelēšanas konstrukcijas:

- relāciju datu modelēšana prasa tādas koncepcijas kā *table*, *column*, *key* un tā tālāk.
- *workflow* modelēšanā ir svarīgi, lai būtu tādas koncepcijas ka *task, performer, split, join* un tā tālāk.
- klasiska objektu orientētas klašu modelēšana ir nepieciešami *class, attribute* un tā tālāk.

Pēdējā standartu versija ir *MOF2*. Viena no svarīgākajām *MOF2* izmaiņām ir tāda, ka tagad *MOF2* un *UML2* koplieto *UML2* infrastruktūras bibliotēku.

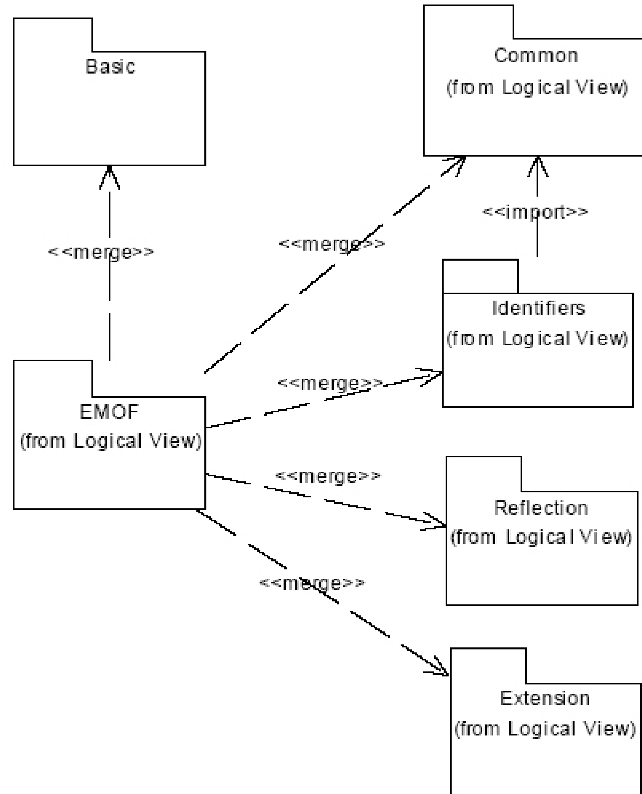
MOF2 modelis sastāv no divām paketēm:

- *Essential MOF (EMOF)*
- *Complete MOF (CMOF)*

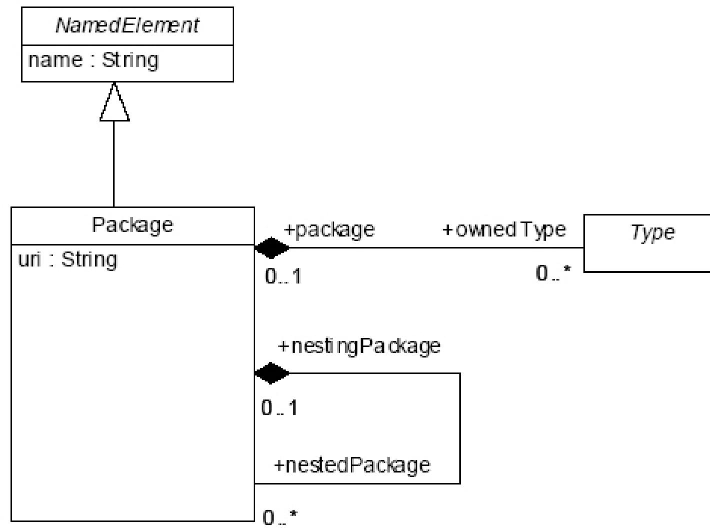
1.3.1. *Essential MOF (EMOF)* modelis

EMOF ir apakškopa no *MOF*, kas tuvu atbilst tām iespējām kuras ir atrodamas *XML* un objektu orientētās valodās. *EMOF* mērķis ir vienkāršu meta modeļu definēšana izmantojot vienkāršas koncepcijas. *EMOF* modelis apvieno (*merge*) *Basic* paketes no *UML2* un iekļauj papildu valodas līdzekļus ar kuriem ir nodefinētas specifikācijas. *EMOF* modelis apvieno *Reflection, Identifiers*, un *Extension* paketes lai nodrošinātu atklāšanas, manipulēšanas, identifikācijas un meta datu paplašināšanas

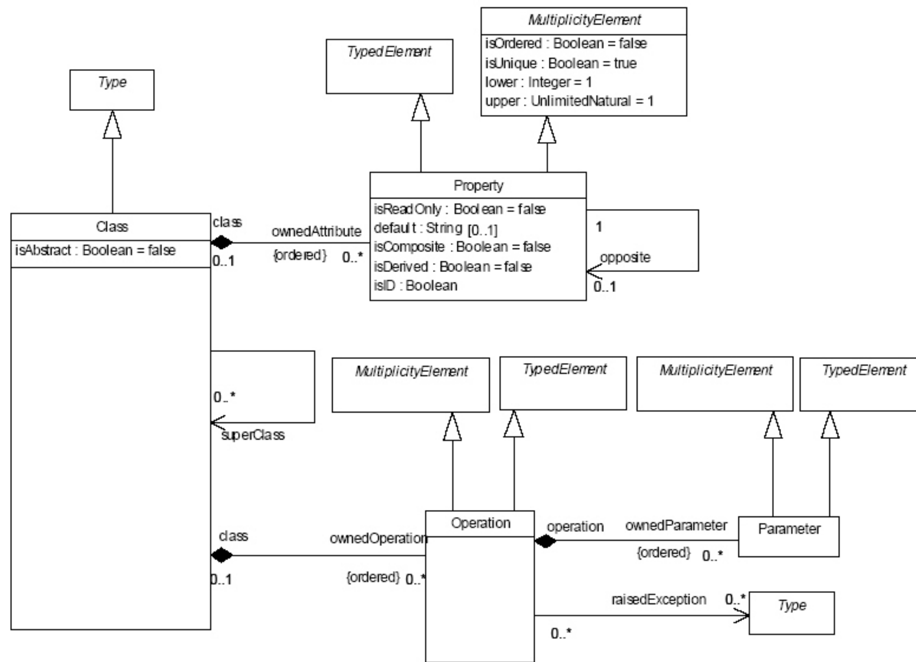
servisus. *EMOF* ir nodefinēts kā *CMOF* modelis, kaut gan pilns atbalsts prasa, lai *EMOF* definētu pati sevi [16].



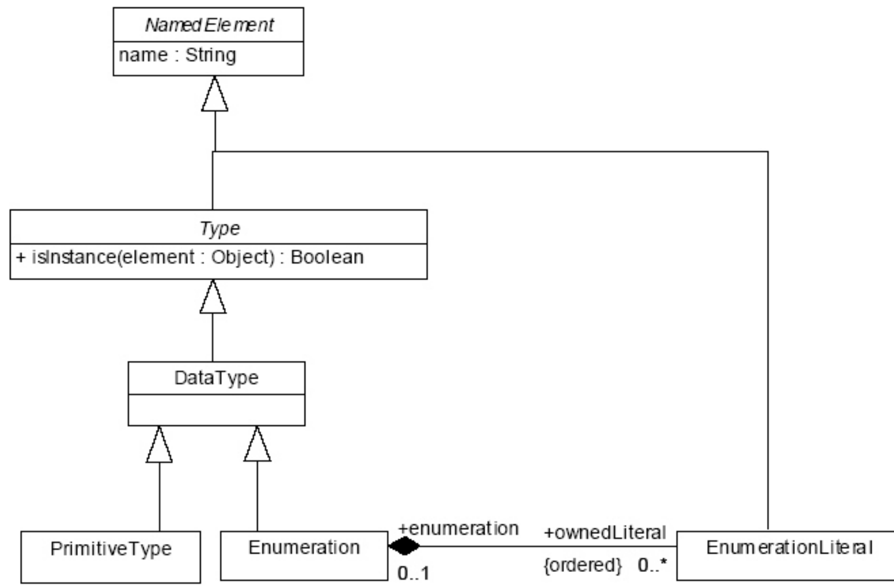
Attēls 1.12. *EMOF* modeļu apskats [16].



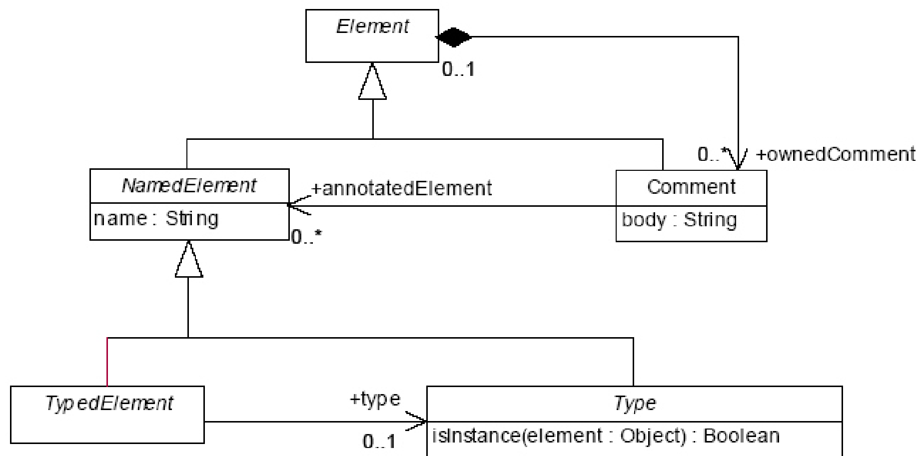
Attēls 1.13. *EMOF Packages* [16].



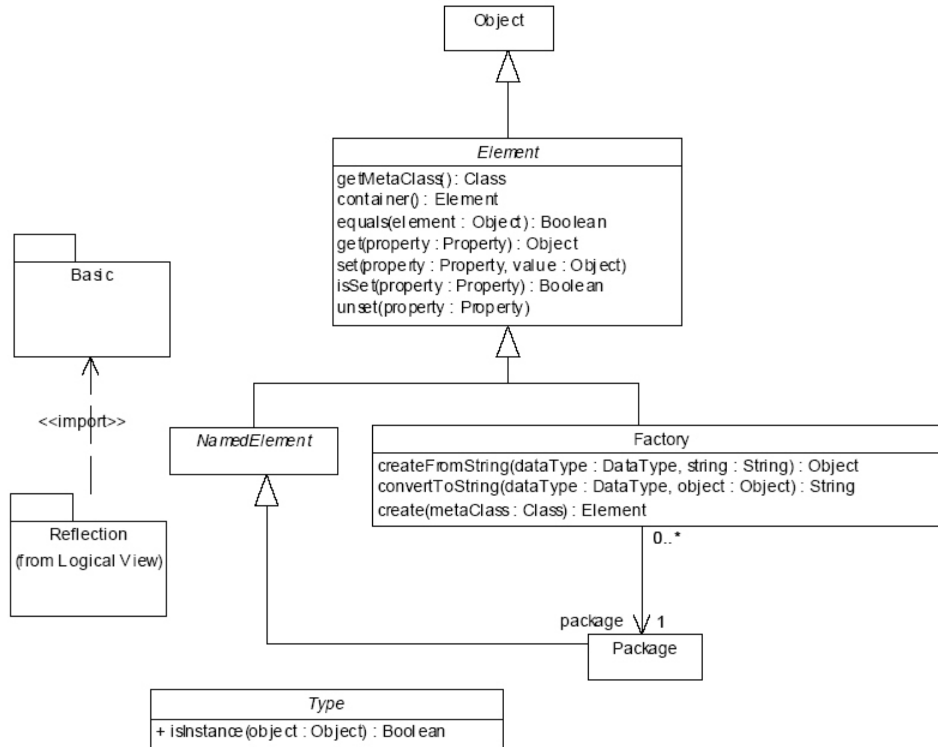
Attēls 1.14. *EMOF Classes* [16].



Attēls 1.15. *EMOF Data Types* [16].



Attēls 1.16. *EMOF Types* [16].



Attēls 1.17. *Reflection package* [16].

1.3.2. *MOF* Meta līmeni (*Metalevel*)

M3 līmenis.

M3 līmenis ir *MOF*, kuras elementi ir *MOF* konstrukcijas, kuras ir piedāvātas meta modeļu definēšanai. Dotās koncepcijas iekļauj sevī *Class, Attribute, Association* un tā tālāk. Konceptuāli ir tikai viens *MOF*. Bieži to sauc par meta-metamodeli. *MOF* apraksta pati sevi.

M2 līmenis.

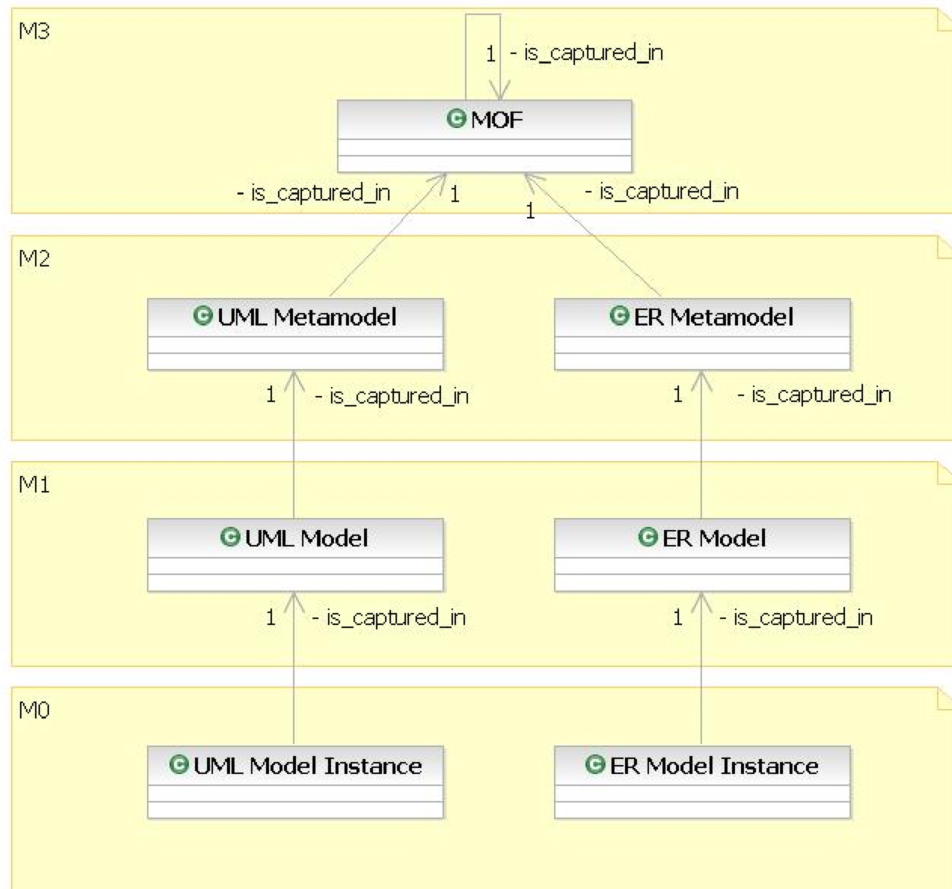
Meta modelis, nodefinēts ar *MOF* konstrukcijām. Piemēram, *UML*, *CWM*. *M2* konstrukcijas ir *M3* konstrukcijas instances.

M1 līmenis.

Modeli, nodefinētas ar *M2* meta modeļu konstrukcijas instancēm. Piemēram, klase “*Customer*”.

M0 līmenis.

Objekti un dati, *M1* konstrukciju instances. Piemēram, konkrēts klients *Joe*.



Attēls 1.18. *MOF* meta līmeņu koncepcija.

1.4. Meta modelēšanas standartu salīdzinājums

Šajā nodaļā tiek salīdzināti *EMOF*, *MOLA*, un *EMF* meta modelēšanas standarti. Tas ir domāts kā pamats modeļu transformāciju algoritmiem. Tiek atrastas galvenās ekvivalentās koncepcijas katrā meta modelēšanas valodā.

Meta modelēšanas konstrukcijas salīdzinājums notiek *M3* līmeni. Tiek salīdzinātas meta modelēšanas konstrukcijas *EMF*, *OMG EMOF*, *MOLA KMM* modelēšanas valodās. Ka jau bija teikts, *MOLA KMM* ir *MOLA* kodējuma meta modelis.

Meta modelēšanas konstrukciju salīdzinājums.

Tabula 1.2. Svarīgāko koncepciju salīdzināšanas tabula.

EMF	*OMG EMOF*	*MOLA KMM*
`EPackage`	`Package`	`Package`
`EClass`	`Class`	`MetaClass`
`EEnum`	`Enumeration`	`MetaEnumeration`
`EDataType`	`DataType`	`Type`
`EAttribute`	`Property`	`MetaAttribute`
`EReference`	`Property`	`MetaAssociation`
`EENumLiteral`	`EnumerationLiteral`	`MetaEnumeration`
`EOperation`	`Operation`	neizmanto
`EParameter`	`Parameter`	neizmanto

1.4.1. EPackage

Tabula 1.3. `EPackage` atribūtu un asociāciju specifikācija.

Atribūts	Apraksts
`name`	paketes nosaukums
`nsURI`	paketes *Namespace URI*
`nsPrefix`	*Namespace* prefikss
`eClassifiers`	Klasi, *enumeration* un datu tipi šajā paketē
`eSubpackages`	Iekļautas paketes

Tabula 1.4. `EPackage` atribūtu un asociāciju atbilstības salīdzinājums.

EMF	*OMG EMOF*	*MOLA KMM*
`name`	`name`	`name`
`eClassifiers`	`ownedType`	`classes`
`eSubpackages`	`nestedPackage`	`packages`

1.4.2. EClass

Tabula 1.5. *EClass* atribūtu un asociāciju specifikācija

Atribūts	Apraksts
name	Nosaukums
instanceClass	Instances tips. *null* apzīmē dinamisko klasi.
defaultValue	Noklusēta vērtība
abstract	`abstract` atslēgvārds
interface	`interface` atslēgvārds
eAttributes	Šis klases atribūti
eReferences	Saistītas asociācijas
eOperations	Šis klases operācijas
eSupertypes	Šis klases supertips

Tabula 1.6. *EClass* atribūtu un asociāciju atbilstības salīdzinājums.

EMF	*OMG EMOF*	*MOLA KMM*
`name`	`name`	`name`
`abstract`	`isAbstract`	`is_abstract`
`interface`	neizmanto	neizmanto
`eAttributes`	`ownedAttribute`	`ownedAttribute`
`eReferences`	`ownedAttribute`	`outAssoc`
`eOperations`	`ownedOperation`	neizmanto
`eSupertypes`	`superClass`	`parent`

1.4.3. EDataType

Tabula 1.7. `EDataType` atribūtu un asociāciju specifikācija.

Atribūts	Apraksts
`name`	Nosaukums
`instanceClass`	Instances tips. *null* apzīmē dinamisko klasi.
`defaultValue`	Noklusēta vērtība
`serializable`	Jā `serializable` ir `false` tad visi atribūti ir `transient`

Tabula 1.8. `EDataType` atribūtu un asociāciju atbilstības salīdzinājums.

EMF	*OMG EMOF*	*MOLA KMM*
`name`	`name`	`name`
`instanceClass`	`type`	`type`

1.4.4. EAttribute

Tabula 1.9. `EAttribute` atribūtu un asociāciju specifikācija.

Atribūts	Apraksts
`name`	Nosaukums
`eType`	Atribūta tips, jābūt `EDataType`
`changeable`	Apzīmē vai atribūts varētu būt modificēts. Gadījumā, kad `changeable` ir `false`, tad `set()` metode nav noģenerēta dotam atribūtam.
`volatile`	Apzīmē vai atribūts var nokešot (*to be cached*). Jā, *volatile* ir `true`, tad noģenerēta klase nav tādu atribūtu, un atbilstošas `get()` un `set()` metodes ir tukšas.
`transient`	Apzīmē vai ir nepieciešam doto atribūtu saglabāt. Jā, `transient` ir `true`, *XMI* serializētais ignorēs doto atribūtu.
`unique`	Apzīmē vai varētu but vienādas vērtības iekš dota atribūta.
`defaultValue`	Noklusēta vērtība.
`lowerBound`	Iespaido uz `required` atribūtu. Jā, `upperbound` ir 0, tad `required` atribūts ir `false`. Savādāk ir `true`.
`upperBound`	Iespaido uz `many` atribūtu. Jā, `upperBound` ir 1, tad `many` atribūts ir `false`. Savādāk ir `true`.
`many`	Apzīmē vai tas ir vairāk vērtību atribūts vai ne.
`required`	Apzīmē vai atribūts ir obligāts vai ne.
`unsettable`	Apzīmē vai atribūts varētu but `unset`.

Tabula 1.10. `EAttribute` atribūtu un asociāciju atbilstības salīdzinājums.

EMF	*OMG EMOF*	*MOLA KMM*
`name`	`name`	`name`
`eType` `changeable`	`type` `isReadOnly`	`type` neizmanto
`volatile`	`isDerived`	neizmanto
`transient` `unique`	`isDerived` `isUnique`	neizmanto neizmanto
`defaultValue`	`default`	neizmanto
`lowerBound`	`lower`	`cardin.lower`
`upperBound`	`upper`	`cardin.upper`
`many`	`upper > 1`	`cardin.upper > 1`
`required`	`lower >= 1`	`cardin.lower >= 1`
`unsettable`	`lower == 0`	`cardin.lower == 0`

1.4.5. EReference

Tabula 1.11. `EReference` atribūtu un asociāciju specifikācija.

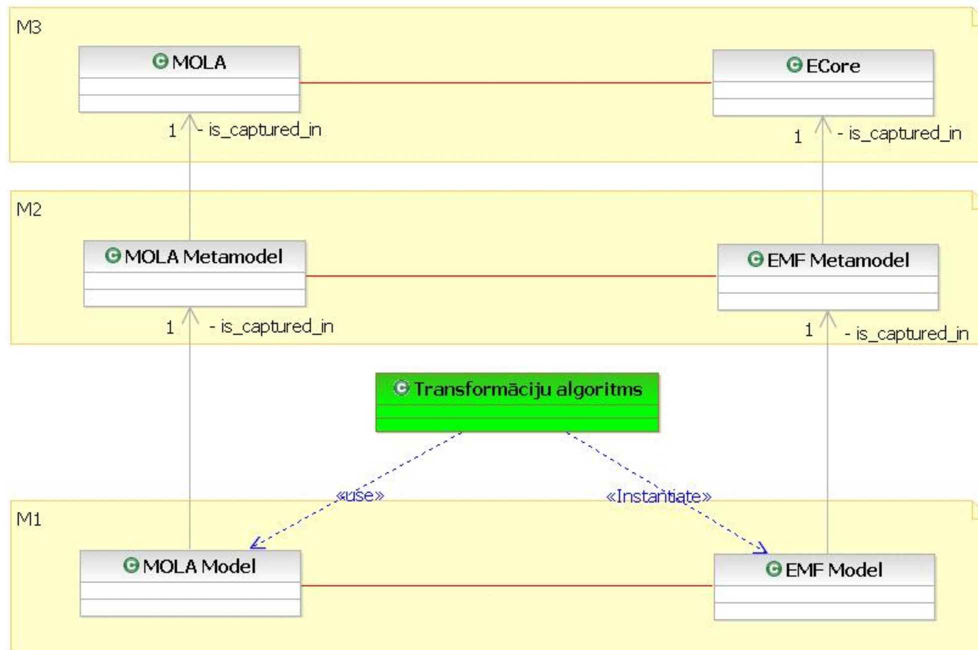
Atribūts	Apraksts
`name`	Nosaukums
`eType`	Referenšu tips.
`changeable`	Apzīmē vai *reference* varētu but modificēta. Gadījumā, kad `changeable` ir `false`, tad `set()` metode nav noģenerēta dotai referencei.
`volatile`	Apzīmē vai atribūts var nokešot (*to be cached*). Jā, `volatile` ir `true`, tad noģenerētai klasei nav tādu atribūtu, un atbilstošas `get()` un `set()` metodes ir tukšas.
`transient`	Apzīmē vai ir nepieciešam doto atribūtu saglabāt. Jā, `transient` ir `true`, *XMI* serializētais ignorēs doto reference.
`unique`	Apzīmē vai varētu but vienādas vērtības iekš dota reference.
`defaultValue`	Noklusēta vērtība.
`lowerBound`	Iespaido uz `required` atribūtu. Jā, `upperbound` ir 0, tad `required` atribūts ir `false`. Savādāk ir `true`.
`upperBound`	Iespaido uz `many` atribūtu. Jā, `upperBound` ir 1, tad `many` atribūts ir `false`. Savādāk ir `true`.
`many`	Apzīmē vai dota reference ir vairākas vērtību vai ne.
`required`	Apzīmē vai reference ir obligāta vai ne.
`containment`	Apzīmē vai reference ir konteinera tipa (*container*) semantika. Piemēram, ja objekts tika pievienots jaunam konteineram, tad objekts automātiski tika aizvākts no veca konteineri.
`container`	Apzīmē vai reference ir konteineris. Tas ir pretstats `containment`.
`resolveProxies`	Apzīmē vai proksi var automātiski risināties (*resolve*)
`eOpposite`	Apzīmē otro galu iekš dotai relācijai.

Tabula 1.12. `EReference` atribūtu un asociāciju atbilstību salīdzinājums.

EMF	*OMG EMOF*	*MOLA KMM*
`name`	`name`	`target_role`
`eType` `changeable`	`type` `isReadOnly`	`type` neizmanto
`volatile`	`isDerived`	neizmanto
`transient` `unique`	`isDerived` `isUnique`	neizmanto neizmanto
`defaultValue`	`default`	neizmanto
`lowerBound`	`lower`	`cardin.lower`
`upperBound`	`upper`	`cardin.upper`
`many`	`upper > 1`	`cardin.upper > 1`
`required`	`lower >= 1`	`cardin.lower >= 1`
`containment`	`lower == 0`	`cardin.lower == 0`
`container` `resolveProxies`	`container` neizmanto	neizmanto neizmanto
`eOpposite`	`opposite`	`target`

2. Transformāciju algoritma formāls apraksts

Darbā ir izstrādāts *Eclipse* ietvara spraudnis (*Eclipse framework plugin*) ar kura palīdzību *MOLAs* modeļa dati tiek pārnesti uz *EMF* modeli.



Attēls 2.1. Meta modeļu salīdzinājums dažādos meta līmeņos.

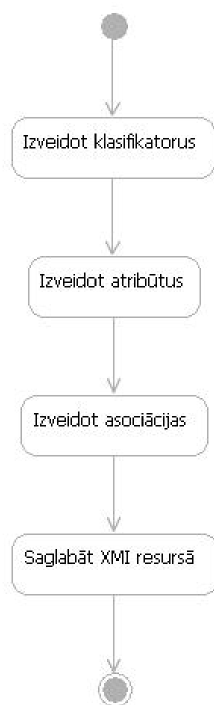
Ka redzams no attēla 2.1. *MOLA* modelis ir *M1* līmeņa modelis. Transformācijas rezultātā tiek izveidota ekvivalents *EMF* modelis. Šajā ziņā, *MOLA* meta modelis ir ieejas (*source*) meta modelis. Bet, *EMF* meta modelis ir izejas (*target*) meta modelis.

Tik pieņemts, ka *MOLA* meta modelis un *EMF* meta modelis ir līdzvērtīgi.

Algoritmam ir nepieciešami sekojoši ieejas parametri:

- Pārveidojamais modelis valodā *MOLA*.
- *EMF* meta modelis kas ir līdzvērtīgs pārveidotā modeļa meta modelim.
- Referenšu serializācijas algoritms.
- Izejas datnes nosaukums. Kur saglabāt jauno *EMF* modeli.

Algoritma secība tika aprakstīta ar sekojošu aktivitātes diagrammu:



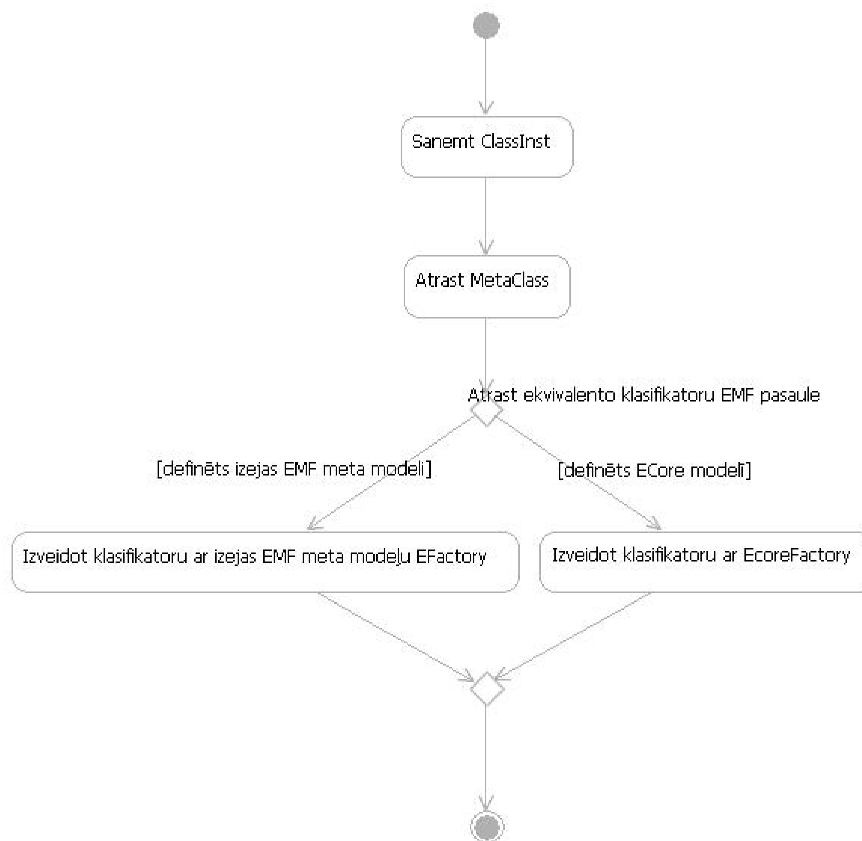
Attēls 2.1. Transformācijas algoritma aktivitātes diagramma.

2.1. Izveidot klasifikatorus

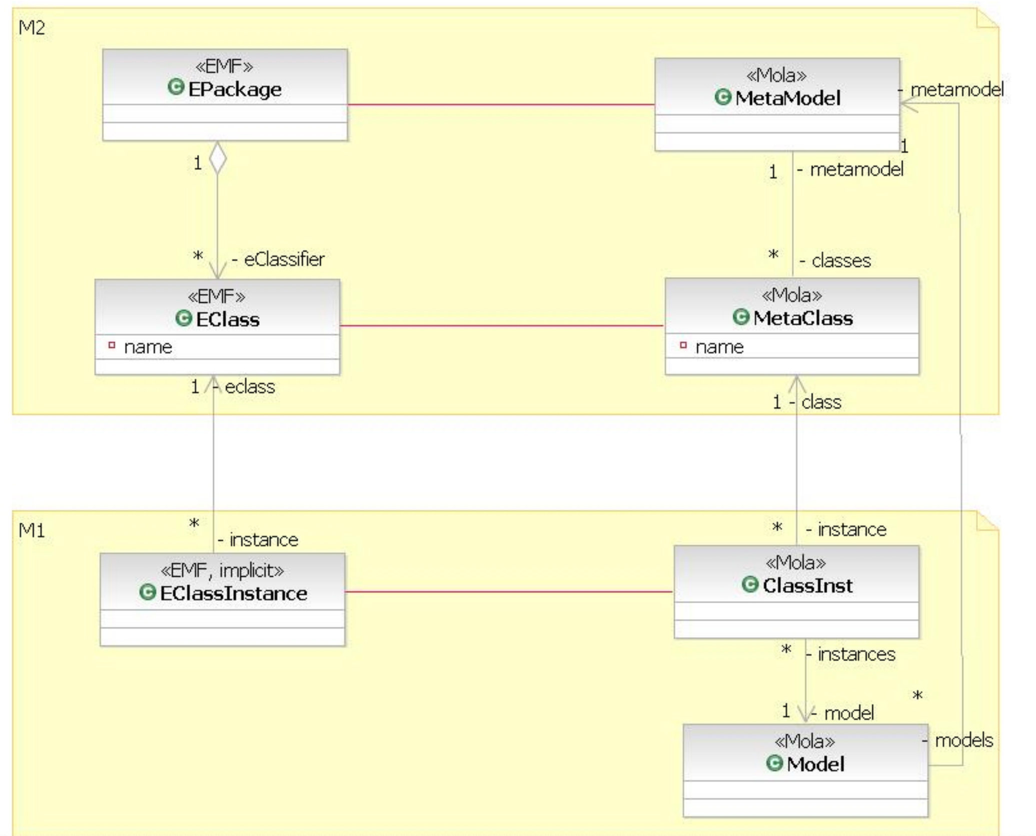
Šī etapa galvenais mērķis – katrai *MOLA* `ClassInst` instancei izveidot ekvivalentu *EMF* klasifikatoru.

Tiek veikta iterācija pa visiem objektiem pa apstrādājamo ieejas *MOLA* modeli. Katrai *MOLA* klasei tiek izveidots ekvivalents klasifikators *EMF* pasaulē.

Attēlā 2.2. ir paradīta izveidot klasifikatoru aktivitātes diagrammu.



Attēls 2.2. Izveidot klasifikatoru aktivitātes diagramma.



Attēls 2.3. Izveidot klasifikatoru konceptuāla diagramma (klasifikators ir definēts izejas EMF meta modelī).

Attēlā 2.3. var redzēt izveidot klasifikatoru konceptuālo diagrammu. Šajā diagrammā, var redzēt svarīgākas koncepcijas *MOLA* un *EMF* pasaulē. Specialas attēlojuma (*mapping*) asociācijas tiek izmantotas, lai sasaistītu ekvivalentus koncepcijas dažādos meta līmeņos.

Transformācija notiek *M1* līmenī, katram *MOLA* `ClassInst` objektam ir veidots ekvivalents `EClassInstance` objekts. `EClassInstance` ir konceptuāla klase ar *implicit* atslēgvārdu (*keywords*), un apzīmē *M1* līmeņa `EObject` objektu kurš ir atbilstoša *M2* līmeņa `EClass` instance.

Lai izveidotu jauno `EClassInstance` ir nepieciešams atrast ekvivalentus klasifikatorus *M2* līmenī. Ka ir redzams attēlā 2.3., *MOLA* `MetaClass` atbilst *EMF EClass*. Ir jāatceras, ka *EClass* ir *M2* līmeņa objekts, un, dotajā diagrammā, pārstāvē nevis konkrēto *EClass* objektu ka tādu, bet tos visus klasifikatorus kuri ir nodefinēti izejas *EMF* meta modelī (šajā diagrammā tas ir *EPackage*). Līdzīgi ir ar *MOLA M2*

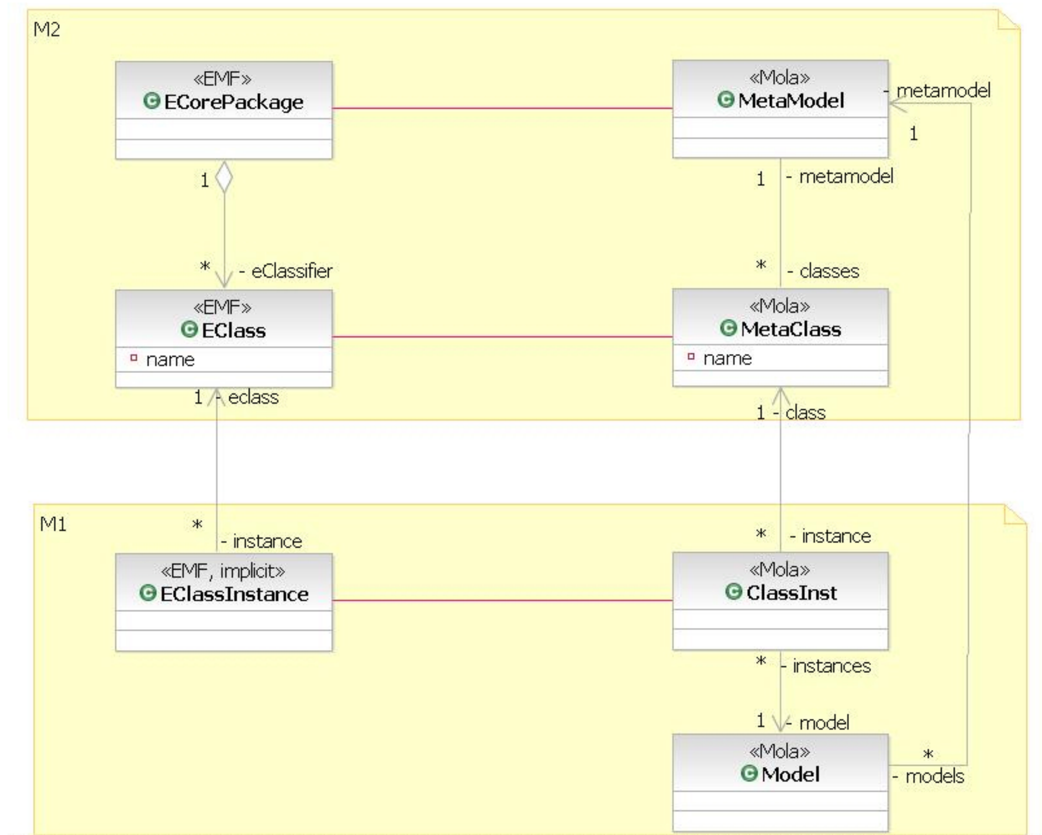
līmeņa koncepcijām, kur, piemēram, *MOLA* `MetaClass` reprezentē klasifikatorus nodefinētus ieejas *MOLA* meta modelī.

EClass meklēšana notiek ar `getEClassifier()` metodi no *`EPackage`* klases. Kā parametrs tiek dota atbilstoša *`MetaClass`* atribūta *`name`* vērtība. Līdz ar to, ka ieejas un izejas meta modeli ir līdzvērtīgi, tad katram *MOLA MetaClass* objektam vienmēr var atrast ekvivalento *EMF EClass* objektu ar doto *name* atribūta vērtību.

Ja kādam objektam nevar atrast atbilstoša ekvivalentu, tad tas tiek ignorēts.

Lai izveidotu jauno *`EObject`* (konceptuāli – *`EClassInstance`*) objektu ir nepieciešams izsaukt `create()` metodi no *`EFactory`*, kurš kā parametru saņem tikko atrasto meta klasi *`EClass`*.

Vajag atzīmēt to faktu, ka meklējamais klasifikators varētu būt nodefinēts vai norādīts izejas *EMF* meta modelī, vai tas varētu būt nodefinēts *Ecore* modelī, kura elementi šajā gadījumā ir pārnesti uz vienu līmeni zemāk (no *M3* uz *M2*). Atbilstoši, pirmajā gadījumā, mēs strādājām ar izejas *EMF* meta modeli *`EPackage`* un *`EFactory`* klasēm (attēls 2.3.), bet, otrajā gadījumā, strādājām ar *Ecore* modeļu *`EPackage`* un *`EFactory`* instances klasēm (attēls 2.4.).



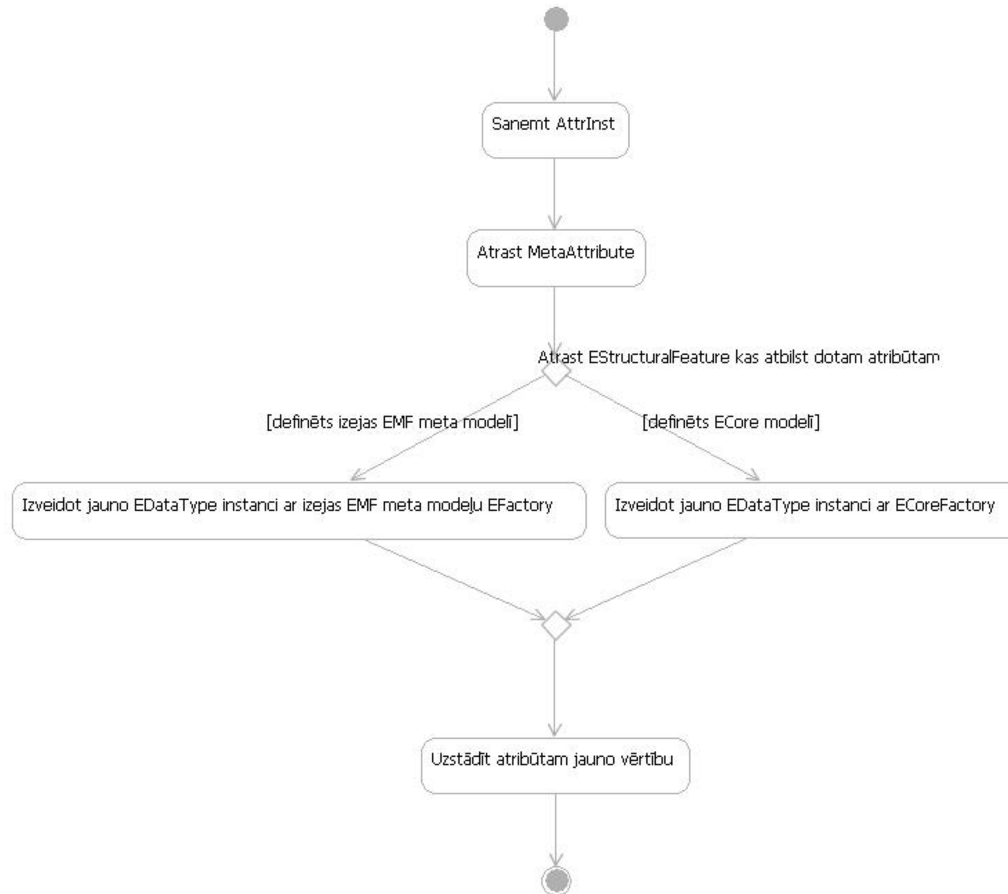
Attēls 2.4. Izveidot klasifikatoru konceptuāla diagramma (klasifikators ir definēts *ECorePackage* modelī).

Piemērs,

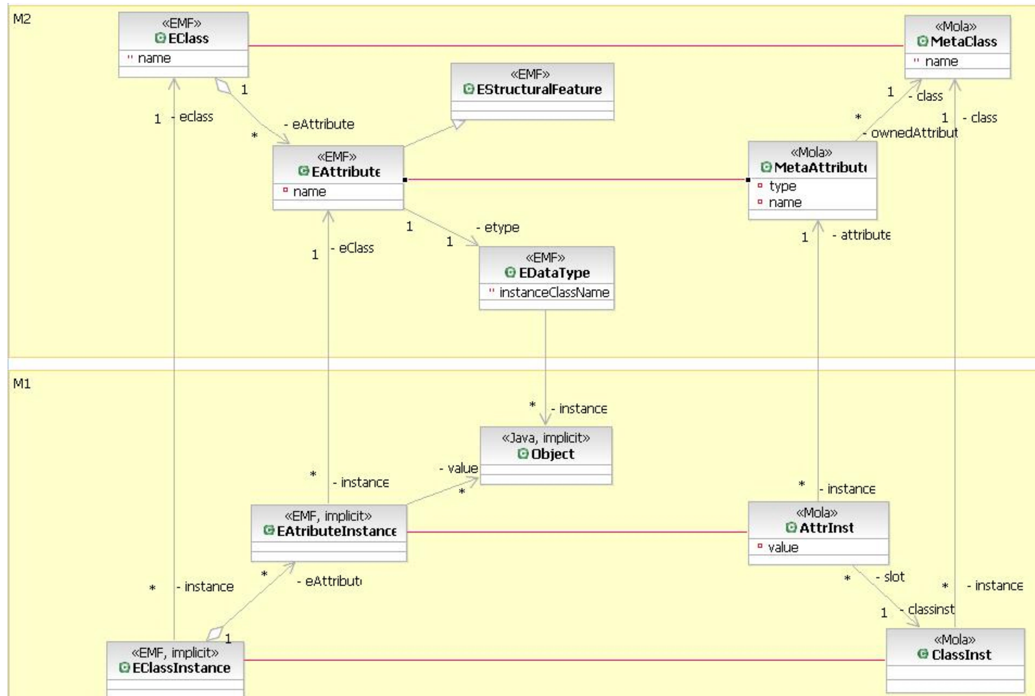
```
private EObject createEObject(String metaClassName)
{
        EClass eClass = (EClass) eMetaPackage.getEClassifier(metaClassName);
        if (eClass != null)
        {
                return eMetaPackage.getEFactoryInstance().create(eClass);
        }
        eClass = (EClass) EcorePackage.eINSTANCE.getEClassifier(metaClassName);
        if (eClass != null)
        {
                return EcoreFactory.eINSTANCE.create(eClass);
        }

        System.out.println("Not find " + metaClassName);
        return null;
}
```

2.2. Izveidot atribūtus

Izveidojot visus klasifikatorus, nākamais solis ir izveidot tām atribūtus. Katram dotam atribūtam *MOLA* modelī klasei tiek izveidots ekvivalents atribūts *EMF* modelī.



Attēls 2.5. Izveidot atribūtus aktivitātes diagramma.



Attēls 2.6. Izveidot atribūtus konceptuāla diagramma.

Attēlā 2.6. var redzēt izveidot atribūtus konceptuālo diagrammu. Šajā diagrammā, var redzēt svarīgākas koncepcijas *MOLA* un *EMF* pasaulē. Specialas attēlojuma (*mapping*) asociācijas tiek izmantotas, lai sasaistītu ekvivalentus koncepcijas dažādos meta līmeņos.

Transformācija notiek *M1* līmenī, kur katram *MOLA AttrInst* tiek veidots ekvivalents *EMF EAttributeInstance*. *EAttributeInstance* ir konceptuāla klase ar *implicit* atslēgvārdu (*keywords*), un apzīmē *M1* līmeņa `EObject` objektu kurš ir atbilstoša *M2* līmeņa `EAttribute` instance.

Lai izveidotu jauno *EAttributeInstance* ir nepieciešams atrast ekvivalentus klasifikatorus *M2* līmenī. Ka ir redzams attēlā 2.6., *MOLA* `MetaAttribute` atbilst *EMF* `EAttribute`. Ir jāatceras, ka `EAttribute` ir *M2* līmeņa objekts, un, dotajā diagrammā, pārstāvē nevis konkrēto `EAttribute` objektu ka tādu, bet tos visus atribūtus kuri ir nodefinēti sasaistītam *EClass* klasifikatoram izejas *EMF* meta modelī. Līdzīgi ir ar *MOLA M2* līmeņa koncepcijām, *MOLA MetaAttribute* šajā diagrammā reprezentē sasaistītus *MetaClass* atribūtus ieejas *MOLA* meta modelī.

Ar `getEStructureFeature()` metodi var pēc nosaukuma (šajā gadījumā, tas ir apstrādāto *MetaAttribute* atribūta `name` vērtība) atrast *EAttribute*

objektu kas reprezentē doto atribūtu. Līdz ar to, ka ieejas un izejas meta modeli ir līdzvērtīgi, tad katram *MOLA MetaAttribute* objektam var vienmēr atrast ekvivalento *EMF EAttribute* objektu ar doto *name* atribūta vērtību.

No *EAttribute* var uzzināt *EDataType* - dotā atribūta tipu. Tālāk, izmantojot `createFromString()` metodi no `EFactory` klasi, var izveidot Object objektu - jaunu instance šim datu tipam. Jaunam datu tipam tik piešķirta *AttrInst* atribūta *value* vērtība.

Ar `eSet()` metodi var uzlikt dotam atribūtam šo jauno vērtību. Ir svarīgi cik doto atribūta instances varētu būt šim objektam. Ja ir vairāk nekā viena tad tiek strādāts ar `List` objektu.

Līdzīgi kā iepriekšējā gadījumā, vajag atzīmēt to faktu, ka meklējamais datu tips varētu būt nodefinēts vai noradīts izejas *EMF* meta modelī, vai tas varētu būt nodefinēts *Ecore* modelī, kura elementi šajā gadījumā ir pārnesti uz vienu līmeni zemāk (no *M3* uz *M2*). Atbilstoši, pirmajā gadījumā, mēs strādājām ar izejas *EMF* meta modeli *`EPackage`* un *`EFactory`* klasēm, bet, otrajā gadījumā, strādājām ar *Ecore* modeļu *`EPackage`* un *`EFactory`* instances klasēm.

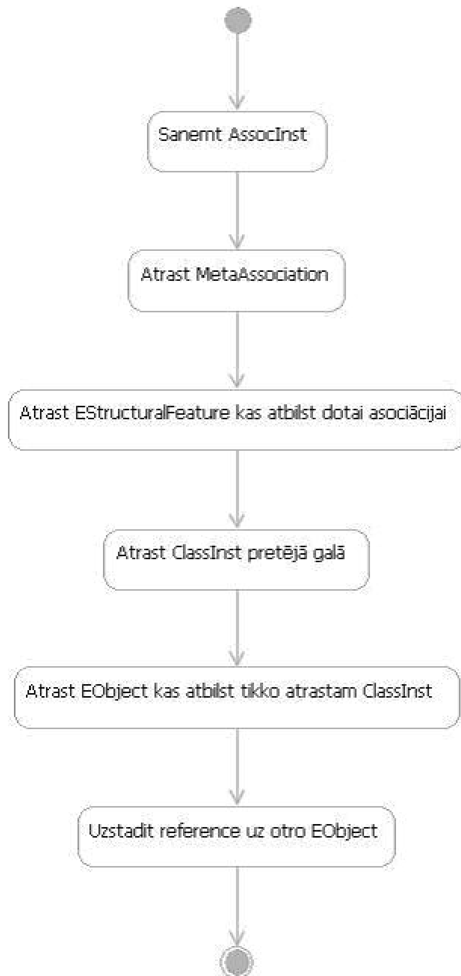
Piemērs,

```
private void createAttributes(EObject eObject, ClassInst classInstSrc)
{
	System.out.println("createAttributes" + classInstSrc);

	EClass metaClass = eObject.eClass();

	List attrInstances = classInstSrc.findAllAttrInst();

	for (Iterator iter = attrInstances.iterator(); iter.hasNext();)
	{
		AttrInst attrInst = (AttrInst) iter.next();

		MetaAttribute metaAttribute = attrInst.findMetaAttribute();

		if (metaAttribute == null)
		{
			continue;
		}

		EStructuralFeature eStructuralFeature = metaClass
				.getEStructuralFeature(metaAttribute.getName());

		if (eStructuralFeature == null)
		{
			continue;
		}

		EDataType eDataType = (EDataType) eStructuralFeature.getEType();
		Object value = createDataTypeValue(eDataType, attrInst);
		setEObjectProperty(eObject, eStructuralFeature, value);
	}
}

private Object createDataTypeValue(EDataType eDataType, AttrInst attrInst)
{
	if (belongToPackage(eMetaPackage, eDataType.getName()))
	{
		return eMetaPackage.getEFactoryInstance().createFromString(
				eDataType, attrInst.getValue());
	}

	if (belongToPackage(EcorePackage.eINSTANCE, eDataType.getName()))
	{
		return EcoreFactory.eINSTANCE.createFromString(eDataType, attrInst
					.getValue());
	}

	return null;
}
```

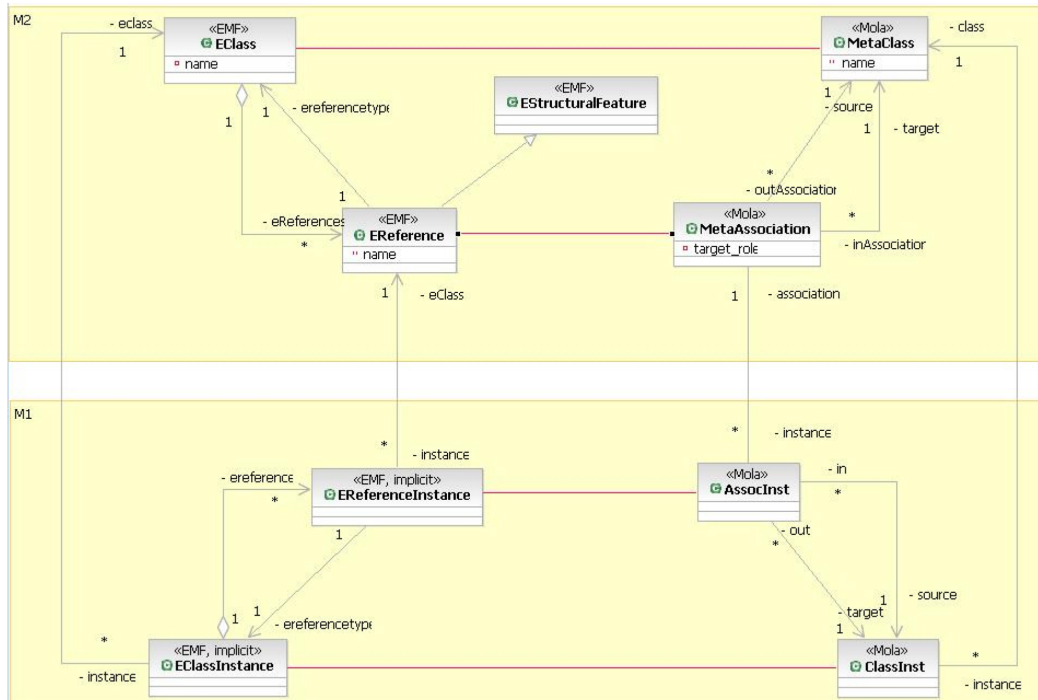
```
private void setEObjectProperty(EObject eObject, EStructuralFeature sf,
                Object object)
{
        if (sf.isMany())
        {
                List list = (List) eObject.eGet(sf);
                list.add(object);
        }
        else
        {
                eObject.eSet(sf, object);
        }
}
```

2.3. Izveidot asociācijas

Izveidojot visus klasifikatorus un atribūtus, nākamais solis ir izveidot visas relācijas starp klasifikatoriem. Katrai dotajai asociācijai *MOLA* modelī tika meklēta ekvivalenta asociācija *EMF* modelī.



Attēls 2.7. Izveidot asociācijas aktivitātes diagramma.



Attēls 2.8. Izveidot asociācijas konceptuāla diagramma.

Attēlā 2.8. var redzēt izveidot asociācijas konceptuālo diagrammu. Šajā diagrammā, var redzēt svarīgākas koncepcijas *MOLA* un *EMF* pasaulē. Specialas attēlojuma (*mapping*) asociācijas tiek izmantotas, lai sasaistītu ekvivalentus koncepcijas dažādos meta līmeņos.

Transformācija notiek *M1* līmenī, kur katram *MOLA MetaAssociation* tiek veidots ekvivalents *EMF EReferenceInstance*. *EReferenceInstance* ir konceptuāla klase ar *implicit* atslēgvārdu (*keywords*), un apzīmē *M1* līmeņa `EObject` objektu kurš ir atbilstoša *M2* līmeņa *EReference* instance. Vajag atzīmēt, ka katram atsevišķam *MetaClass* objektam tiek aplūkotas tikai izejošas asociācijas, citiem vārdiem, tikai tas asociācijas kuram dotais *MetaClass* objekts ir *source* atribūta vērtība.

Lai izveidotu jauno *EReferenceInstance* ir nepieciešams atrast ekvivalentus klasifikatorus *M2* līmenī. Ka ir redzams attēlā 2.6., *MOLA MetaAssociation* atbilst *EMF EReference*. Ir jāatceras, ka *EReference* ir *M2* līmeņa objekts un, dotajā diagrammā, pārstāvē nevis konkrēto *EReference* objektu ka tādu, bet tas visas asociācijas kuras ir nodefinētas sasaistītam *EClass* klasifikatoram izejas *EMF* meta modelī. Līdzīgi ir ar *MOLA M2* līmeņa koncepcijām, *MOLA MetaAssociation* šajā diagrammā reprezentē sasaistītus *MetaClass* asociācijas ieejas MOLA meta modelī.

Ar `getEStructureFeature()` metodi var pēc nosaukuma (šajā gadījumā, *`target_role`* atribūta vērtība no atbilstoša *MetaAssociation*) atrast *EReference* objektu kas reprezentē doto asociāciju. Līdz ar to, ka ieejas un izejas meta modeli ir līdzvērtīgi, tad katram *MOLA MetaAssociation* objektam var vienmēr atrast ekvivalento *EMF EReference* objektu ar doto *name* atribūta vērtību. Tiek izmantota *MOLA MetaAssociation target_role* atribūta vērtība, jo tieši šī vērtība veido loma nosaukumu dotai asociācijai.

Tālāk ir nepieciešams atrast ekvivalentu klasifikatoru asociācijas otrajā galā. Kā jau zināms, visi klasifikatori jau ir izveidoti pirmajā etapā. Meklēšana notiek pēc apstrādāta *AssocInst target* atribūta vērtības.

Ar `eSet`() metodi var nodefinēt dotai asociācijai vērtību. Ir svarīgi cik daudz ir tāda tipa asociācijas šim objektam. Ja ir vairāk nekā viens, tad tiek strādāts ar `List` objektu.

Līdzīgi kā iepriekšējā gadījumā, vajag atzīmēt to faktu, ka meklējama asociācija varētu būt nodefinēta vai noradīta izejas *EMF* meta modelī, vai tas varētu būt nodefinēta *Ecore* modelī, kura elementi šajā gadījumā ir pārnesti uz vienu līmeni zemāk (no *M3* uz *M2*). Atbilstoši, pirmajā gadījumā, mēs strādājām ar izejas *EMF* meta modeli *`EPackage`* un *`EFactory`* klasēm, bet, otrajā gadījumā, strādājām ar *Ecore* modeļu *`EPackage`* un *`EFactory`* instances klasēm.

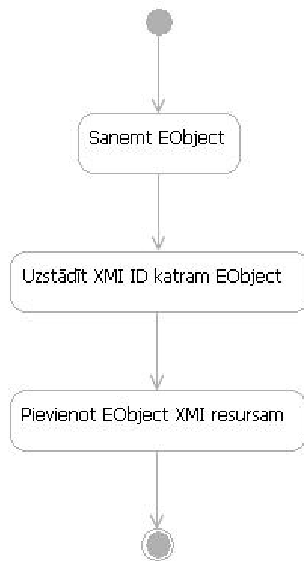
Piemērs,

```
private void createRelations()
{
        for (Iterator iter = classInst2EObjectMap.entrySet().iterator(); iter
                                .hasNext();)
        {
                Map.Entry element = (Map.Entry) iter.next();

                ClassInst classInst = (ClassInst) element.getKey();
                EObject eObject = (EObject) element.getValue();

                createRelations(eObject, classInst);
        }
}

private void createRelations(EObject eObject, ClassInst classInstSrc)
{
        System.out.println("createRelations" + classInstSrc);

        EClass metaClass = eObject.eClass();

        List associations = classInstSrc.findAllSourceAssociation();

        for (Iterator iter = associations.iterator(); iter.hasNext();)
        {
                AssocInst assocInst = (AssocInst) iter.next();

                MetaAssociation metaAssociation = assocInst.getMetaAssociation();

                if (metaAssociation == null)
                {
                        continue;
                }

                EStructuralFeature eStructuralFeature = metaClass

        .getEStructuralFeature(metaAssociation.getTrgRoleName());

                ClassInst classInstTrg = assocInst.getTrgClassInst();

                if (classInstTrg == null)
                {
                        continue;
                }

                Object eObjectTrg = classInst2EObjectMap.get(classInstTrg);

                setEObjectProperty(eObject, eStructuralFeature, eObjectTrg);
        }
}
```

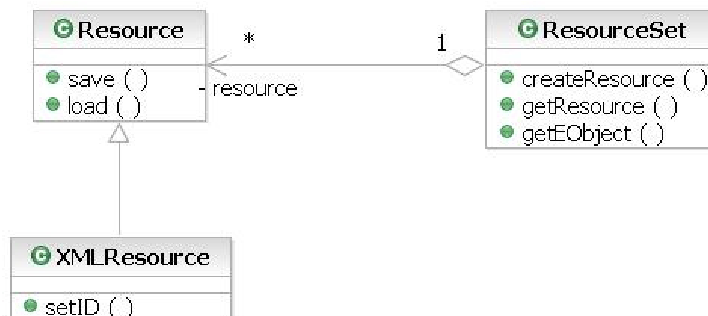
```
private void setEObjectProperty(EObject eObject, EStructuralFeature sf,
               Object object)
{
       if (sf.isMany())
       {
               List list = (List) eObject.eGet(sf);
               list.add(object);
       }
       else
       {
               eObject.eSet(sf, object);
       }
}
```

2.4. Saglabāt XMI resursu

Lai saglabātu modeli tika izmantotas *EMF Persistence API*. *Persistence API* pamatā ir četri saskarnes: `Resource`, `ResourceSet`, `Resource.Factory` un `URIConvertor`. *EMF* nodrošina darbu ar *XML* un *XMI* resursiem. Lai saglabātu modeli *XMI 2.0* formātā tika izmantots `XMIResourceImpl`.



Attēls 2.9. Saglabāt *XMI* resursu aktivitātes diagramma.



Attēls 2.10. *Ecore* resursu klases.

Uzstādīt *XMI ID* katram `EObject`

Gadījumā, kad ir svarīgi kāds mehānisms tiek izmantots objekta identifikācijai *XMI* datnēs, tad vajag uzstādīt *XMI ID* katram `EObject`. Otrais mehānisms šeit būtu *XMI PATH* izteiksme.

Pievienot `EObject` *XMI* resursam.

Lai pievienotu `EObject` resursam tiek izmantota `getContents`() metode `XMIResourceImpl` klasi. Attiecīgi, ja ir nepieciešams pievienot jaunu objektu *XMI* resursā, tad ir nepieciešams to pievienot šim sarakstam. Objekts ir arī saglabāts tad, ja tam ir ietvēruma asociācija (*containment association*) ar kādu citu objektu, kurš jau ir šajā sarakstā. Līdz ar to ir nepieciešams pievienot tikai tos resursus, kuriem nav ietvēruma asociācija ar kādu citu objektu un tas ir gadījums kad `eObject.eContainer`() ir tukš.

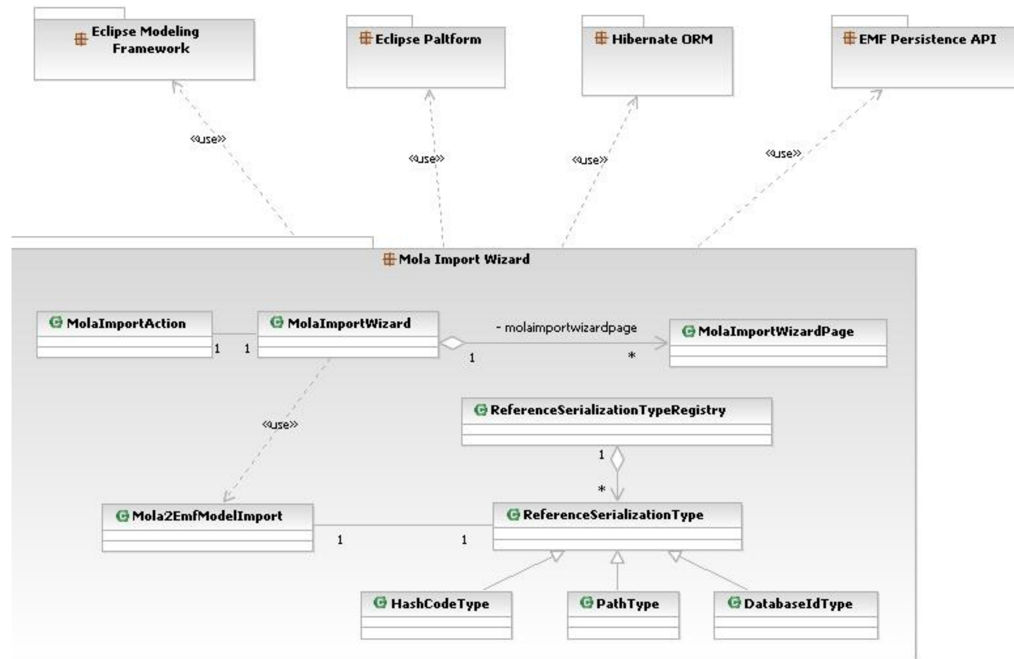
Piemērs,

```
private void saveResource() throws IOException
{
        ResourceSet resourceSet = new ResourceSetImpl();
        resourceSet.getResources().add(resource);

        Map options = new HashMap();
        options.put(XMLResource.OPTION_USE_ENCODED_ATTRIBUTE_STYLE,
                        Boolean.TRUE);

        for (Iterator iter = getSortedEObjectList().iterator(); iter.hasNext();)
        {

                EObject eObject = (EObject) iter.next();

                ClassInst classInst = (ClassInst) eObject2classInstMap.get(eObject);

                String id = referenceSerializationType.getID(classInst, eObject);
```

```
                if (id != null)
                {
                        resource.setID(eObject, id);
                }

                if (isRootElement(eObject))
                {
                        resource.getContents().add(eObject);
                }
        }
        resource.save(options);
}
private boolean isRootElement(EObject eObject)
{
        return eObject.eContainer() == null;
}
```

3. Praktiska realizācija

Darbā ir izstrādāts *Eclipse* ietvara spraudnis (*Eclipse framework plugin*) ar kura palīdzību *MOLA* modeļa dati var tikt pārnesti uz *EMF* modeli.

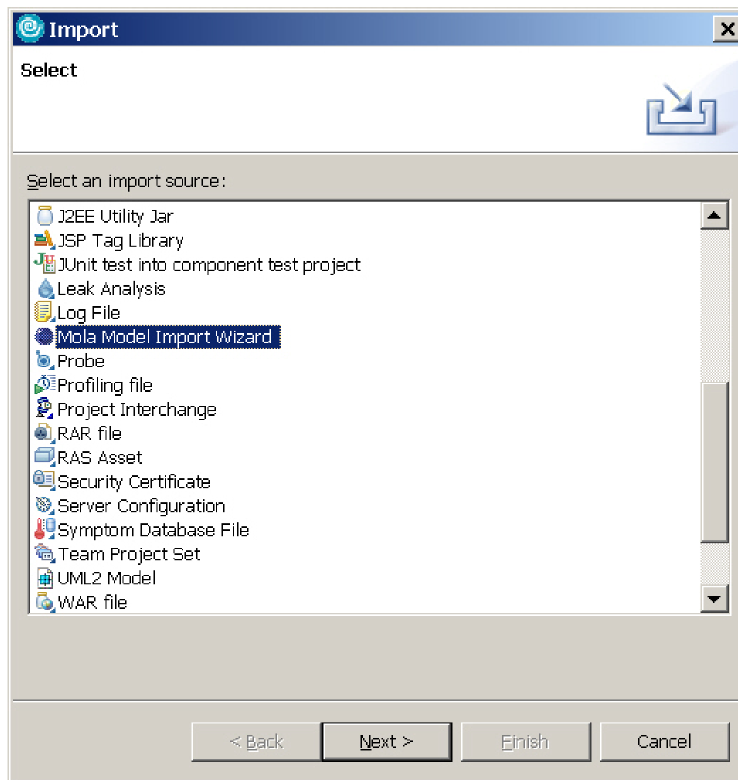


Attēls 3.1. *Mola Import Wizard* arhitektūra.

3.1. Lietotāju saskarne

Doto *Eclipse* ietvara spraudni var izpildīt divos veidos:

- Kā *Import Wizard* izvēloties "*Mola Model Import Wizard*".

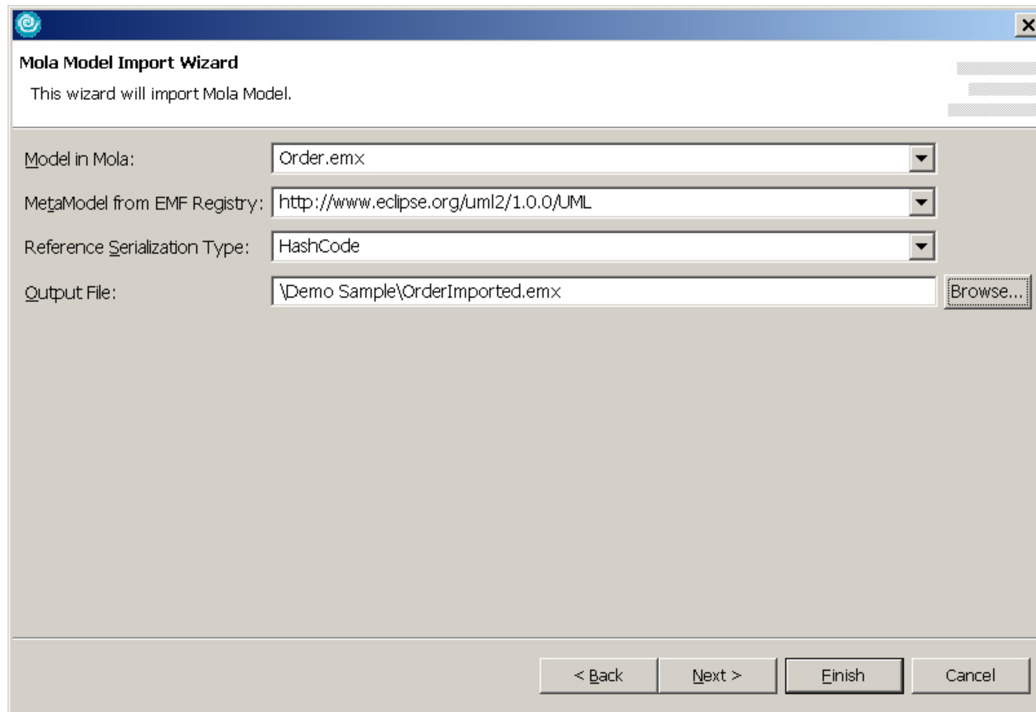


Attēls 3.2. *Import Wizard* logs.

- No izvēlnes izvēloties "*Import From Mola Wizard*" no "*Mola*" izvēlnes.

Attēls 3.3. "*Import From Mola Wizard*" izvēlne.

Mola Model Import Wizard sastav no diviem logiem. Pirmajā logā (attēls 3.4.) var specificēt galvenos parametrus modeļa importam. Otrajā logā (attēls 3.6.) var nodefinēt galveno *MOLA* modeli, to modeli kas tiek rakstīta pa priekšu *XMI* datnē.



Attēls 3.4. *Mola Model Import Wizard*, pirmais ekrāns.

Model in Mola

Šajā sarakstā ir visi modeļi kuri ir pieejami *MOLA* (dati tiek lasīti no *Model* tabulas). Ir jāizvēlas to *MOLA* modeli kurš ir domāts importam *EMF*.

MetaModel from EMF Registry

Šajā sarakstā ir visi meta modeļi kas ir pieejami no *EMF Registry*. Ir jāizvēlas tas *MetaModelis* no *EMF Registry* kurš atbilst importētam modelim.

Noklusētā vērtība – http://www.eclipse.org/uml2/1.0.0/UML, kas atbilst *UML2* meta modelim.

Reference Serialization Type

Ir jāizvēlas referenšu serializācijas tips. Pagaidām tiek piedāvāti sekojoši varianti (pirmie divi varianti ir plaši lietoti modeļa kodēšanai *EMF* vidē):

Path – *XMI Path* izteiksme. Noklusētais standarts. Identifikatoru kodēšana ar hierarhiskām referencēm un instanču kartes numuriem atbilstošajā klasē.

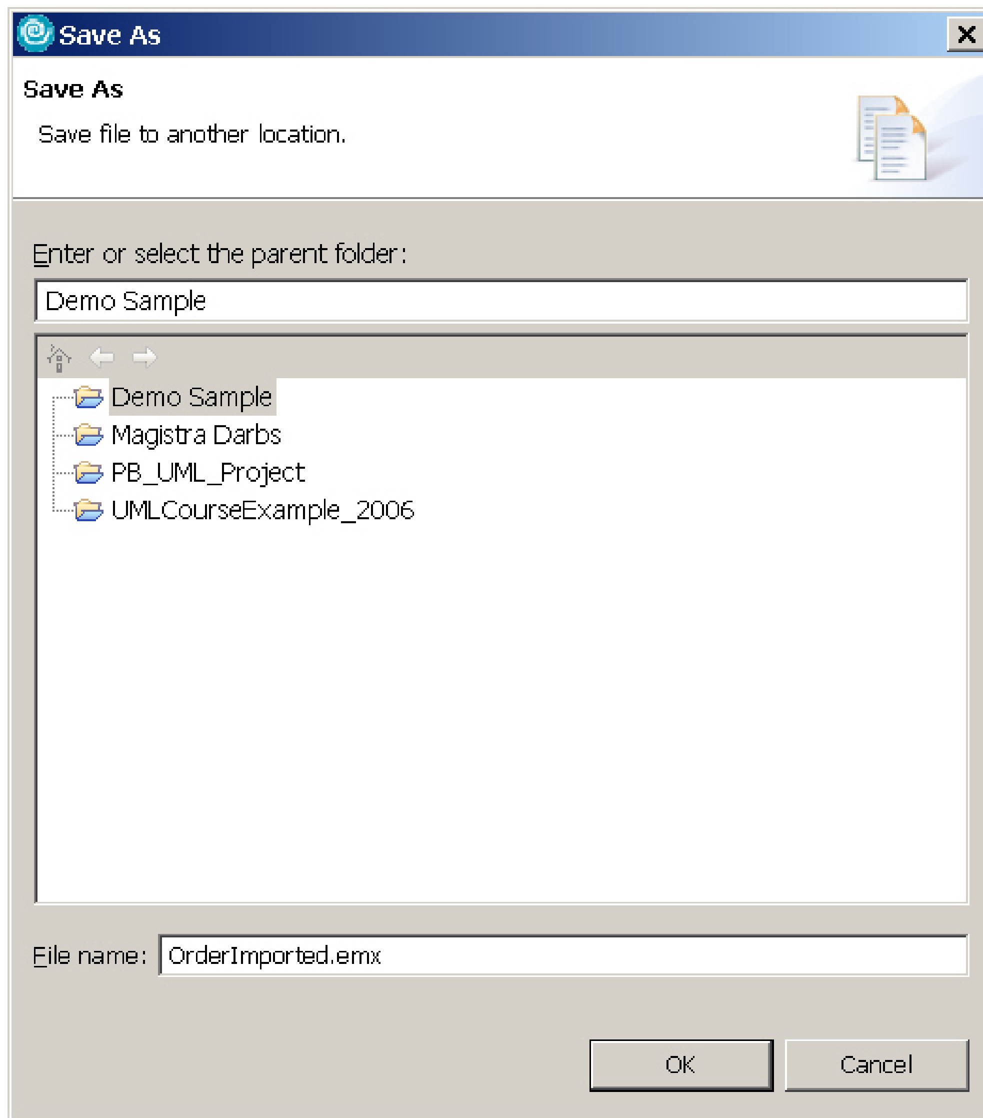
HashCode – Atbilstošo *EObjectu hashcode*() vērtība. Unikālais identifikators, kādā mērā tas ir ekvivalents *GUID* (***G**lobally **U**nique **Id**entifier*) kas tiek plaši lietots *RSA* un citos modelēšanas rīkos.

DataBaseId – Atbilstošo *ClassInst ID*. Diezgan vienkārša implementācija, tik izmantots atbilstoša instances datubāzes identifikators.

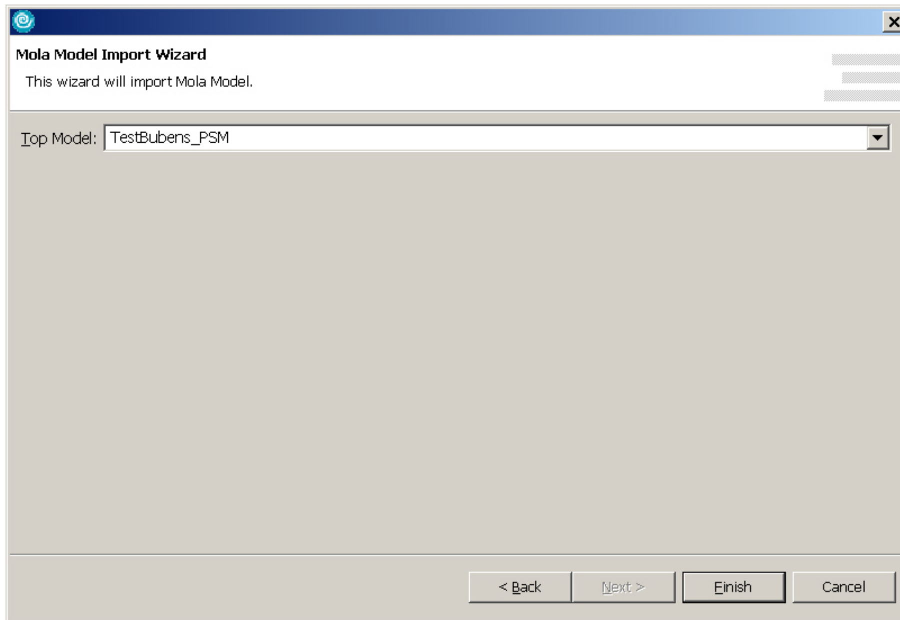
Noklusēta vērtība – “*HashCode*”.

Output File

Izejas datnes nosaukums. Lai izvelēties datne ir nepieciešams nospiest pogu *Browse* un tad atveras *Save As...* logs (attēls 3.5.).



Attēls 3.5. Izejas datnes izvēles dialogs.



Attēls 3.6. *Mola Model Import Wizard*, otrais ekrāns. Galveno modeļu izvēlne.

Top Model

Galvenais modelis iekš dota modeļi. Tas ir gadījums kad modelis sastāv no vairākiem modeļiem. Ar šo opciju var noradīt to modeli kas tiek rakstīta pa priekšu *XMI* datnē. Lielākā mērā tas ir *RSA* rīka prasība.

3.2. Eclipse spraudņa konfigurācija

3.2.1. *Overview* skats

Šajā skatā var norādīt identifikatoru, versijas numuru, nosaukumu, un implementēšanas klasi.

Attēls 3.7. *Overview* skats.

3.2.2. *Dependencies* skats

Dependencies skatā var norādīt kuri ietvara spraudni ir nepieciešami un ir izmantoti ar doto ietvaru spraudni.

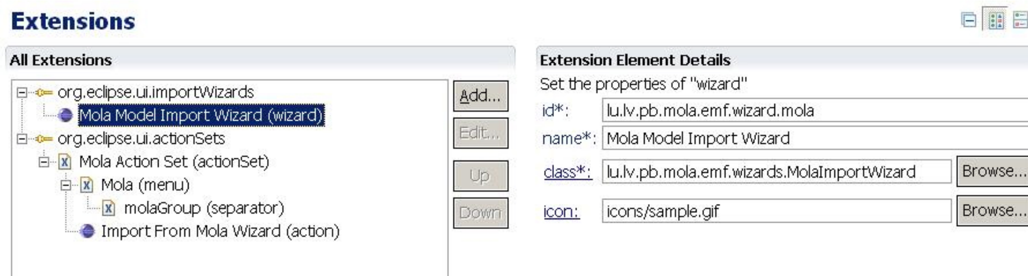
Attēls 3.8. *Dependencies* skats.

3.2.3. *Extensions* skats

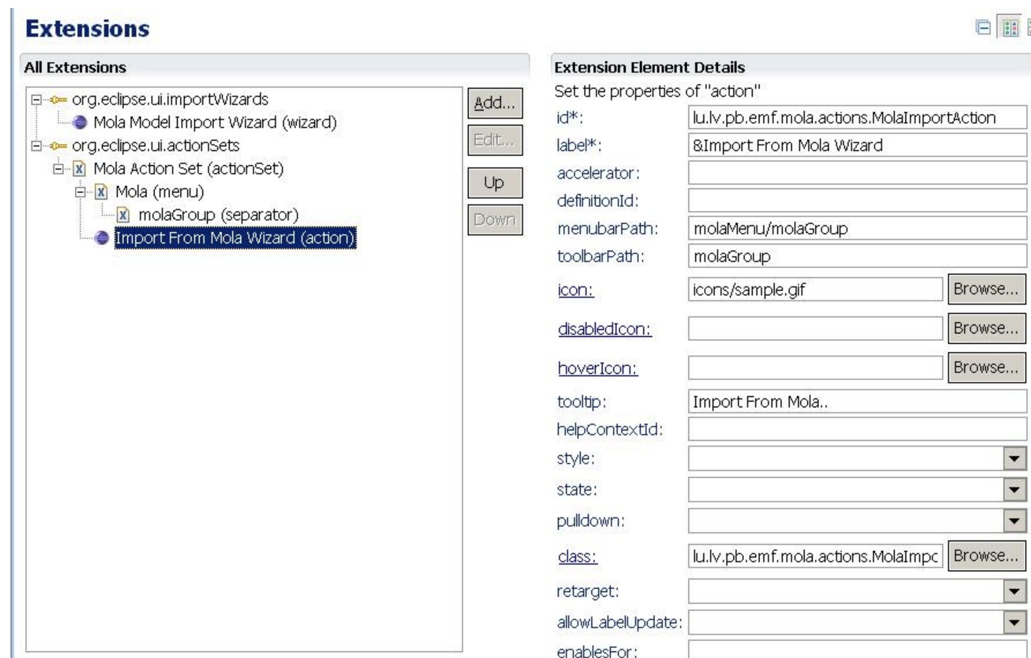
Extension skatā var nodefinēt tos paplašinājuma punktus, kuri tiek paplašināti ar doto spraudni. Kā redzams, izstrādātais spraudnis paplašina *org.eclipse.ui.importWizards* un *org.eclipse.ui.actionSets* funkcionalitātes punktus.

org.eclipse.ui.importWizards – paplašinājums *Import Wizard* funkcionalitātes punktam (attēls 3.2. un 3.9.).

org.eclipse.ui.actionSets – paplašinājums *Workbench* izvēlnes funkcionalitātes punktam (attēls 3.3. un 3.10.).



Attēls 3.9. *Extensions* skats (*org.eclipse.ui.importWizard*).



Attēls 3.10. *Extensions* skats (*org.eclipse.ui.actionSets).*

3.2.4. *Plugin.xml* datne saturs

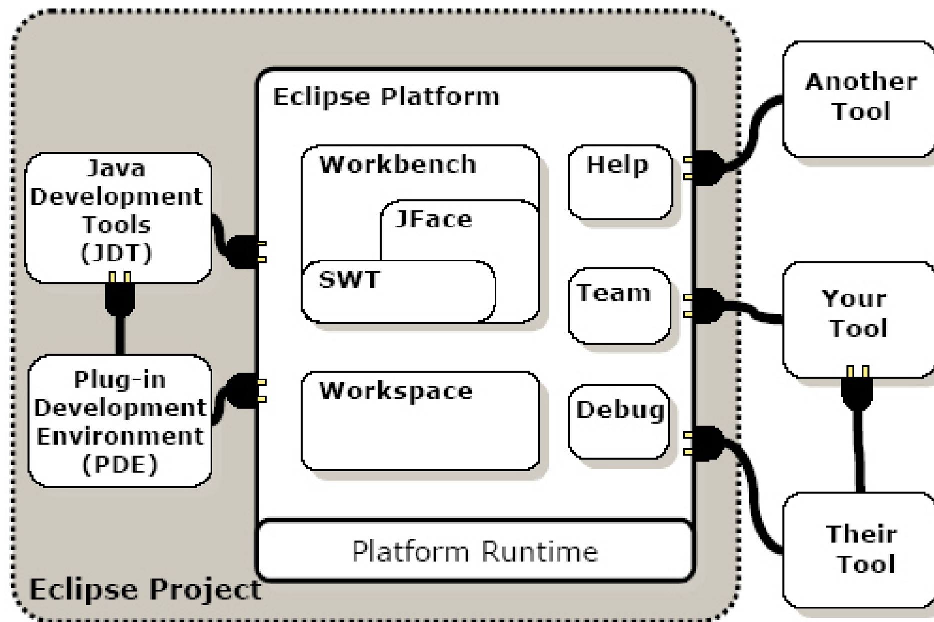
```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension
         point="org.eclipse.ui.importWizards">
      <wizard
            class="lu.lv.pb.mola.emf.wizards.MolaImportWizard"
            icon="icons/sample.gif"
            id="lu.lv.pb.mola.emf.wizard.mola"
            name="Mola Model Import Wizard"/>
   </extension>
   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            id="lu.lv.pb.emf.mola.actionSet"
            label="Mola Action Set"
            visible="true">
         <menu
               id="molaMenu"
               label="&amp;Mola">
            <separator name="molaGroup"/>
         </menu>
         <action
               class="lu.lv.pb.emf.mola.actions.MolaImportAction"
               icon="icons/sample.gif"
               id="lu.lv.pb.emf.mola.actions.MolaImportAction"
               label="&amp;Import From Mola Wizard"
               menubarPath="molaMenu/molaGroup"
               toolbarPath="molaGroup"
               tooltip="Import From Mola.."/>
      </actionSet>
   </extension>

</plugin>
```

3.3. Eclipse Platformas arhitektūra

Mūsdienās *Eclipse Platforma* visvairāk ir zināma ka viens no labākiem *Java* valodas izstrādāšanas rīkiem (*IDE*). Bet, ka jau bija teikts, *Eclipse Platforma* netikai ir pamats lai konstruētu šāda tipa riku (*IDE*), bet arī ir pamats lai konstruētu cita tipa aplikācijas un rīkus. Pat neobligāti saistītus ar programmatūras izstrādāšanas uzdevumiem. Piemēram, *Eclipse RCP Platforma* varētu but izmantota lai veidotu komplicētus aplikācijas citās sfērās [8].

Eclipse Platformu pamatā ir mikro kodols *Platformu Runtime*. Kaut gan, tajā ir diezgan daudz jau iebūvētu funkcionalitāte, lielākā daļa no tā ir ļoti abstrakta. Visa funkcionalitāte tiek apgādāta ar *Eclipse* ietvara spraudniem.

Pati par sevi *Eclipse Platforma* sastāv no vairākām apakšu sistēmām.



Attēls 3.10. *Eclipse* Platformas arhitektūra.

Platformu Runtime galvenais uzdevums ir atklāt kuri *Eclipse* ietvara spraudni ir pieejami dotajā *Eclipse* konfigurācijā. Līdz ar to ka *Eclipse* ietvaru spraudņu skaits varētu but diezgan paliels, tie ir ielādēti vai aktivizēti tikai tad kad ir nepieciešami.

Eclipse Platforma definē mehānismus kurus ir jāizmanto un likumi kuram ir jāseko lai panāktu labāku riku integrāciju.

Eclipse ietvara spraudnis

Ietvara spraudnis (*Eclipse plugin*) ir vismazākā *Eclipse* funkcionalitāte vienība kas varētu but izstrādāta un piegādāta atsevišķi. Parasti, nepārak liels rīks ir uzrakstīts ka atsevišķs ietvara spraudnis. Komplicēta rika funkcionalitāte varētu but sadalīta starp vairākiem spraudniem.

Katram *Eclipse* ietvara spraudnim ir *XML* manifestu datne kas apraksta doto spraudni. Piemēram, paplašinājumus punktus (*extension points*), kuri vai ir definēti ar dotu spraudni, vai ir nepieciešami šis spraudna darbībai.

Workbench

Eclipse lietotāju interfeisa karkass. Plaši izmanto *SWT* (*Standard Widget Toolkit*) un *JFace* bibliotēkus.

Workbench balstās uz sekojošam lietotaju saskarnes koncepcijām:

- Redaktors (*Editor*). Redaktors atļauj attvert, rediģēt, un saglabāt kādus objektus.
- Skatiens (*View*). Skatieni atspoguļo kādu informāciju par kādu objektu vai vairākām objektiem.
- Perspektīva (*Perspectives*). Apvieno vairākas redaktorus un skatienus.

Java Development Tools (JDT)

Paplašina *Workbench* ar dažādam iespējam priekš *Java* koda rediģēšanai, kompilācijai, apskatīšanai, atkļūdošanai un tā tālāk.

Plug-in Developoment Environment (PDE)

Veido dažādi rīki lai automatizēt un atvieglot jaunu ietvaru *Eclipse* spraudni izstrādāšanu.

3.4. EMF Persistence API

Lai saglabātu modeli tika izmantotas *EMF Persistence API*. *Persistence API* pamatā ir četras saskarnes: *Resource*, *ResourceSet*, *Resource.Factory* un *URIConvertor* [12].

URI (Uniform Resource Identifier)

URI ir standarts lai identificētu un atrastu dažāda tipu datus. Dati varētu atrasties kāda datnē uz lokāla diska. Ka arī tas varētu atrasties kāda cita glabātuves tipa un veida.

EMF diezgan plaši izmanto *URI*. Piemēram, lai identificētu paketes, un ka arī resursus un objektus tajos [12].

Resource

`Resource` saskarne ir `EObject` persistence konteineris. Ar *URI* palīdzību tika definēta faktiska glabāšanas vieta.

Lai pievienotu `EObject` resursam tiek izmantota `getContents`() metode `XMIResourceImpl` klasi. Attiecīgi, ja ir nepieciešams pievienot jaunu objektu *XMI* resursā, tad ir nepieciešams to pievienot šim sarakstam. Objekts ir arī saglabāts tad, ja tam ir ietvēruma asociācija (*containment association*) ar kādu citu objektu, kurš jau ir šajā sarakstā [12].

Resource.Factory

Ir izmantots lai izveidotu jaunus resursus. *Resursu Fabriku* var atrast izmantojot reģistru. Kaut gan *Resursu Fabrika* atbilst nevis konkrētam *URI*, bet *URI* kategorijai. Piemēram, iebūvētais reģistrs atļauj piesaistīt *Resursu Fabriku* kādai *URI* shēmai vai datnes paplašinājuma tipam. Pēc noklusēšanas tika izmantota *XMIResourceFactoryImpl*, un tas notiek org.eclipse.emf.ecore.xmi *Eclipse ietvara spraudņa* ielādēšanas laikā.

```
<extension point = "org.eclipse.emf.ecore.extension_parser">
  <parser type="*"
     class="org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl"/>
</extension>
```

ResourceSet

`ResourceSet` tā ir kopa ar resursiem kas bija kopa izveidoti vai ielādēti.

`createResource()` metode ir izmantota lai izveidotu jaunu, tukšu resursu šajā resursu kopa.

`getResource()` metode arī izveido jauno resursu, bet tas mēģina to ielādēt izmantojot resursa *URI*.

`getEObject()` metode atļauj nolasīt kādu `EObjectu` iekš dota resursa kopa, izmantojot *URI* fragmentu daļu.

Nav rekomendēts izveidot resursu ar `new` operatoru [12].

XMLResourceImple un XMIResourceImpl.

Nodrošina darbu ar *XML* un *XMI* resursiem.

XML saglabāšanas opcijas:

`OPTION_LINE_WIDTH`. Tips – `Integer`. Ir izmantots lai nodefinētu maksimālo vienas līnijas platumu *XML* datnē.

`OPTION_DECLARE_XML`. Tips – `Boolean`. Nosaka vai ir nepieciešams rakstīt *XML* deklarācijas daļu.

`OPTION_USE_ENCODED_ATTRIBUTE_STYLE`. Tips – `Boolean`. Ja dota opcija ir `true`, tad *EMF* vienmēr serializē referense ka *URI* iekš atribūta vērtības, nesakoties vai referense ir lokāla vai norada uz citu dokumentu.

`OPTION_SKIP_ESCAPE`. Tips – `Boolean`. Nosāka vai ir nepieciešams kodēt `“<”`, `“>”`, un `“&”` izmantojot iepriekš definētus entītijas `"<"`, `">"`, un `"&"`.

OPTION_PROCESS_DANGLING_HREF. Tips - String. Iespējamas vērtības: "*THROW*", "*RECORD*", vai "*DISCARD*".

OPTION_SCHEMA_LOCATION. Tips – Boolean. Nosaka vai ir nepieciešams rakstīt xsi:schemaLocation atribūtu dokumentā. Tas ir alternatīva pieeja *Package Register*.

OPTION_XML_MAP. Tips – XMLResource.XMLMap. Ir izmantota lai nodefinēt *Ecore* un *XMI* elementu atbilstību. Piemēram, ir iespējams nomainīt *XML* elementa nosaukumu.

OPTION_ENCODING. Tips – String. Ir izmantota lai nomainīt kodēšanas formātu. Noklusēta vērtība "*ACII*".

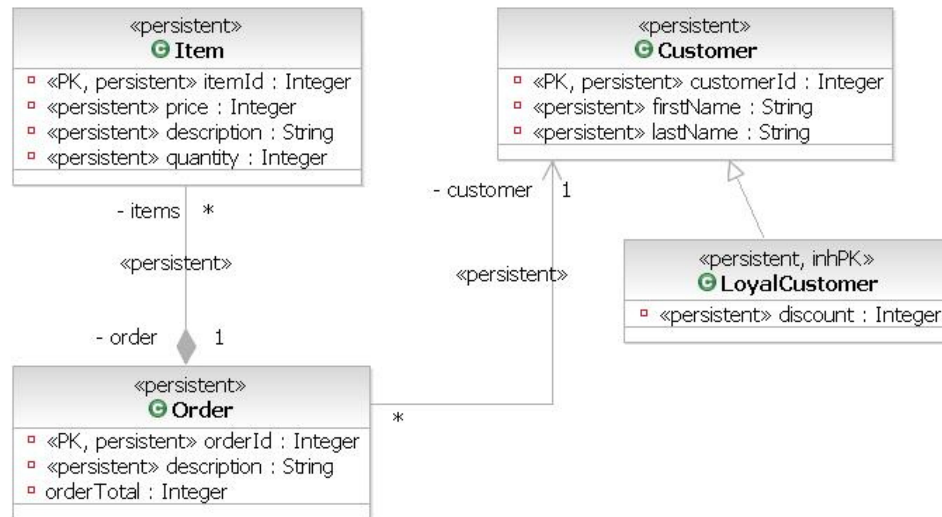
4. Demonstrācijas piemērs

Demonstrācijas piemēra uzdevums bija pierādīt, ka ar *EMF* palīdzību *MOLA* var sadarboties ar *EMF* balstītu *RSA* modelēšanas rīku.

Demonstrācijas piemēra gaitā, EMF modelis tiek pārveidots MOLA modeli. Tad tiek palaista MOLA transformācijas programma. Tālāk, izmantojot darbā aprakstītu *Eclipse* ietvara spraudni *MOLA* modelis tiek transformēts uz *EMF* modeli.

4.1. Ieejas EMF modeļu nodefinēšana ar RSA rīku

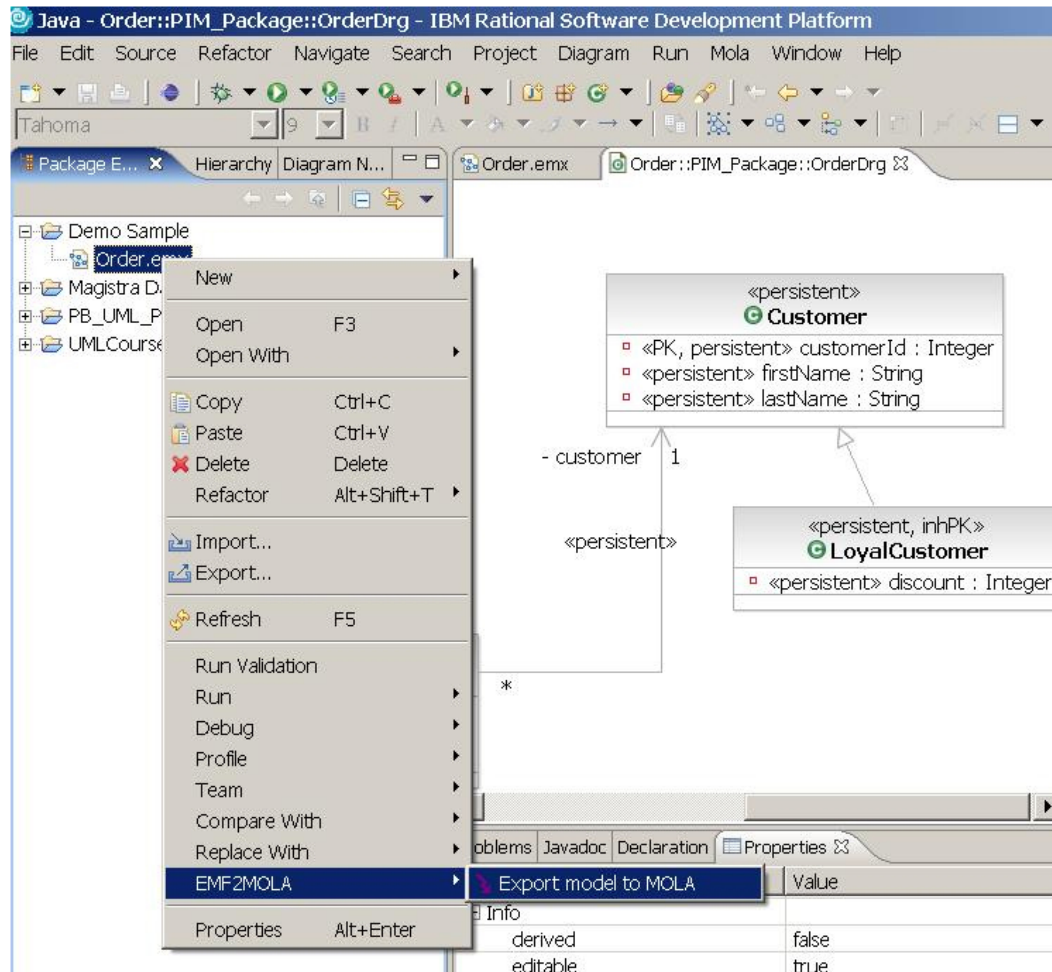
Attēlā 4.1. ir parādīts ieejas *EMF* modelis kurš ir nodefinēts izmantojot *EMF* balstītu *RSA* rīku.



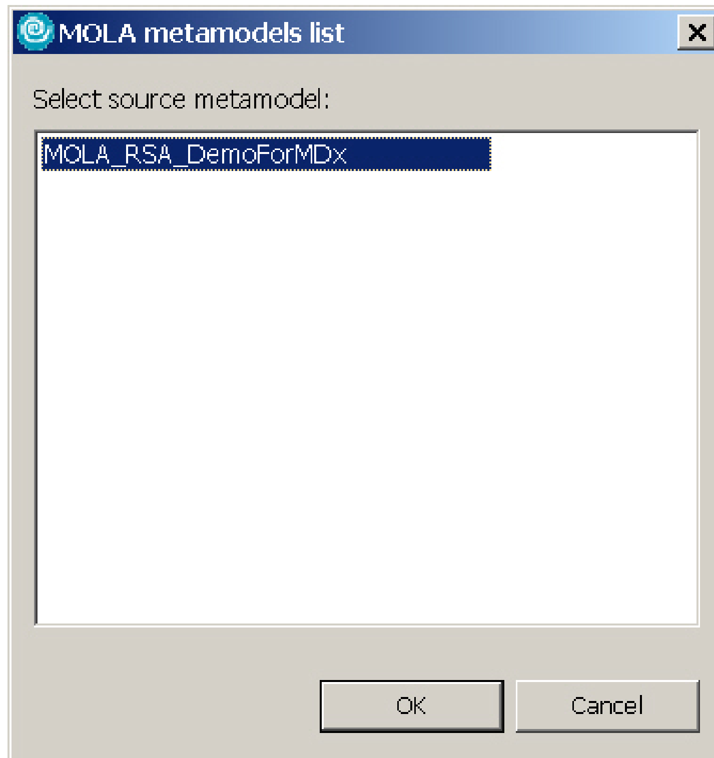
Attēls 4.1. Ieejas modeļa klašu diagramma.

4.2. Modeļu transformācija uz MOLA modeli

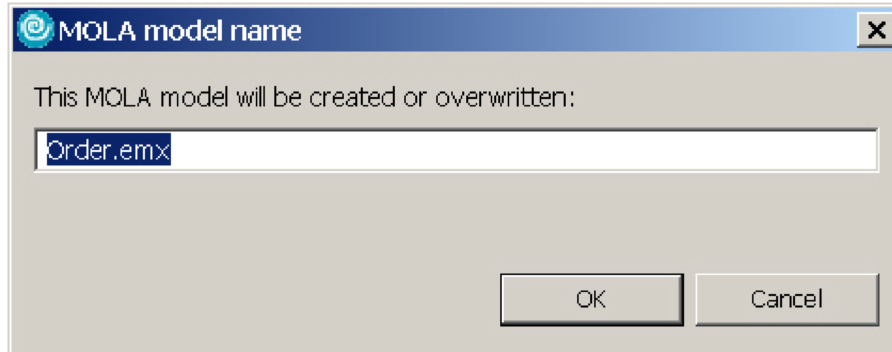
Transformācija notiek ar *EMF2MOLA Eclipse* ietvara spraudni. Šis ietvara spraudni izstrādāja maģistrants A. Haņin [17]. Attēlos 4.2. – 4.5. var redzēt darbību ar doto spraudni.



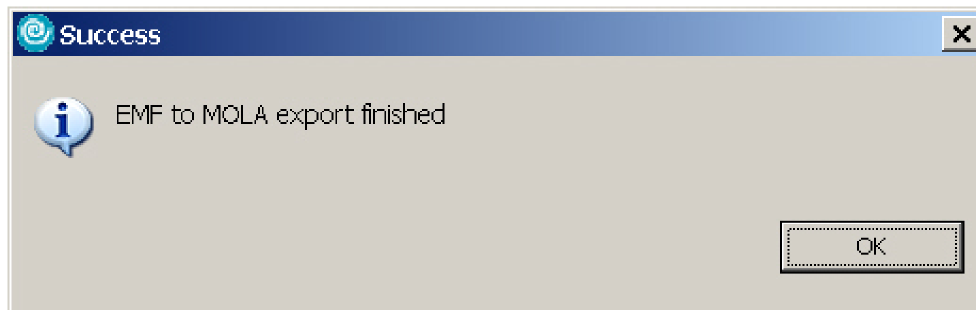
Attēls 4.2. *EMF2MOLA Eclipse* ietvara spraudņa izvēlne.



Attēls 4.3. *MOLA metamodels list* skats.



Attēls 4.4. *MOLA model name* skats.



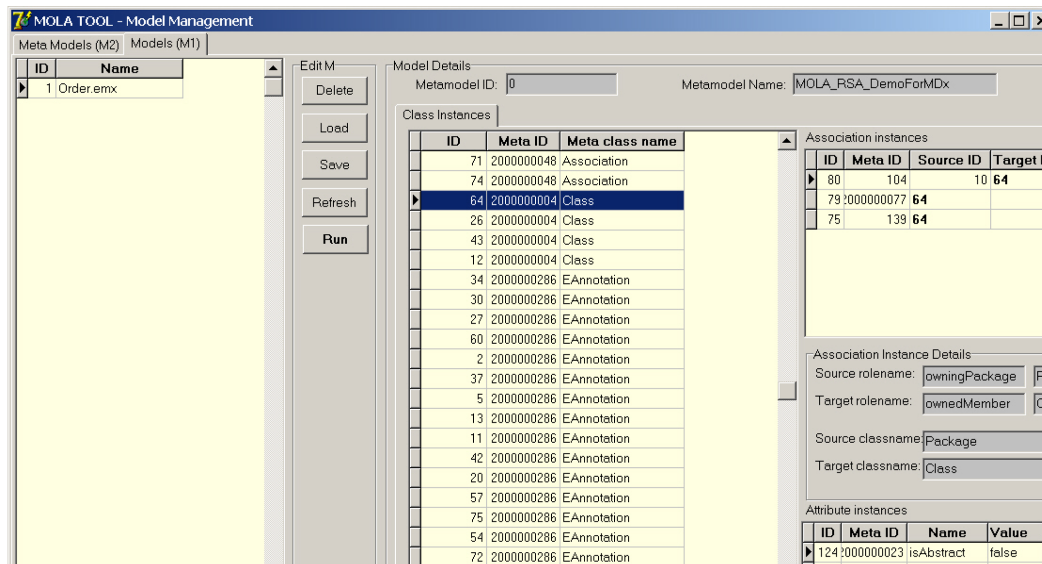
Attēls 4.5. *EMF to MOLA export finished* dialogs.

4.3. Modeļu transformācija ar MOLA rīku

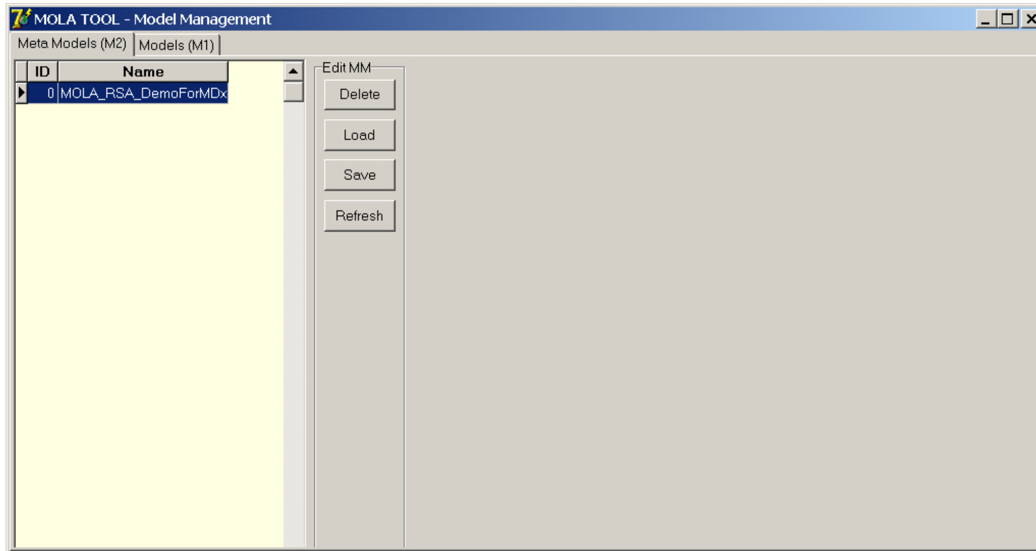
Modeļu transformācija var izpildīt ar *MOLA TOOL* rīku [14]. Lai izpildītu transformāciju ir nepieciešams nospiest *Run* pogu.

Attēlā 4.6. var redzēt apstrādāto *MOLA* modeli. Attēlā 4.7. var redzēt *MOLA* meta modeļi kas ir līdzvērtīga *EMF UML2* meta modelim.

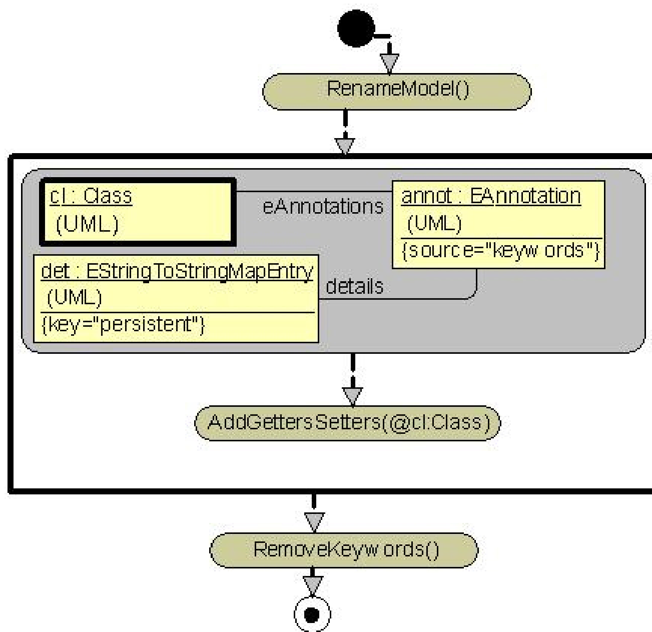
Attēlos 4.8. – 4.11. tiek paradīta *MOLA* transformācijas programma kas ir izmantota demonstrācijas piemērā. Dotas transformācijas programmas rezultātā, katram atribūtam ar *persistent* atslēgvārdu tiek uzģenerēts `get` un `set` metodes, toties pats *persistent* atslēgvārds tika nodzēst no modeļi.



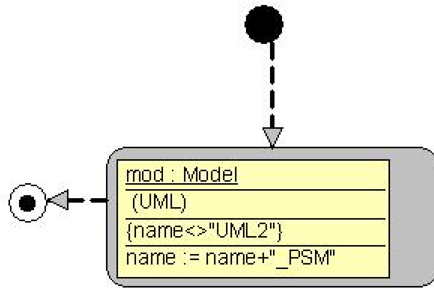
Attēls 4.6. *MOLA TOOL - Models (M1)* skats.



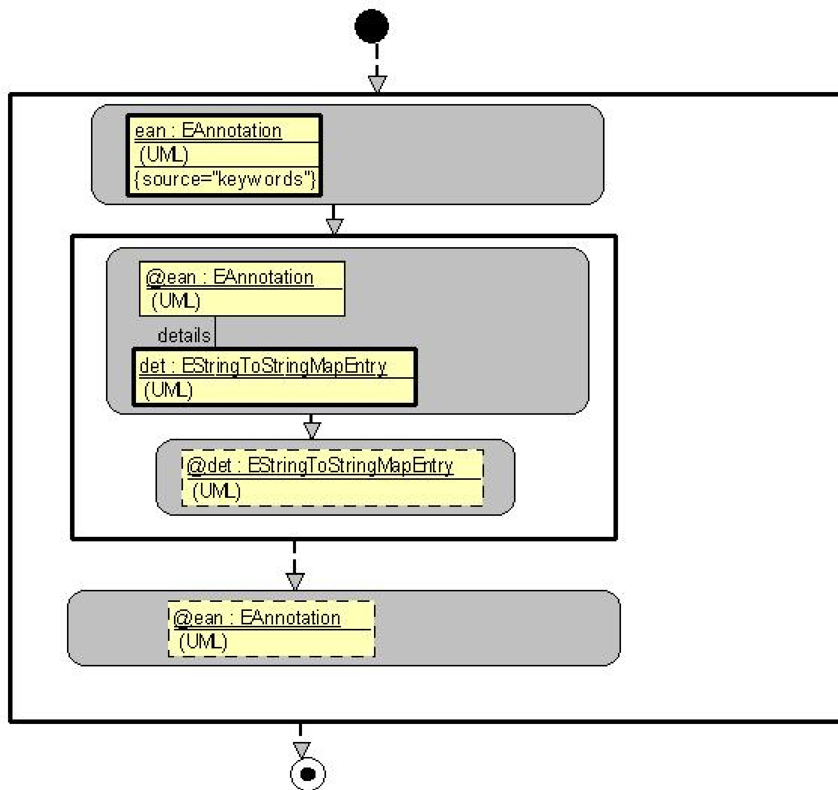
Attēls 4.7. *MOLA TOOL - Meta Model (M2)* skats.



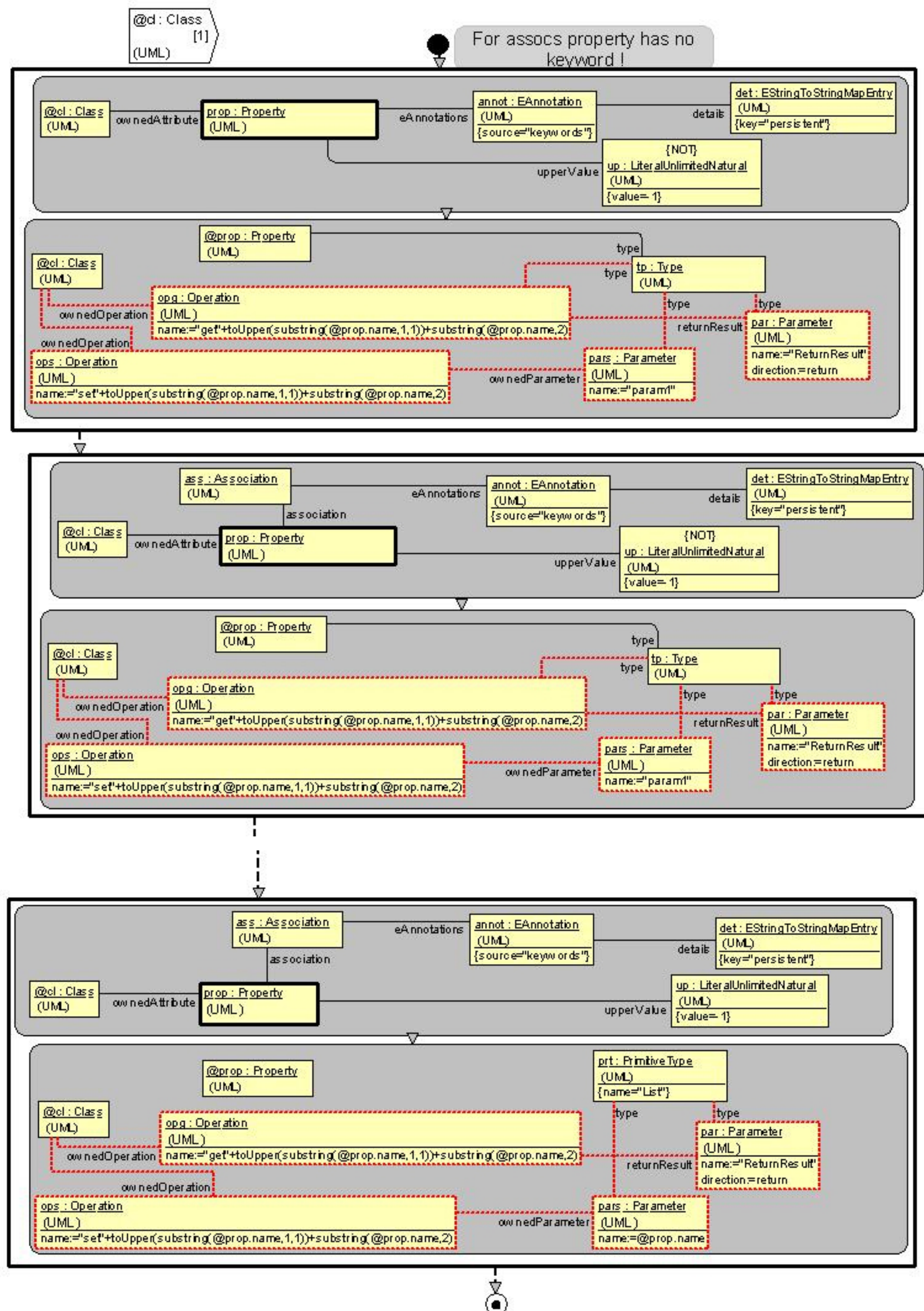
Attēls 4.8. MOLA programma *RSA_PIM_to_PSM*



Attēls 4.9. MOLA apakšprogramma *RenameModel*



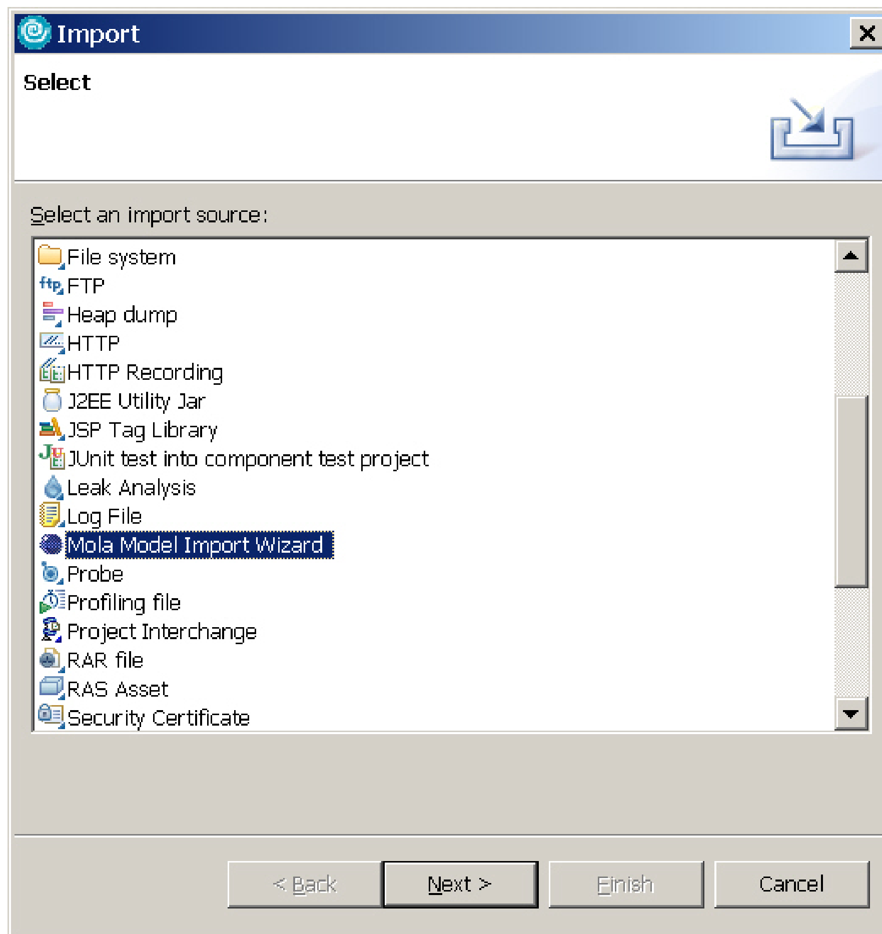
Attēls 4.10. MOLA apakšprogramma *RemoveKeywords*



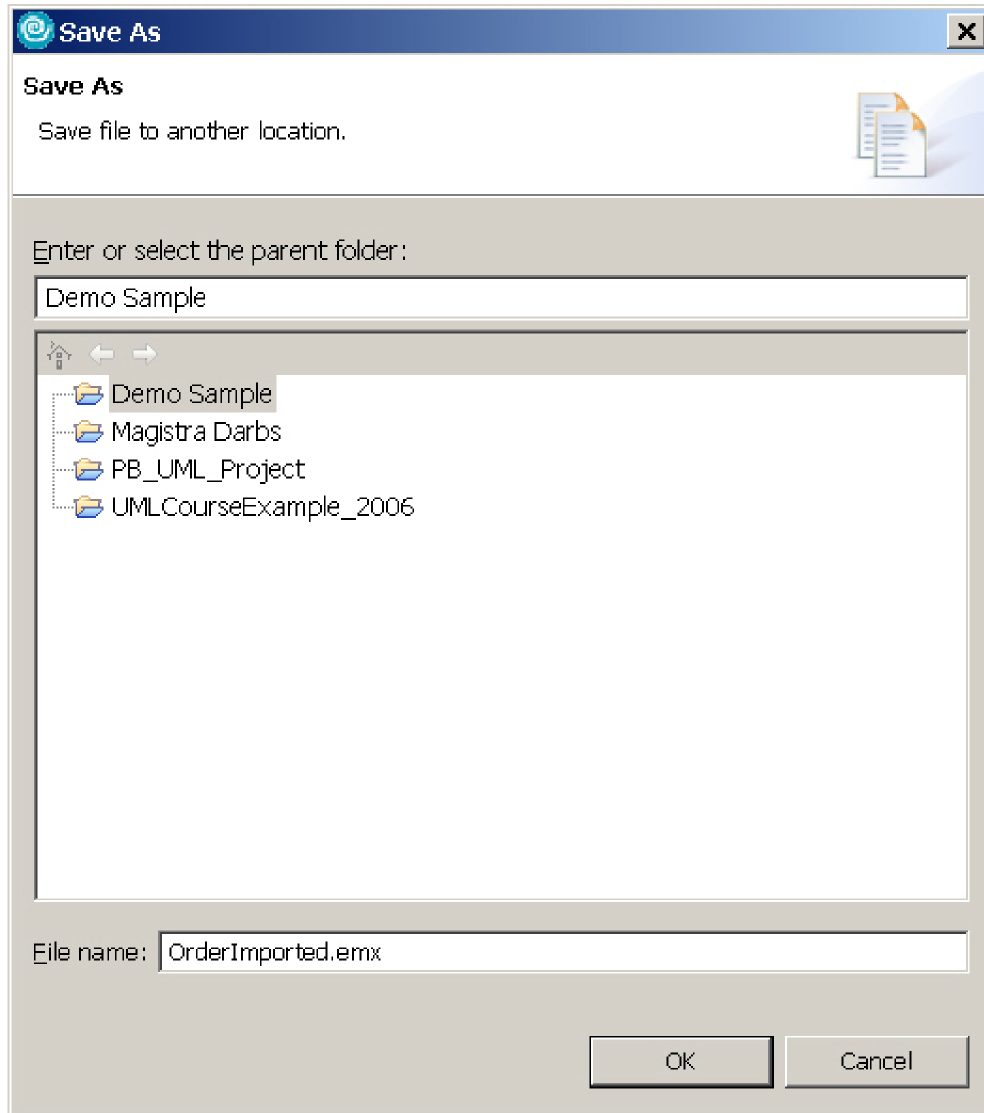
Attēls 4.11. MOLA apakšprogramma *AddGettersSetters*.

4.4. MOLA modeļa transformācija atpakaļ uz EMF modeli

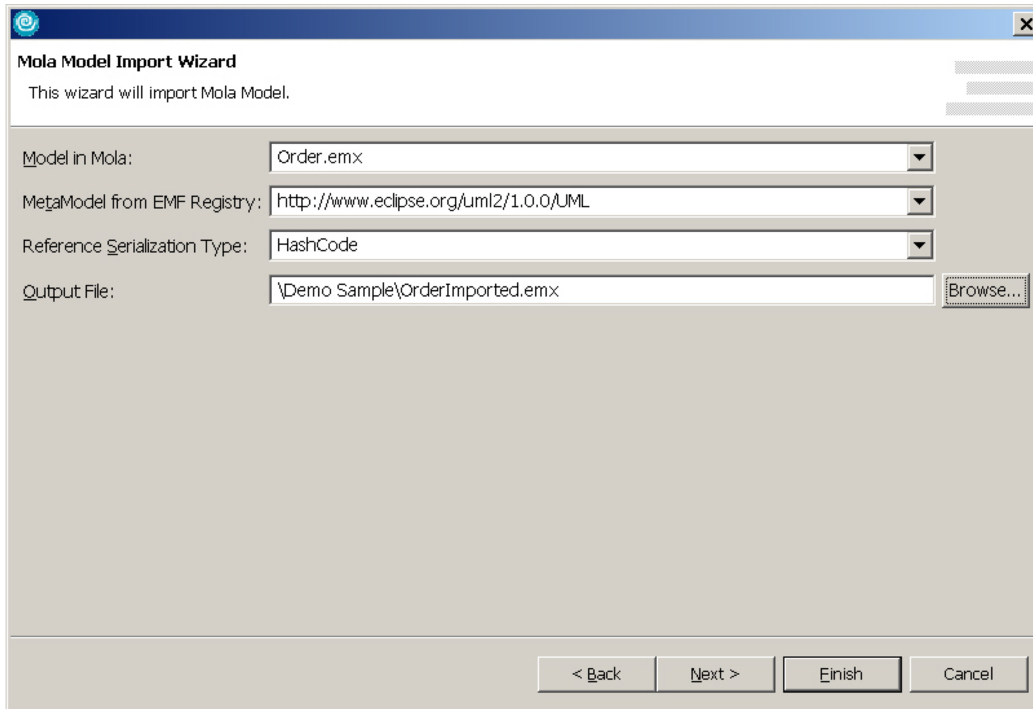
Tālāk, izmantojot darbā aprakstītu *Eclipse* ietvara spraudni, *MOLA* modelis tiek transformēts uz *EMF* modeli. Attēlos 4.12. – 4.15. var redzēt darbību ar doto spraudni.



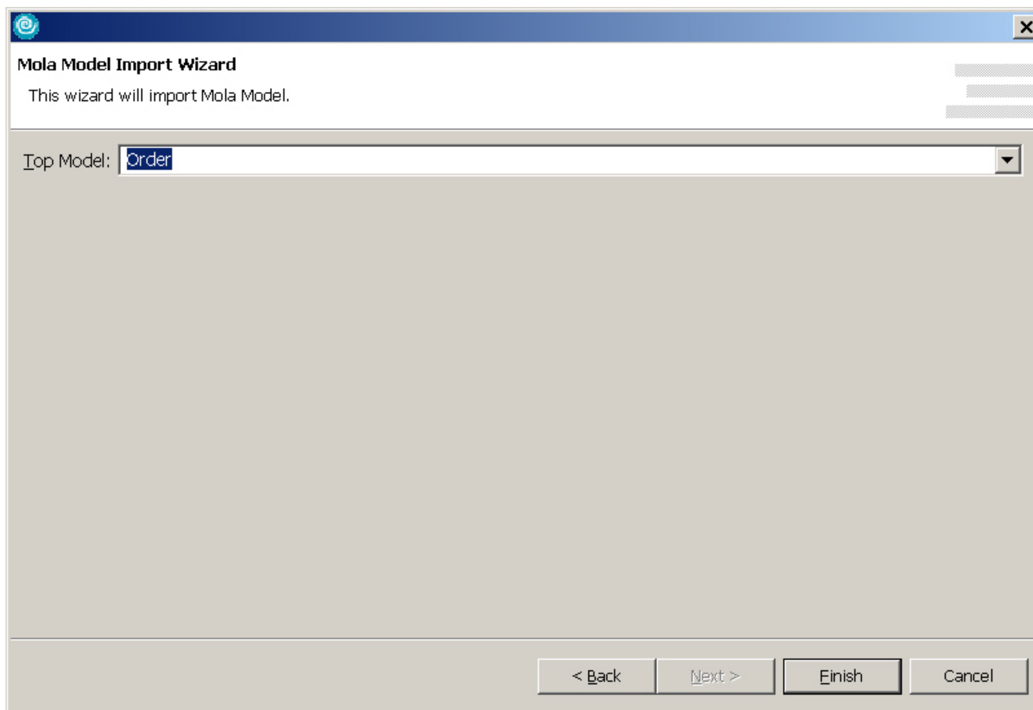
Attēls 4.12. *Import Wizard* izvēlne.



Attēls 4.13. Izejas datnes izvēles logs.



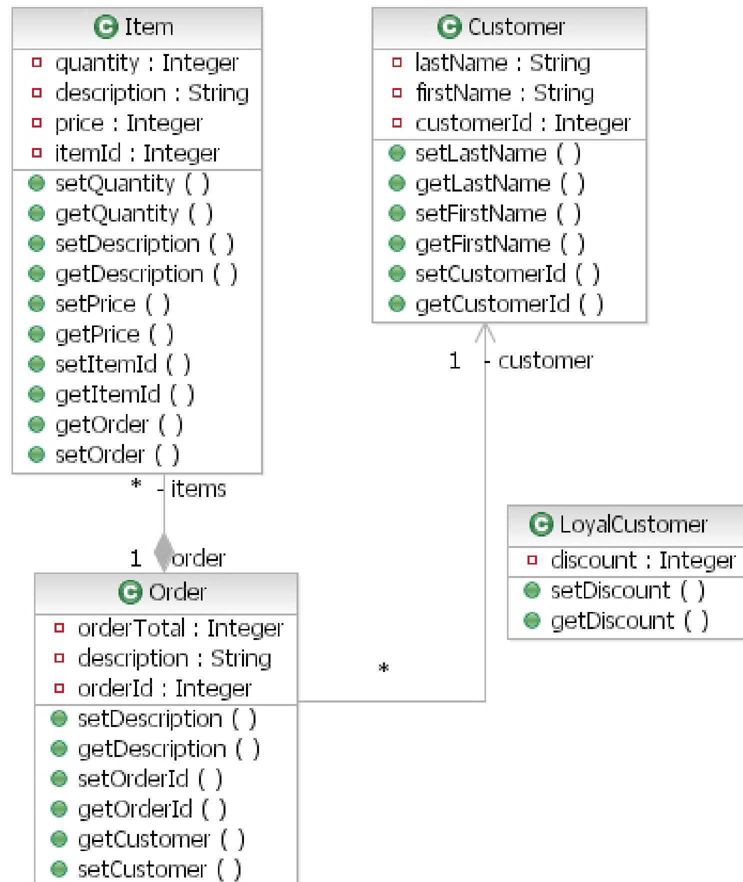
Attēls 4.14. *Mola Model Import Wizard* logs.



Attēls 4.15. *Mola Model Import Wizard* logs – galveno modeļu izvēle.

4.5. Izejas modeļa klašu diagramma

Attēlā 4.16 tik paradīta izejas klašu diagramma. Var redzēt ka *MOLA* modeļu transformācijas rezultātā katram atribūtam ar *persistent* atslēgvārdu tika uzģenerēts `get` un `set` metodes, turklāt pats *persistent* atslēgvārds tika nodzēst no modeļi.



Attēls 4.16. Izejas modeļa klašu diagramma.

Nobeigums

Darbā ir izpētīta atbilstība starp metamodelēšanas līdzekļiem *Eclipse EMF* un *OMG EMOF*, kā arī *EMOF* modifikāciju, kura tiek izmantota valodā *MOLA*. Uz šīs izpētes pamata ir izstrādāti algoritmi, kā modeļu datus pārveidot atbilstoši metamodeļiem.

Darba autors ir izstrādājis *Eclipse* ietvara spraudni (*Eclipse framework plugin*), kas realizē dotos transformācijas algoritmus. Eclipse ietvara spraudnim ir izveidots intuitīva lietotāja saskarne, ar kuras palīdzību var ērti izvēlēties transformējamo (ieejas) *MOLA* modeli, kā arī var ērti nodefinēt *EMF* modeļa saglabāšanas parametrus.

Ar demonstrācijas piemēru tika pārbaudīts, ka ar šo jauno rīku ir iespējama *MOLA* un *RSA* modelēšanas rīku sadarbspēja. Potenciāli, tas varētu būt jebkurš cits *EMF* balstīts rīks.

Domājot uz nākotnes versiju, tad būtu labi atdalīt *Hibernate jarus* atsevišķā *Eclipse* ietvara spraudnī (vislabāk ja šeit būtu izmantots jau kāds gatavs plaši lietojams *Eclipse* ietvara spraudnis) .

Būtu labi, ja pastāvētu iespēja definēt *MOLA* datubāzes savienojuma parametrus modeļa importēšanas laikā. Pagaidām tos ir nepieciešams mainīt *hibernate.cfg.xml* datnē un pēc tam atsāknēt (*restart*) *Eclipse* Platformu.

Arī būtu labi, ja būtu parādīts transformācijas rezultāts. Cik klases, atribūtus, un asociācijas tikušas sekmīgi transformētas un kuriem nav izdevies atrast ekvivalentus dotajā izejas *EMF* meta modelī.

Arī ir nepieciešams apdomāt tālāko *MOLA* rīka integrāciju *Eclipse* Platformā. Piemēram, piedāvāt līdzekļus, lai definētu un darbinātu transformācijas procesu.

Literatūras saraksts

[1] A.Kleppe, J.Warmer, W.Bast, *MDA Explained: The Model Driven Architecture. Practise and Promise*. Addison Wesley, 2004.

[2] David S. Frankel, *The Expanding Significance of the Eclipse Modeling Framework (EMF). MDA Journal,* March 2004.

[3] Eclipse, *Eclipse Tools - EMF*, http://www.eclipse.org/emf/, [tiešsaiste, atsauce 2006.gada 5.jūnijs]

[4] Eclipse, *Eclipse Projects – Eclipse Modeling Project*, http://www.eclipse.org/modeling/, [tiešsaiste, atsauce 2006.gada 5.jūnijs]

[5] Eclipse, *Eclipse Projects – Model Driven Development integration*, http://www.eclipse.org/mddi/, [tiešsaiste, atsauce 2006.gada 5.jūnijs]

[6] D. Keith, A. Gerber, K. Raymond. *Eclipse Modeling Framework (EMF) import/export from MOF/JMI*. Pagamento Project, DSTC Status Report: 3 May 2003.

[7] Stephen J. Mellor, K. Scott, A. Uhl, D. Weise, *MDA Distilled: Principles of Model-Driven Architecture*. Addison Wesley, 2004.

[8] IBM Corp, *Eclipse Platform Technical Overview,*. 2006.

[9] Eclipse, *Eclipse Projects – Model Driven Development Integration*, http://www.eclipse.org/proposals/eclipse-mddi/, [tiešsaiste, atsauce 2006.gada 7.jūnijs]

[10] V. Bacvanski, P. Graff, *Mastering Eclipse Modeling Framework*. InferData, 2005.

[11] Eclipse, *Eclipse Tools – EMF, EMF v1.0 User`Guide*, http://dev.eclipse.org/viewcvs/indextools.cgi/%7Echeckout%7E/emf-home/docs/UG/EMF_v1.0_Users_Guide.html , [tiešsaiste, atsauce 2006.gada 5.jūnijs]

[12] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose*, Eclipse Modeling Framework: A Developer`s Guide*. Addison Wesley, 2003.

[13] C. Griffin, *Indtroduction To Eclipse and the Eclipse Modeling Framework*. InferData, 2005.

[14] A. Kalnins, E. Celms, A. Sostaks, *Model Transformation Approach Based on Mola*, 2003.

[15] MOLA Project, *Reference Manual*, http://mola.mii.lu.lv/, [tiešsaiste, atsauce 2006.gada 5.jūnijs]

[16] OMG Available Specification, *Meta Object Facilities (MOF) 2.0 Core Specification*, ptc/04-10-15.

[17] A. Haņins, *Metodes darbam ar EMF modeļiem, to integrācija valodā MOLA*, LU, Maģistra Darbs, 2006.

[18] A. Sostaks, *MOLA interpretatora realizācija*. LU, Maģistra Darbs, 2005.

Apliecinājums

Ar šo es apliecinu, ka šodien iesniegto maģistra darbu es esmu veicis pašrocīgi un esmu izmantojis tikai tajā norādītos palīglīdzekļus.

Rīgā, 2006. gada 7. jūnijs Paraksts: /P.Bubens/

Reģistrācijas lapa

Maģistra darbs izstrādāts
LU Datorikas nodaļā

Autors:
Fizikas un matemātikas
fakultātes students Pāvels Bubens . . . .
St. apl. Nr. DatZ000154

Darba vadītājs:
Edgars Celms . . . .

LU Matemātikas un informātikas institūts

Recenzents:
Oskars Vilītis . . . .

LU Matemātikas un informātikas institūts

Darbs iesniegts maģistrantūras sekretariātā 2006. g. . . . . . jūnijā.

Pieņēma sekretāre . . . .

Aizstāvēts datorzinātņu maģistra pārbaudījumu komisijas sēdē

2006.g. . . . . .ar atzīmi. . . . .

Protokols Nr. _________

Maģistra pārbaudījumu
komisijas sekretārs . . . .