Metamodeling with Metamodels

Using

UML/MOF including OCL

Introducing Metamodels (Wikipedia)

- A metamodel is a model of a model
- An instantiation of metamodel gives a model
- Metamodeling is the process of generating such metamodels
- Metamodeling is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems
- Metamodeling applies the notions of *meta*- and *modeling* in software engineering and systems engineering
- Metamodels are of many types and have diverse applications

Contributions of Object Management Group (OMG)

- In software engineering, the use of models is an alternative to more common code-based development techniques
- A model always conforms to a unique metamodel
- One of the currently most active branches of Model Driven Engineering is the approach named Model-Driven Architecture (MDA) proposed by OMG (Object Management Group)
- MDA utilizes the language Meta Object Facility (MOF) to write metamodels
- MOF roughly corresponds to the class diagram part of UML including OCL constraints
- Typical metamodels proposed by OMG are UML, OCL, SysML (Systems Modeling Language), or CWM (Common Warehouse Metamodel)
- Such languages can be defined as MOF metamodels, i.e., models formulated with MOF

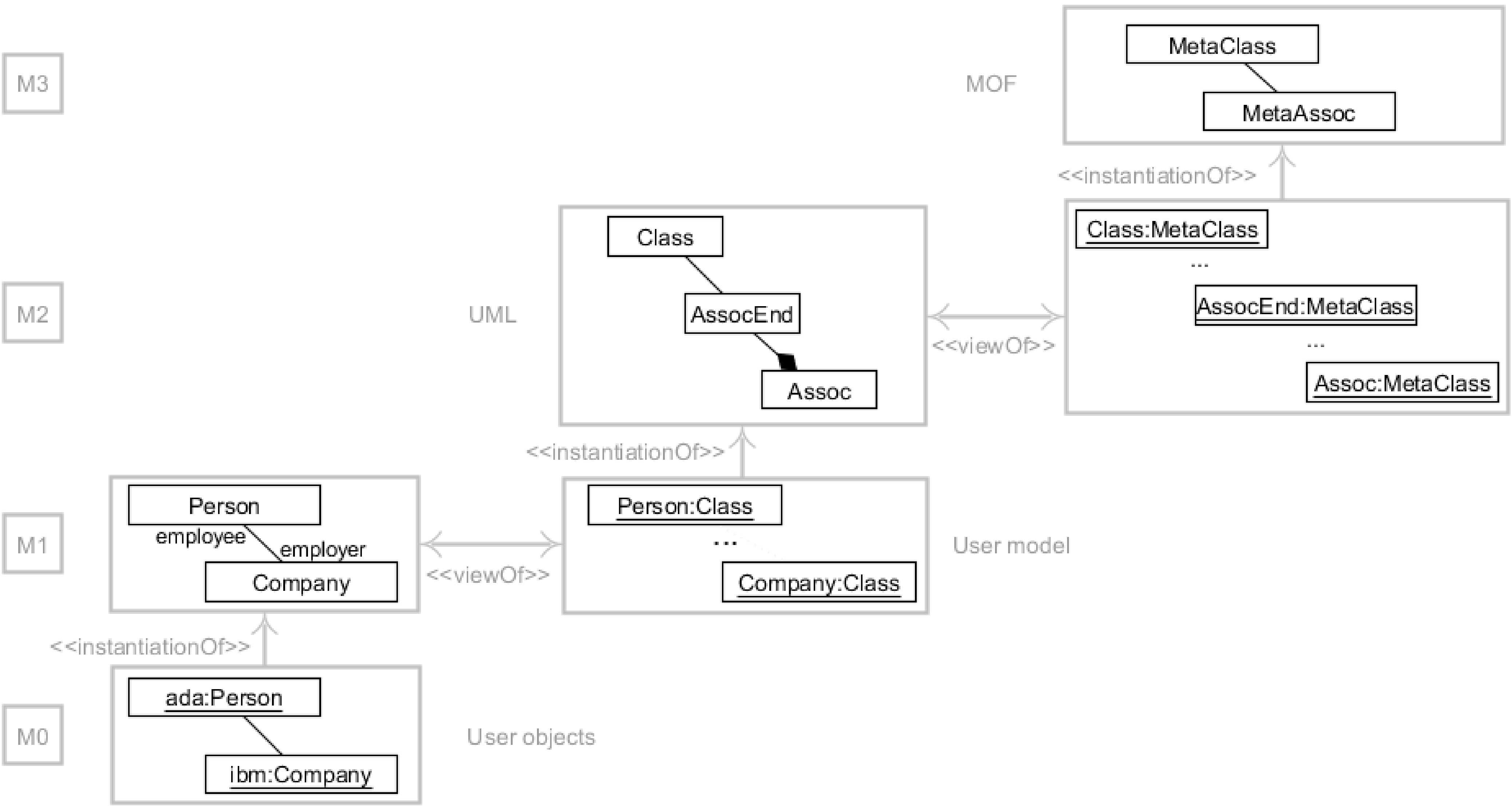
Four Level OMG Model-Driven Architecture (MDA)

Layer	Description	Example
meta-metamodel M3 / MOF	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	*MetaClass, MetaAttribute, MetaOperation*
metamodel M2 / UML	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
model M1 / User model	An instance of a metamodel. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*
user objects (user data) M0	An instance of a model. Defines a specific information domain.	*<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>*

Model-Driven Architecture (MDA) – Analogy Programming

Layer	Analogy Programming Languages	**Example**
meta-metamodel M3 / MOF	EBNF notation for context-free grammars (Extended Backus-Naur Form)	*MetaClass, MetaAttribute, MetaOperation*
metamodel M2 / UML	Syntax definition of one programming language with context-free grammar in EBNF notation, e.g. PASCAL or JAVA	*Class, Attribute, Operation, Component*
model M1 / User model	One specific JAVA program J	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*
user objects (user data) M0	One execution of JAVA program J	*<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>*

Four Level OMG Model-Driven Architecture (MDA)



Class diagram understood as MM (MetaModel) object diagram

- Usual work with class diagrams

  - define first the class diagram

  - develop then various object diagrams

  - tune the class diagram to meet developer needs

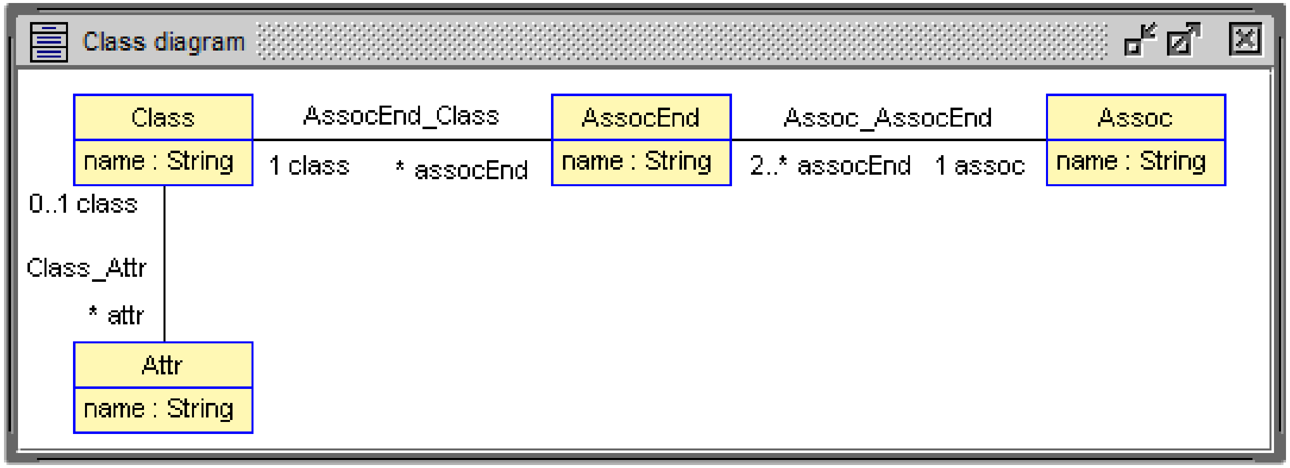
- Approach within metamodeling

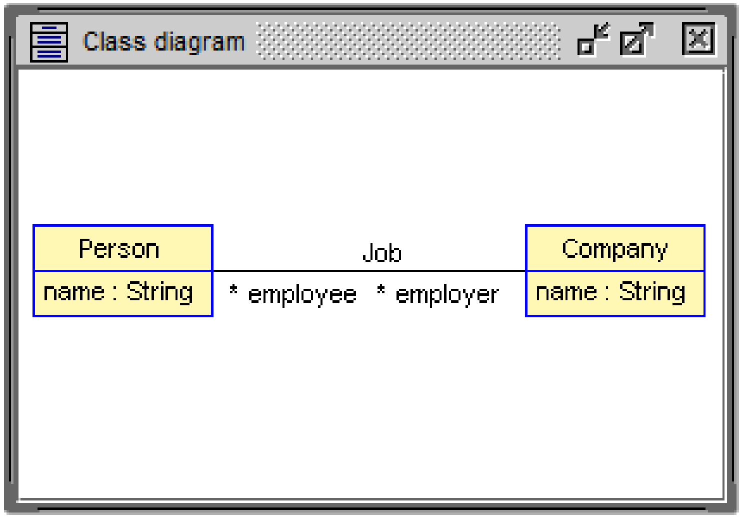
  consider the concepts appearing in a class diagram (class, attribute, association, …)

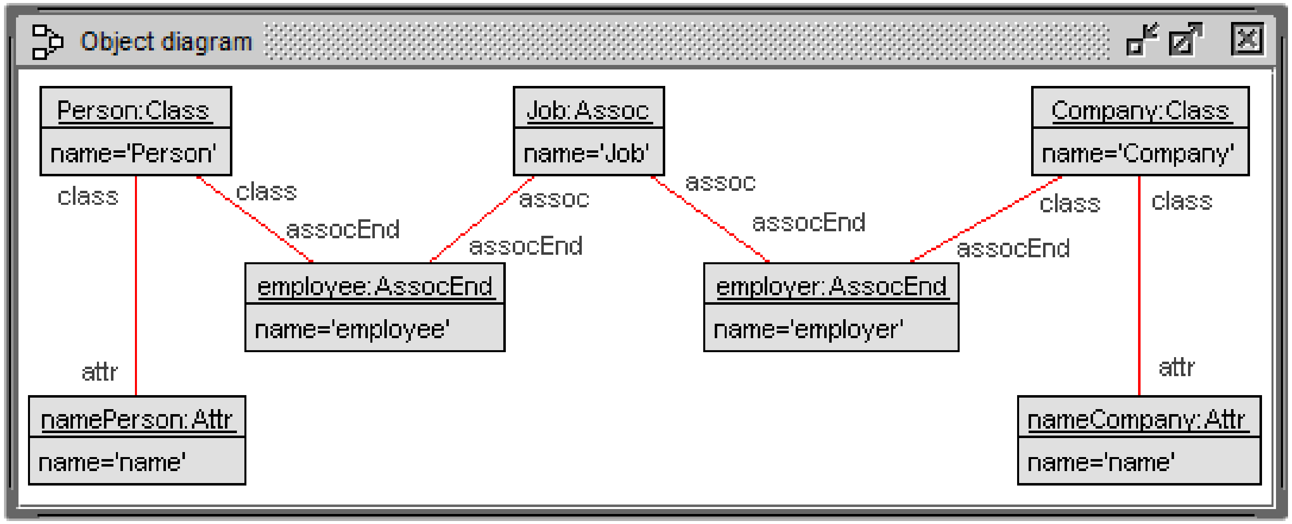
  describe these concepts and their relationships again with a class diagram

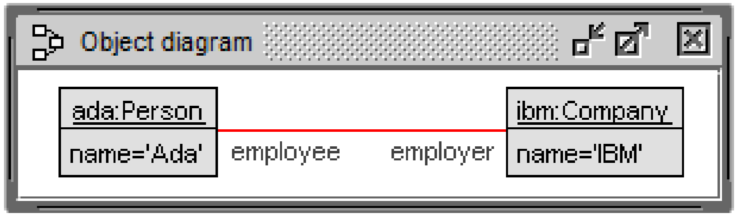
  if class diagrams are a powerful mechanism, why should one not describe class diagrams with class diagrams

Class diagram understood as MM (MetaModel) object diagram





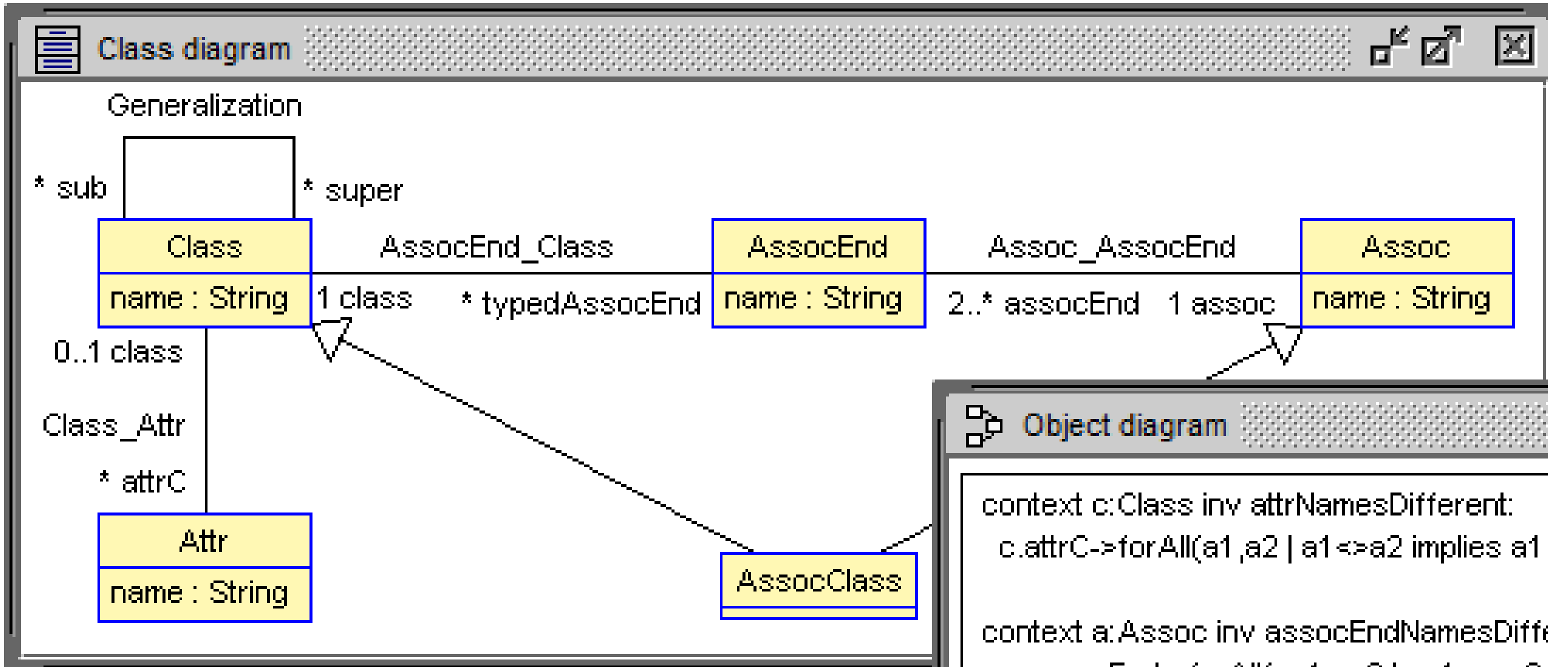




MM Extension: Generalization, Association class

- First MM (4 classes) described classes, attributes, associations, and associations ends
- Consider now also further concepts: generalization between classes and association classes
- Apply invariants in order to achieve only valid class diagrams
- Attribute names within a class are unique
- Attribute name and association end names are different
- Generalization hierarchies are acyclic
- Optional: Exclude multiple inheritance
- Overall result: CD plus the stated invariants determine a set of valid objects diagrams; this set of object diagrams builds the defined (modeling) language
- THUS: Metamodeling is an approach for language development

MM Extension: Generalization, Association class



Object diagram

```
context c:Class inv attrNamesDifferent:
  c.attrC->forAll(a1,a2 | a1<>a2 implies a1.name<>a2.name)

context a:Assoc inv assocEndNamesDifferent:
  a.assocEnd->forAll(ae1,ae2 | ae1<>ae2 implies ae1.name<>ae2.name)

context c:Class inv attrNamesDifferentFromAssocEndNames:
  c.attrC.name->intersection(c.typedAssocEnd.name)->isEmpty()

context ac:AssocClass inv oneName:
  let c:Class=ac in let a:Assoc=ac in c.name=ac.name

context c:Class inv acyclicGeneralization:
  c.super->closure(super)->excludes(c)

context c:Class inv noMultipleInheritance:
  c.super->size<=1
```

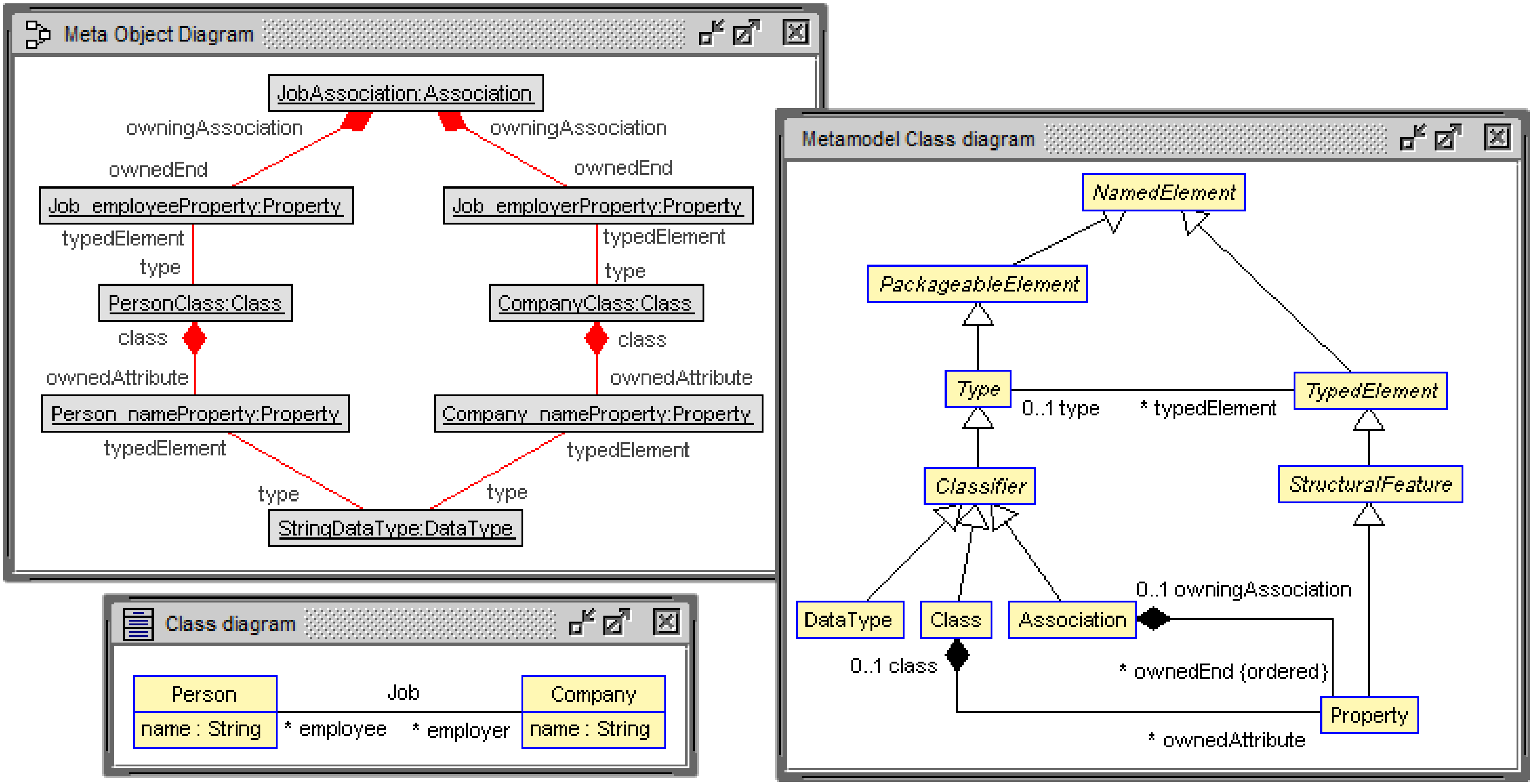
Proper UML 2.4 Metamodel (more complicated)

- UML is defined by a class diagram plus restricting OCL invariants
- This class diagram is called the 'UML Metamodel (MM)'
- UML was developed over the recent years by the OMG
- Various versions were published

  UML 1.1, UML 1.2, …, UML 2.0, …, UML 2.4, ...

  UML 2.4 is an important and well accepted version
- UML uses a different terminology (different class and association end names) than the motivating simple metamodel used above
- Attribute and AssociationEnd objects are commonly treated as Property objects; a Property object 'lives within' a class (then it is an attribute) or the Property object 'lives within' an association (then it is an association end); 'lives within' = composition / black diamond

Proper UML 2.4 Metamodel (more complicated)



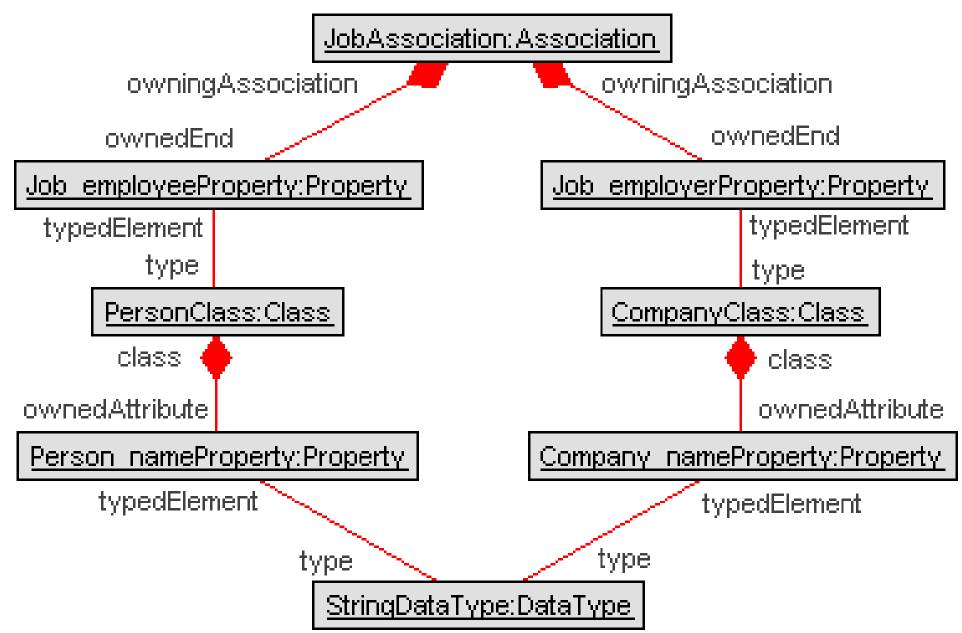
Options through representing CDs with object models

- OCL expressions can be stated on the object diagram representing the class diagram
- USE version available that incorporates UML 2.4 MM and can represent a user class diagram as a UML 2.4 MM object diagram
- Which are the association end names of a given association?
- What are all the class names together with the classes associations end names?
- What are all the class names together with the classes attribute names?
- Which properties (attributes and association ends) are typed through which classes?
- Which properties are typed through Datatypes?
- Such OCL expressions can represent generally interesting features of a class diagram, independent of the particular considered class diagram

OCL expressions for example class diagram

Object diagram

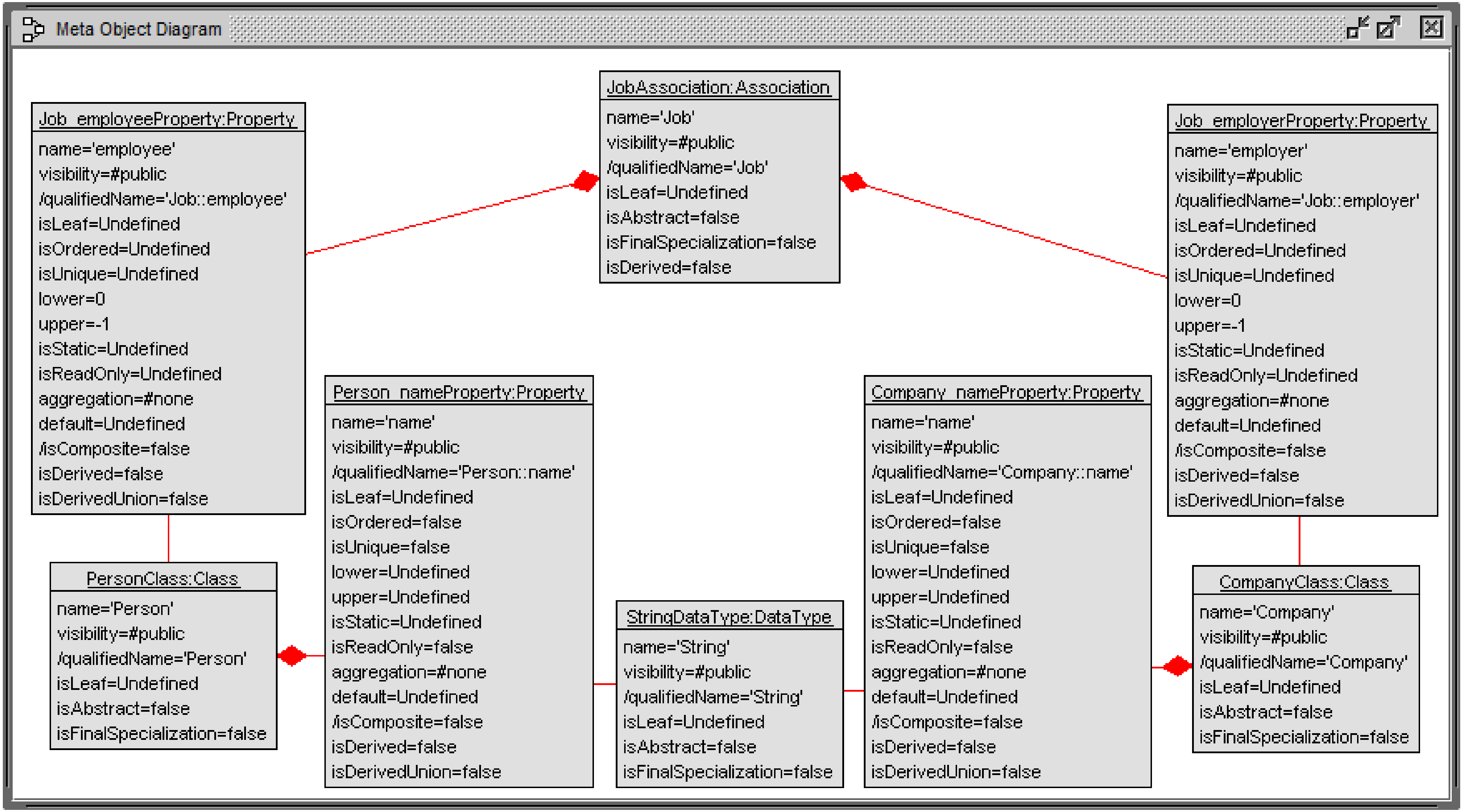
```
JobAssociation.ownedEnd
OrderedSet{Job_employeeProperty,Job_employerProperty} : OrderedSet(Property)

JobAssociation.ownedEnd.name
Sequence{'employee','employer'} : Sequence(String)

Class.allInstances->collect(c|Tuple{N:c.name,A:c.typedElement.oclAsType(Property).name})
Bag{Tuple{N='Person',A=Bag{'employee'}},Tuple{N='Company',A=Bag{'employer'}}} : Bag(Tuple(N:String,A:Bag(String)))

Class.allInstances->collect(c|Tuple{N:c.name,A:c.typedElement.oclAsType(Property).association.name})
Bag{Tuple{N='Company',A=Bag{'Job'}},Tuple{N='Person',A=Bag{'Job'}}} : Bag(Tuple(N:String,A:Bag(String)))

Class.allInstances->collect(c|Tuple{N:c.name,A:c.ownedAttribute.oclAsType(Property).name})
Bag{Tuple{N='Company',A=Sequence{'name'}},Tuple{N='Person',A=Sequence{'name'}}} : Bag(Tuple(N:String,A:Sequence(String)))

Class.allInstances.typedElement
Bag{Job_employeeProperty,Job_employerProperty} : Bag(Property)

DataType.allInstances.typedElement
Bag{Company_nameProperty,Person_nameProperty} : Bag(Property)
```



Attributes in UML 2.4 MM

- The above object diagram (for Person-Job-Company) showed only the objects and links, but not the attributes
- Some details follow ...
- Attribute name (all classes) gives name in form of a String
- Lower and upper bounds of association ends are represented by the Integer attributes 'lower' and 'upper' (for Property);
  upper value '-1' represents '*'
- Attribute 'aggregation' (for Property) distinguishes between 'association', 'aggregation' and 'composition': #none, #shared, #composite (enumeration)
- Boolean attribute 'isAbstract' (for Class) specifies whether the class is abstract or not

Attributes in UML 2.4 MM

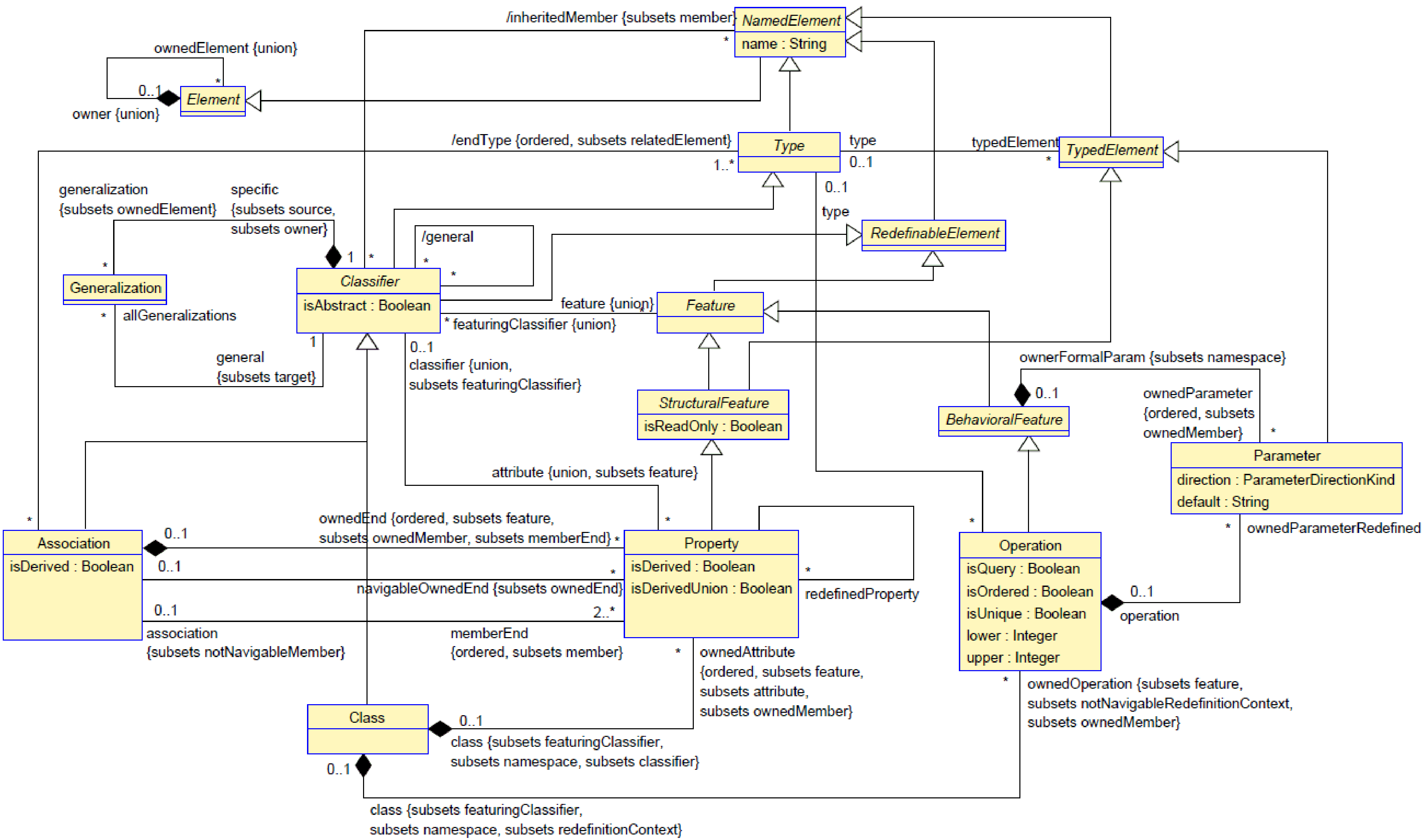


Central elements of UML 2.4 MM

- Much more classes and associations are part of the UML 2.4 MM than the ones that have been shown
- Some details follow …
- Property < StructuralFeature < TypedElement
- Class < Classifier < Type
- Association:
  Type *role* [0..1] type – TypedElement *role* [0..*] typedElement

  StringDataType:DataType<Type *role* type –
  Person_nameProperty:Property<TypedElement *role* typedElement

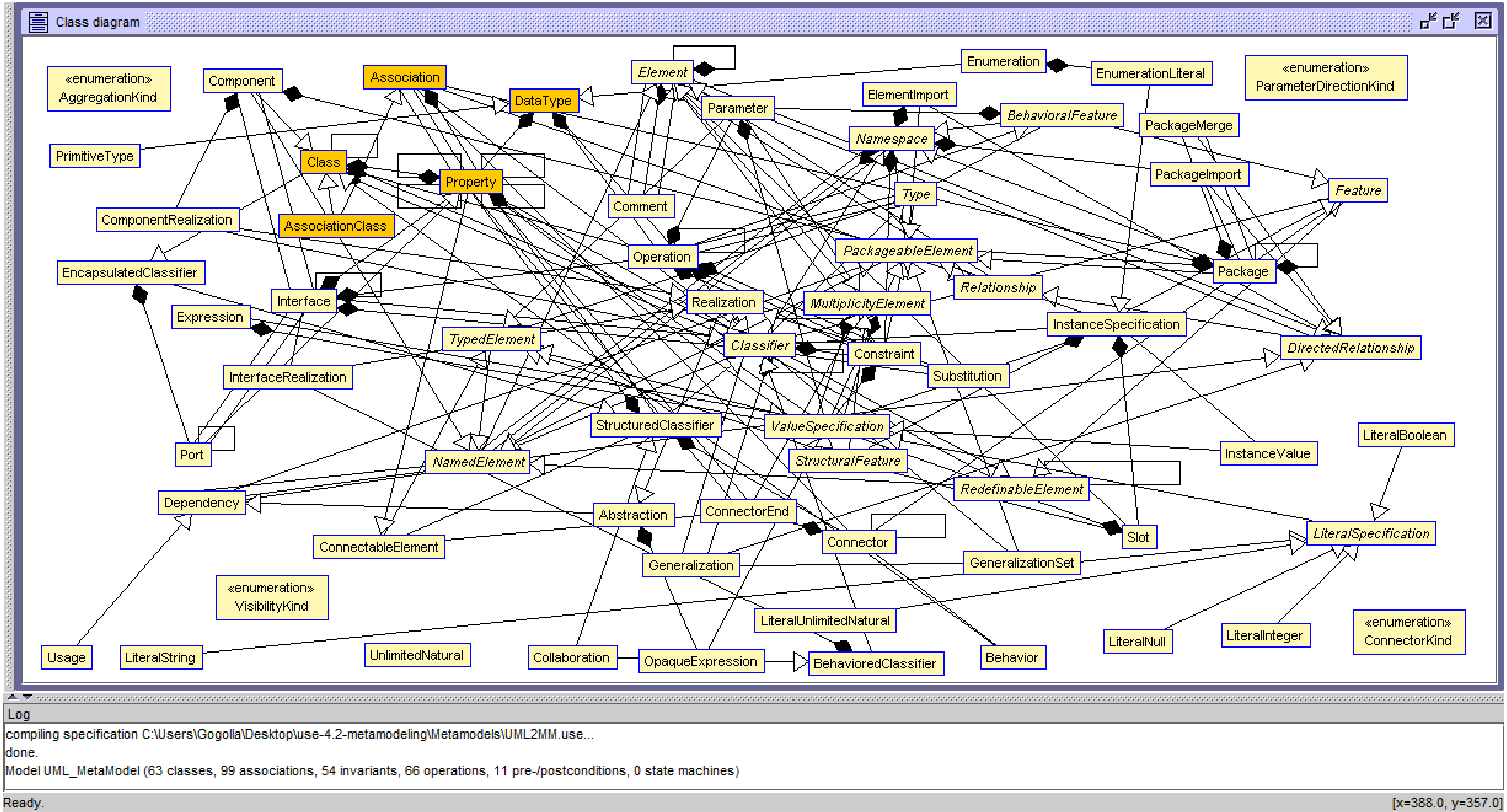
Central elements of UML 2.4 MM



UML 2.4 MM: All classes and associations

- UML 2.4 MM available as a USE model
- 63 classes
- 99 associations
- 54 invariants
- 66 operations

UML 2.4 MM: All classes and associations



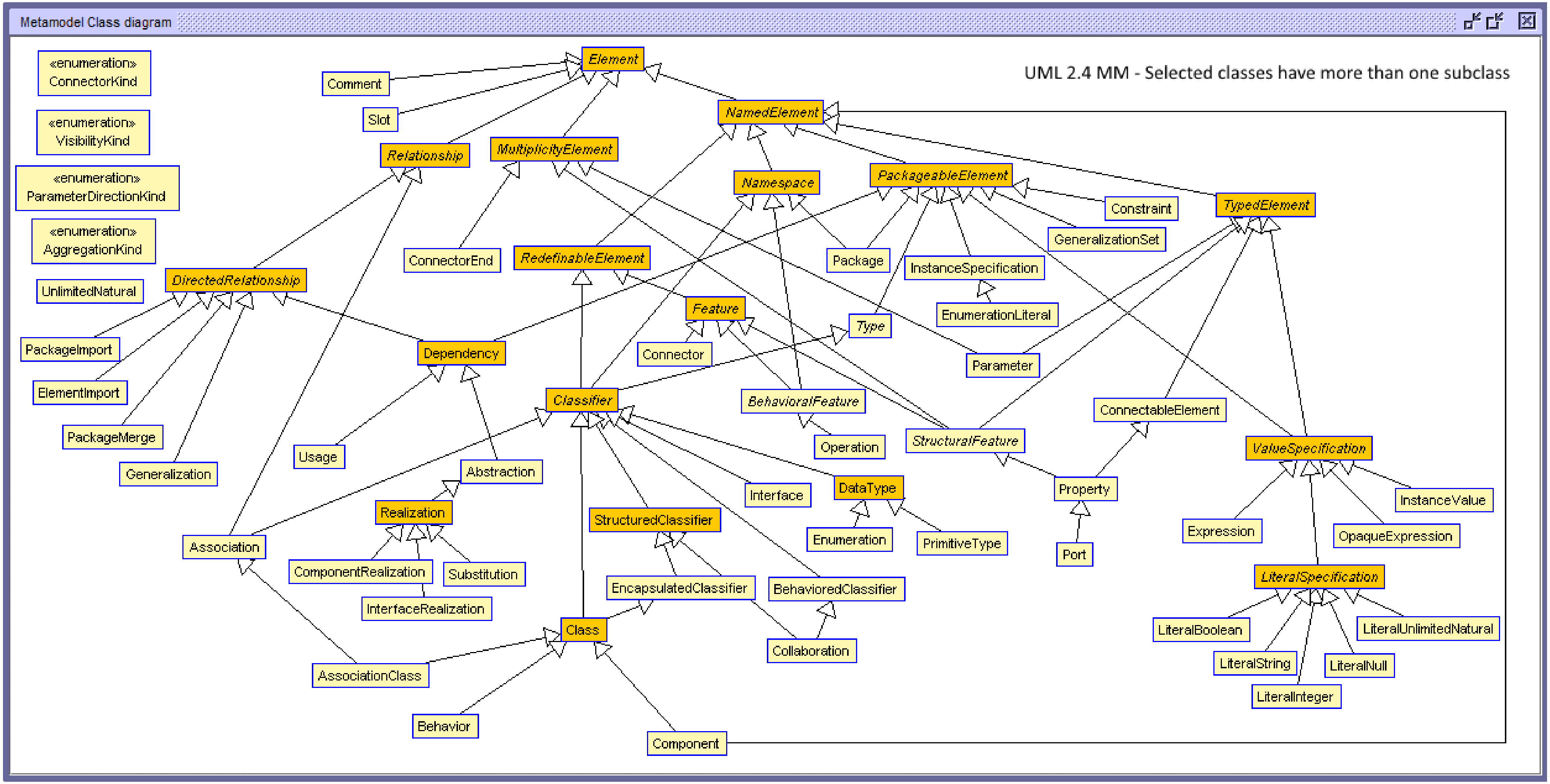
UML 2.4 MM CD complex – Building views gives overview

- Classes with more than one subclass
- Classes with more that one superclass (multiple inheritance used!)
- Classes being 'simple' specializations

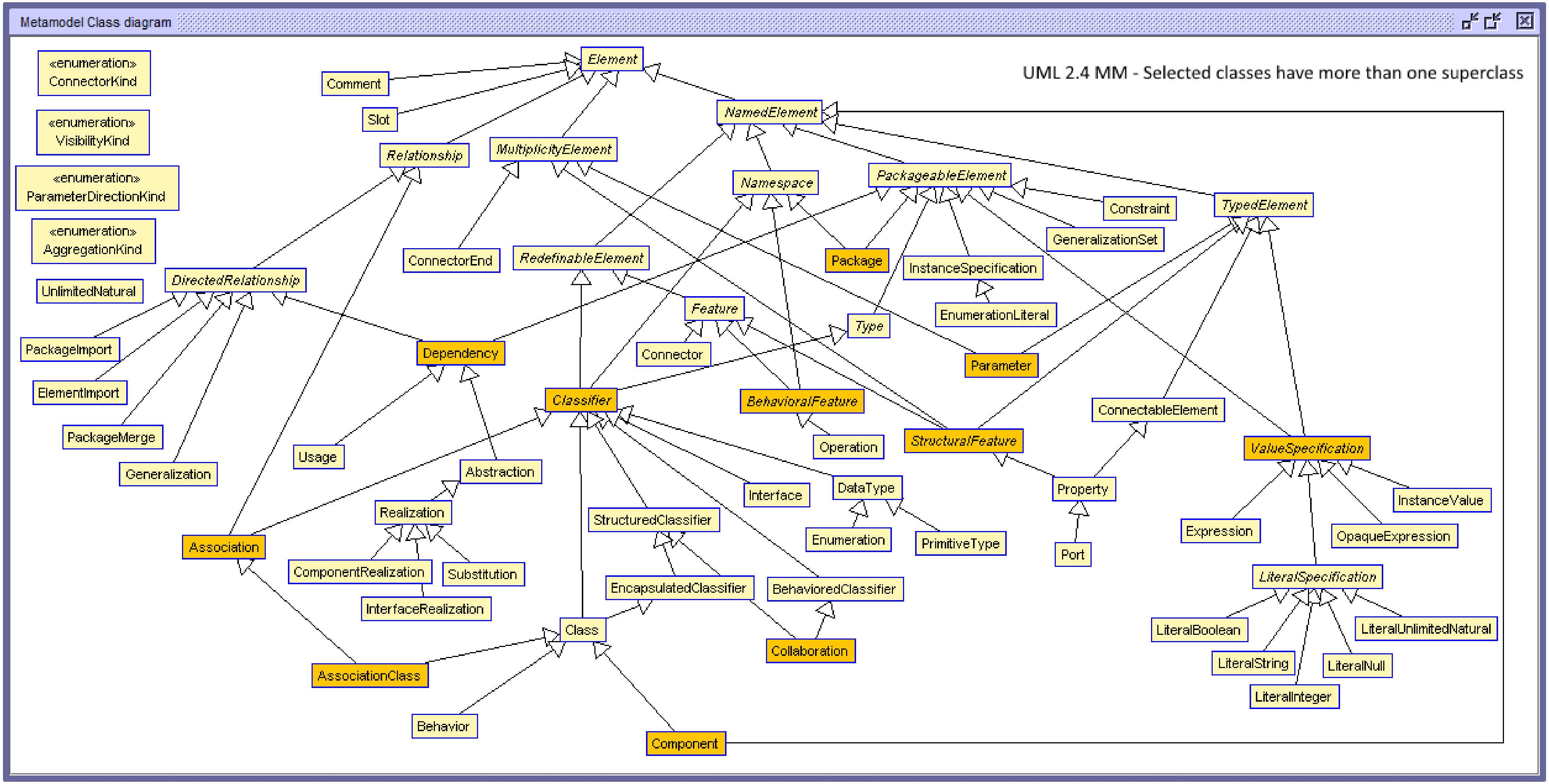
  class c with 'c.sub->isEmpty and s.super->size=1'
- Classes involved in at least 2 generalizations

- At the top of the generalization hierarchy is 'Element'
- Subclasses realize particular functionality; examples follow ...
- NamedElement (with attribute 'name')
- MultiplicityElement (with attributes 'lower' and 'upper')
- TypedElement (with association typedElement - type)

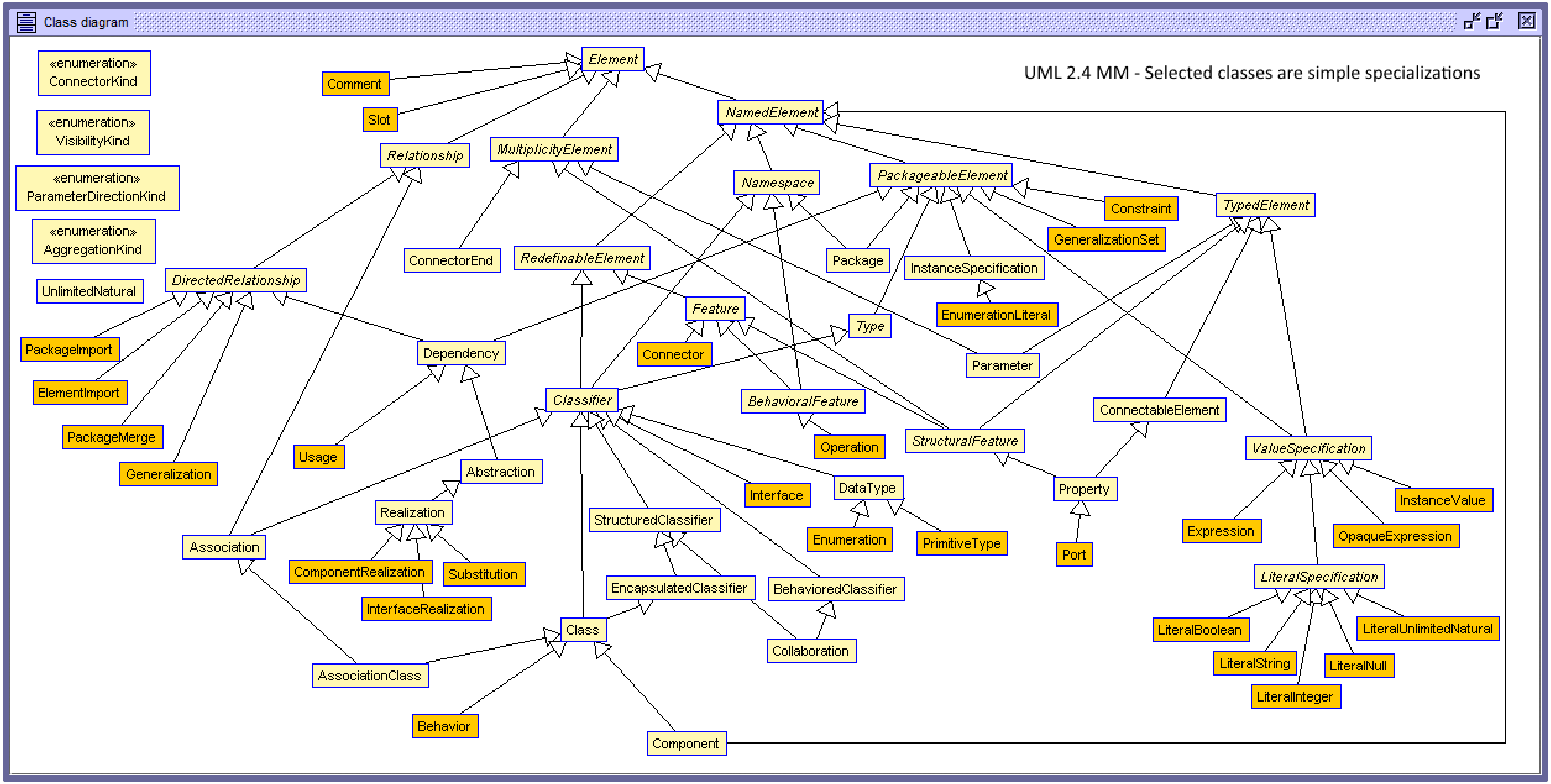
Classes with more that one subclass



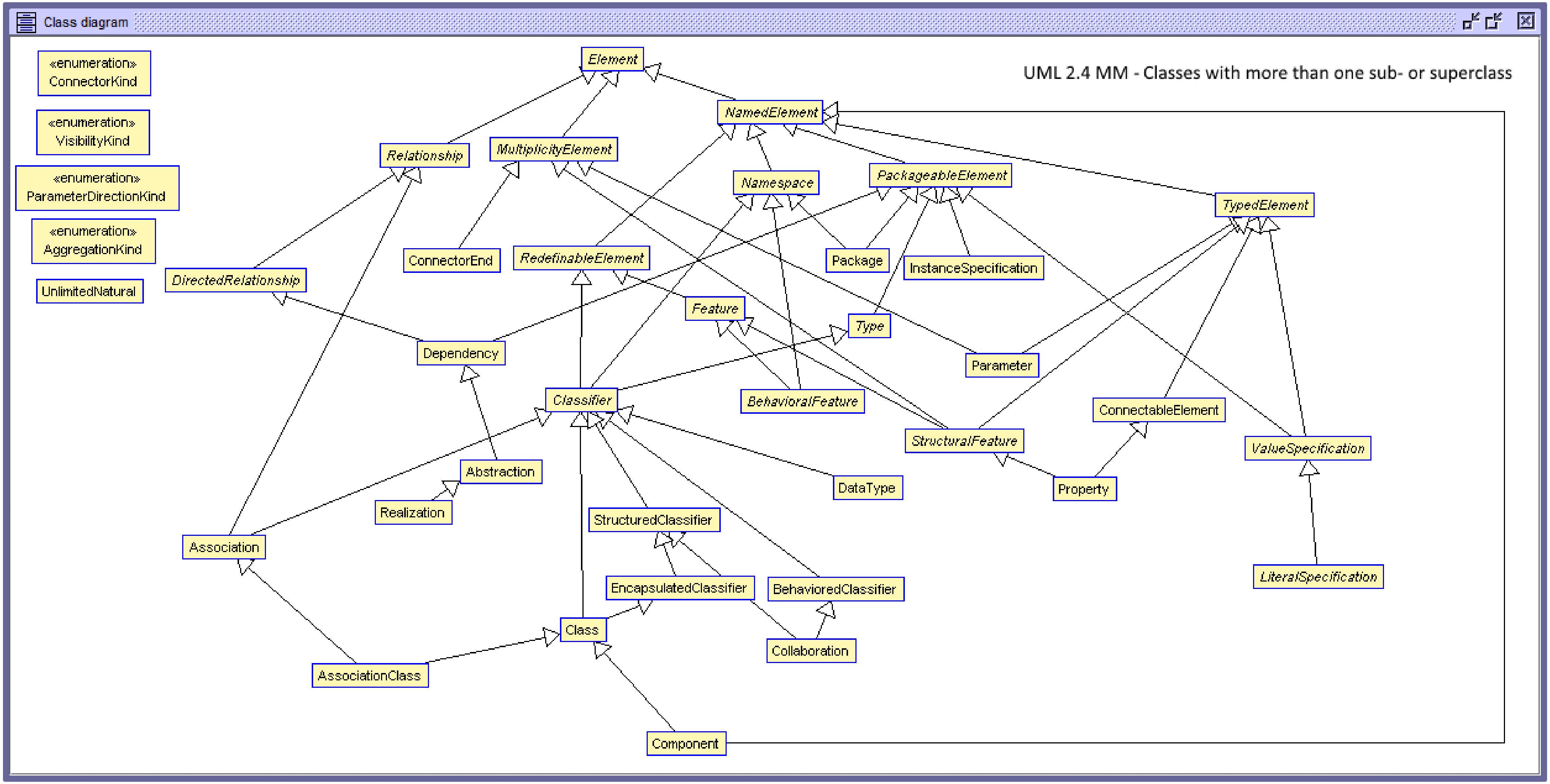
Classes with more than one superclass



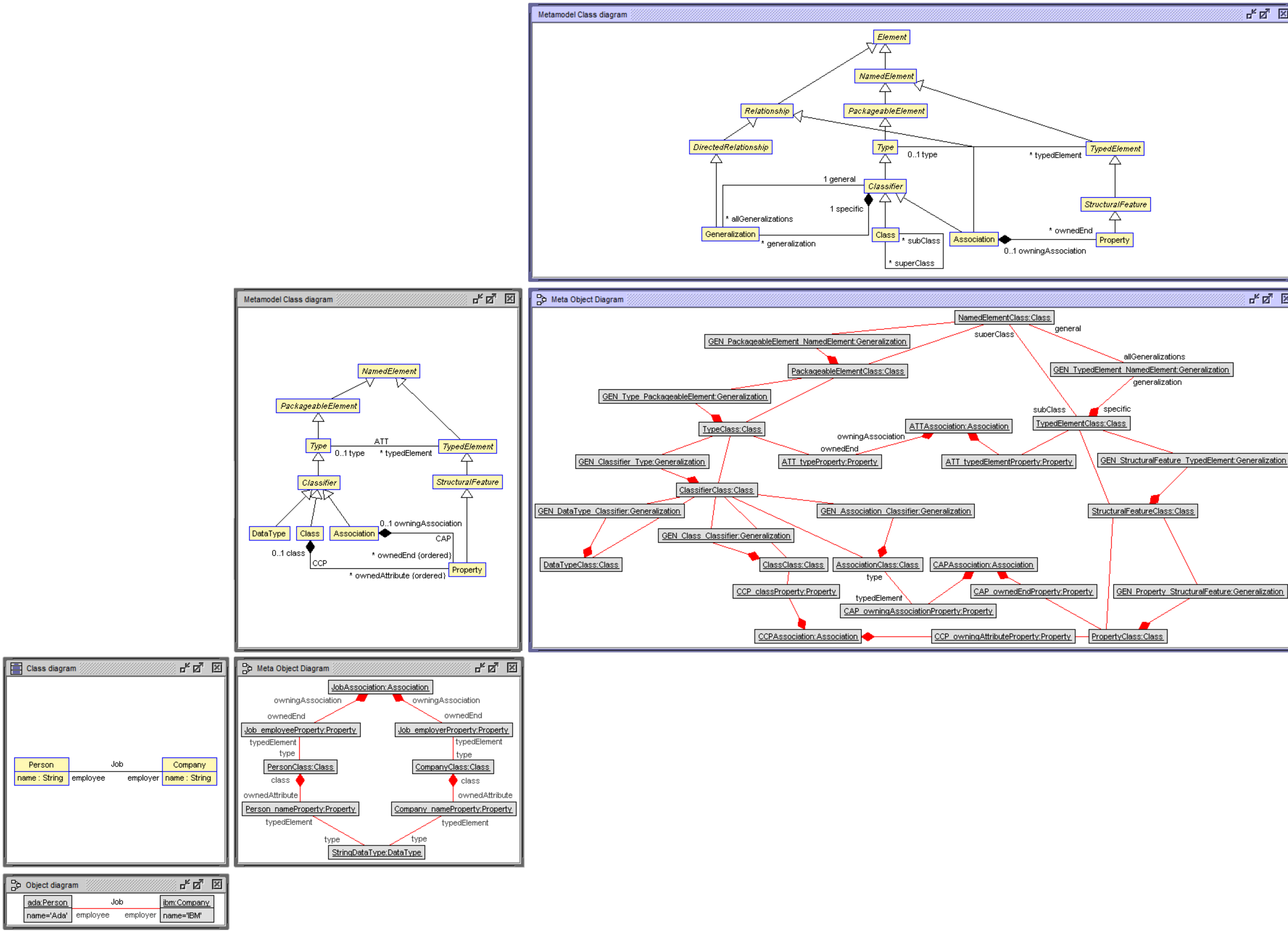
Classes being 'simple' specializations



Classes involved in two Generalizations



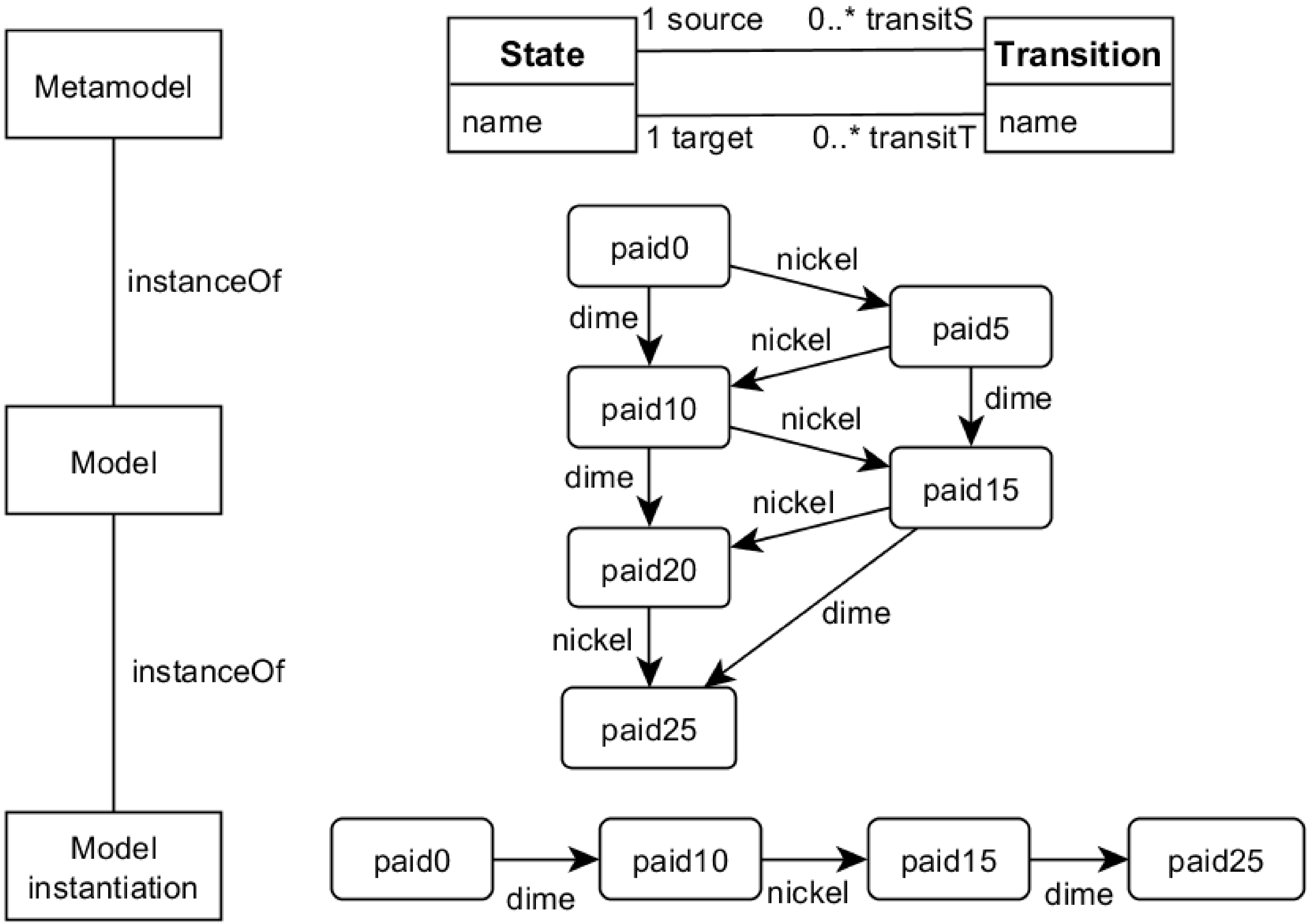
Manifestation of OMG Four Level through USE-MM



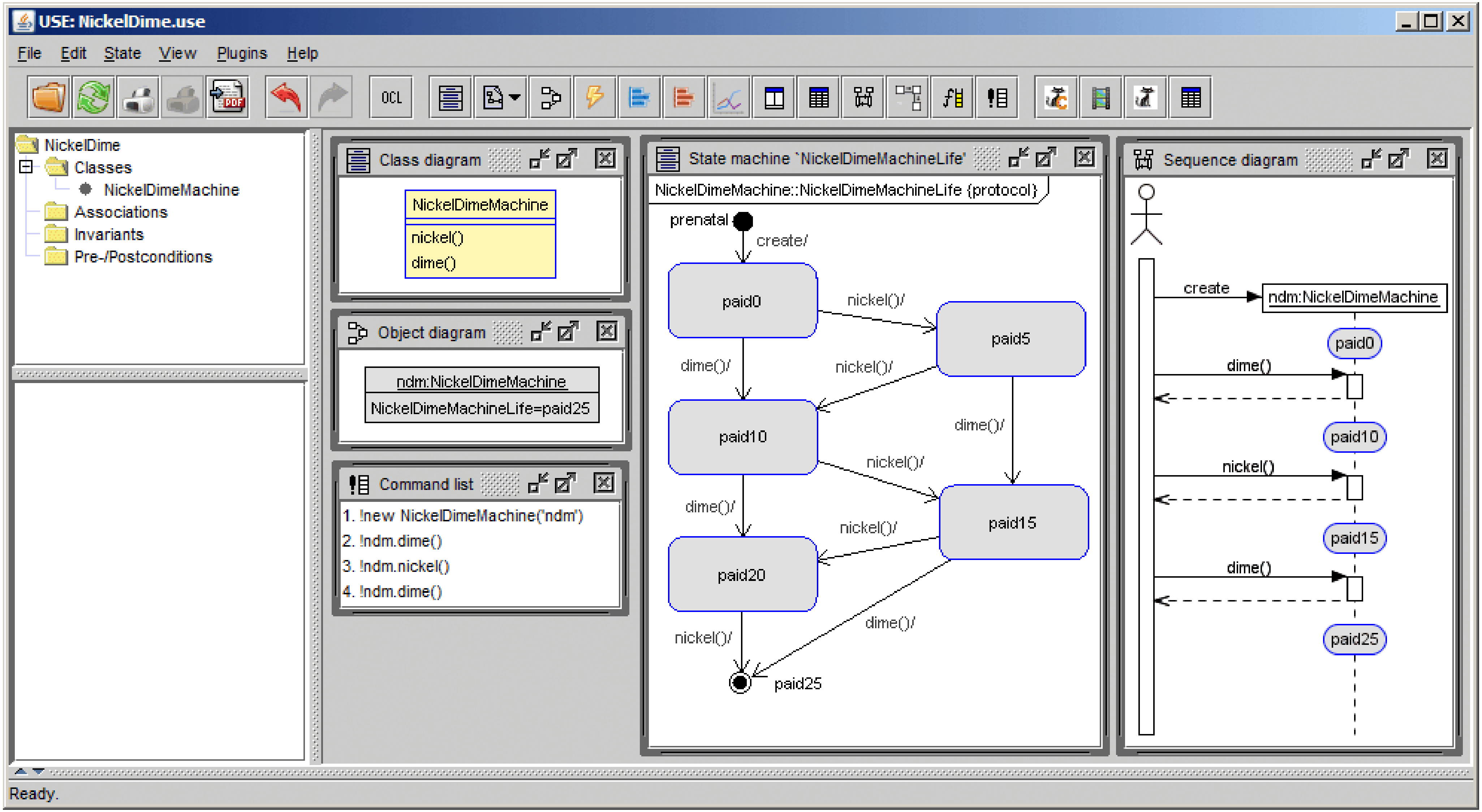
Behavioral Metamodels

- UML MM mainly describes structural aspects
- Behavioral aspects can be handled in metamodels as well
- Example: State machines
- General three level metamodel
- Realization in USE
- Metamodels with more than three levels possible

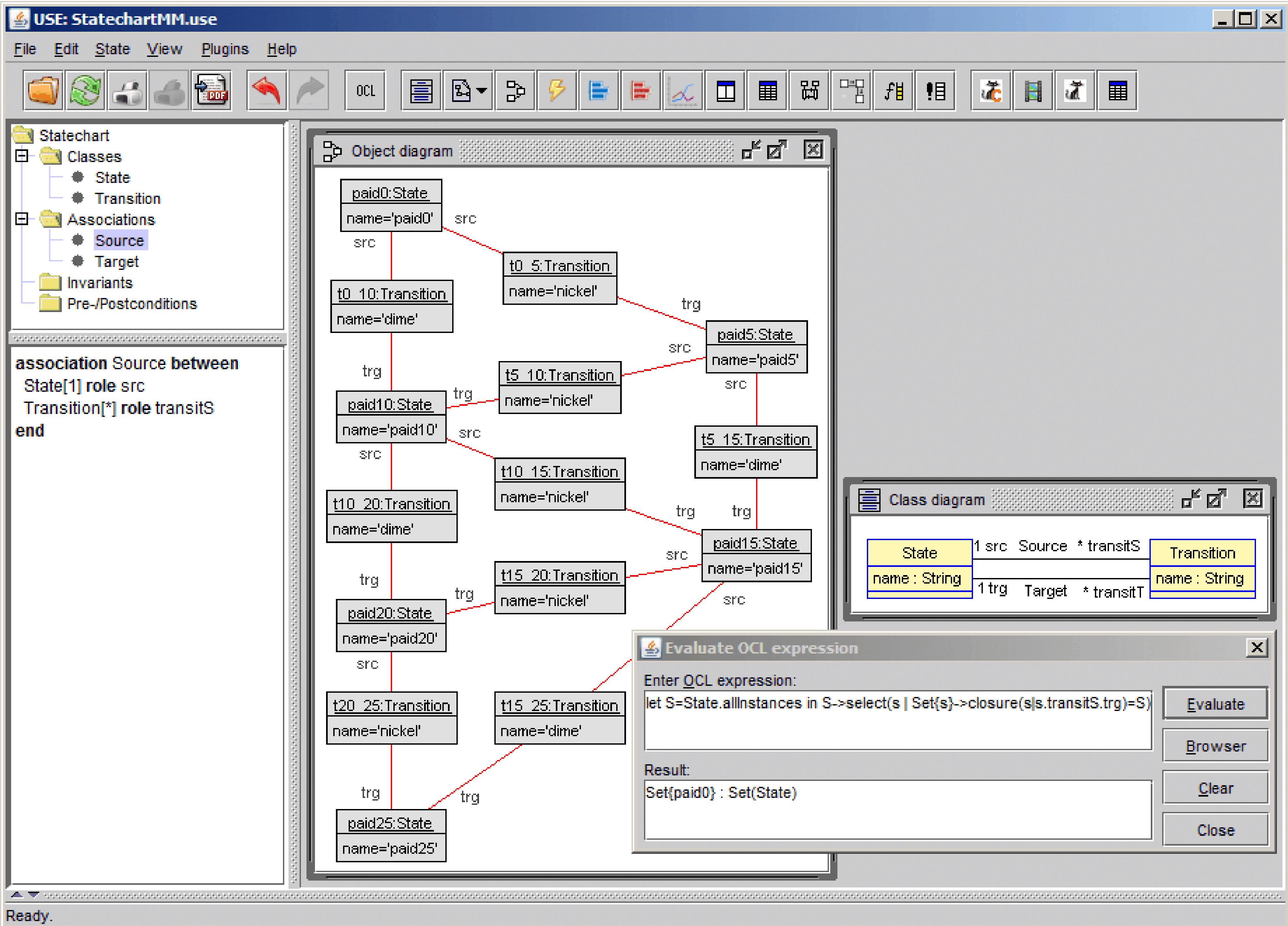
Three level metamodeling



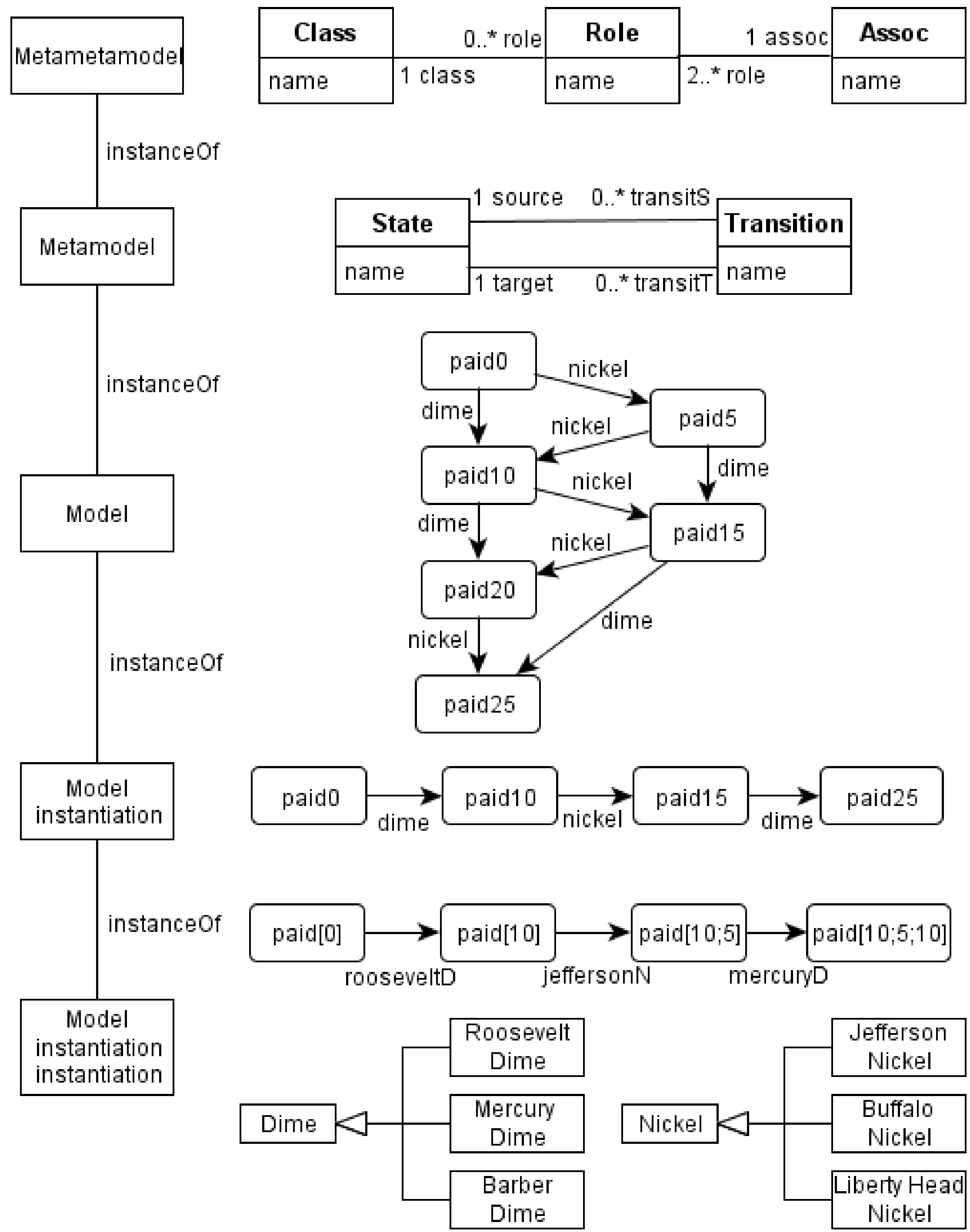
Nickel-Dime machine in USE



State machines in USE



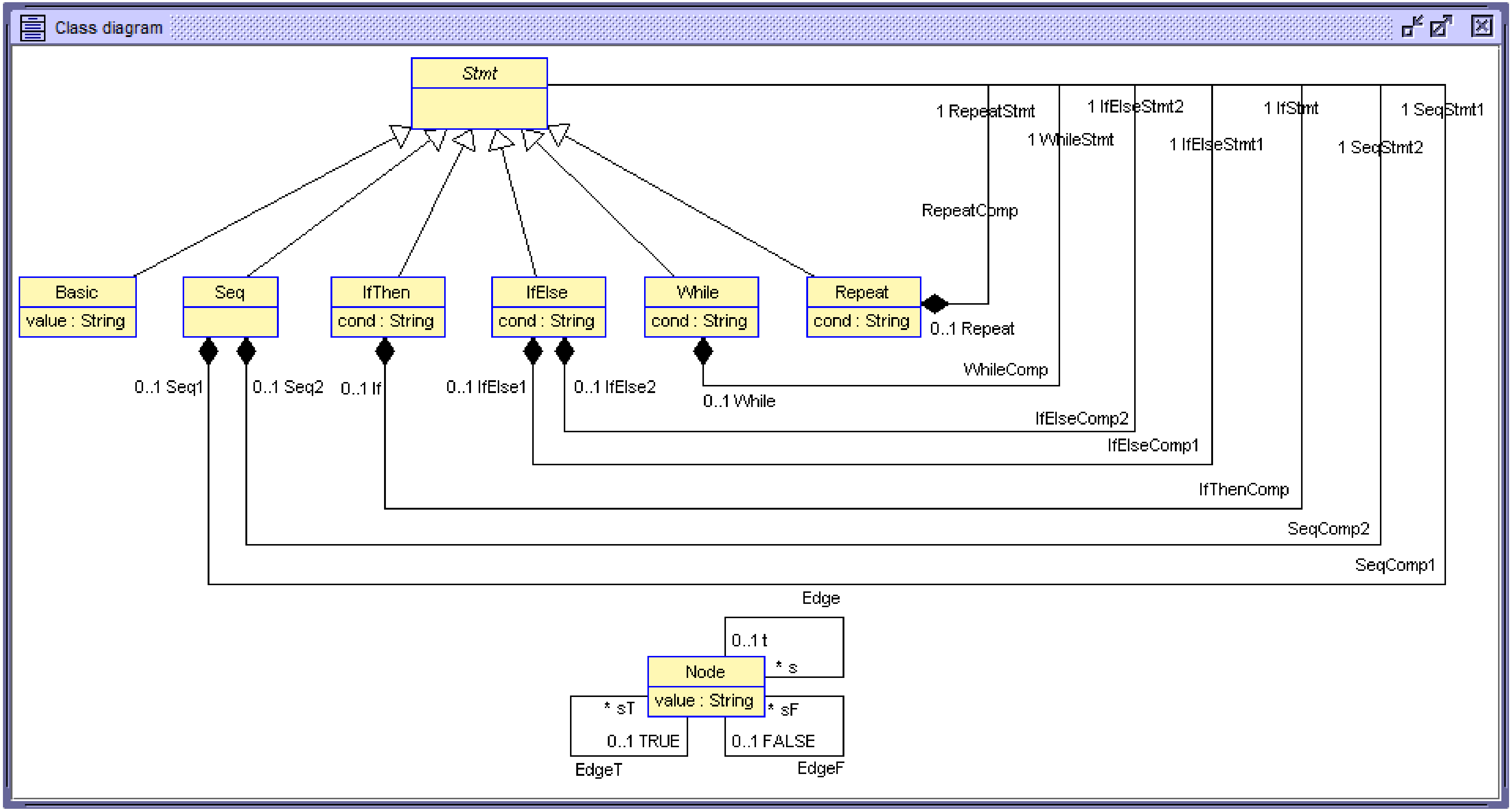
Five level metamodeling



Metamodels: an alternative for language specification

- Usually languages in Computer Science are described with grammars together with an execution mechanism (operational evaluation)
- Metamodels present an alternative
- Syntax and semantics (execution) can be described
- Approach explained by means of a very simple programming language ProgLang
- Two examples
  - Factorial
  - Abstract example with all syntactical options
- Advantage of metamodels for language specification: common description technique (UML/MOF and OCL) for syntax and semantics (execution)

Metamodel for ProgLang Syntax and Semantics



Context-free Grammar for ProgLang

statement ::= id |

statement; statement |

IF id THEN statement END |

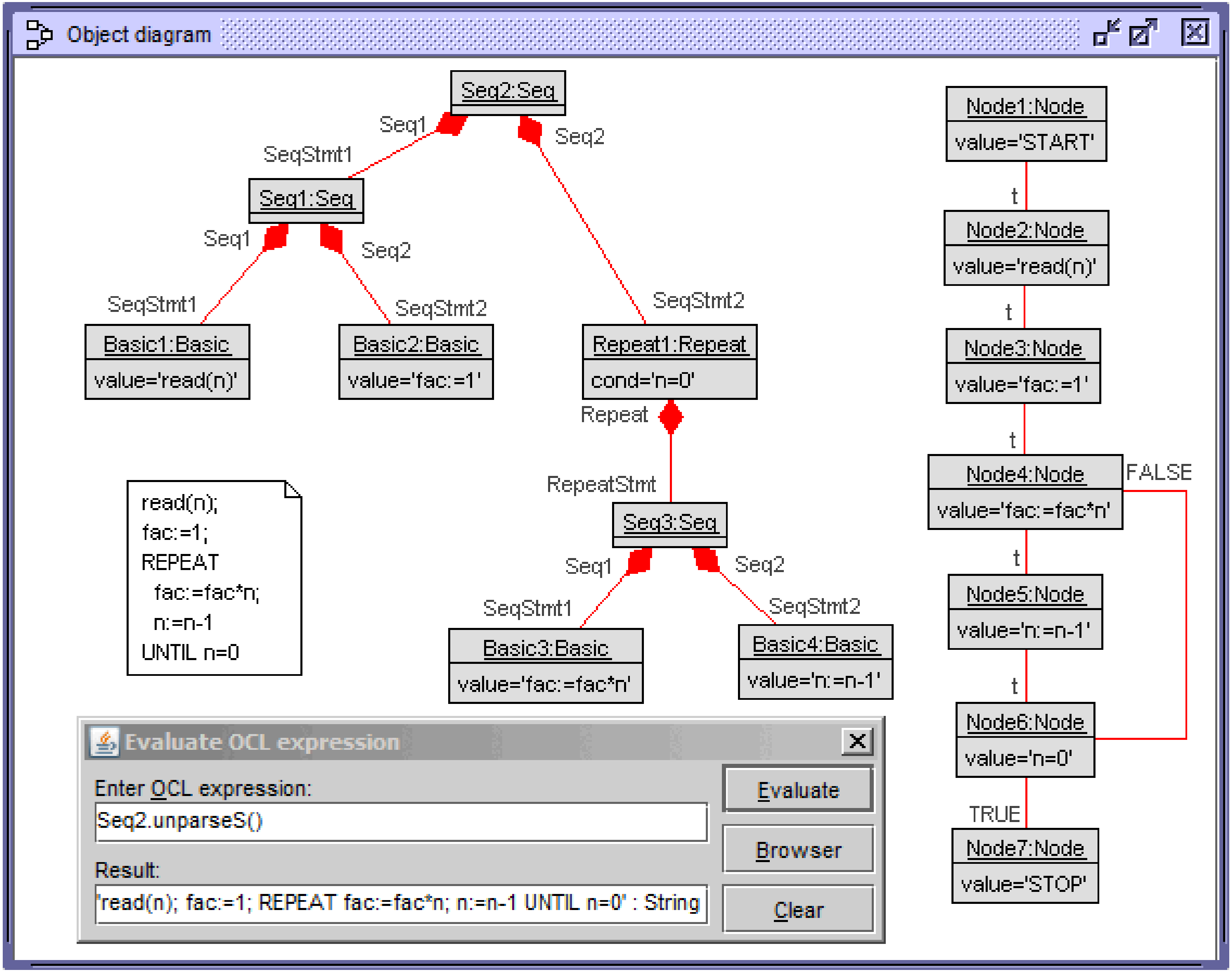
IF id THEN statement ELSE statement END |

WHILE id DO statement END |

REPEAT statement UNTIL id

- Production grammar → New specialized class for Stmt
- Non-terminal statement on right side → Black diamond to Stmt
- Keywords (IF, THEN, …) become part of an operation unparseS()

Factorial in ProgLang

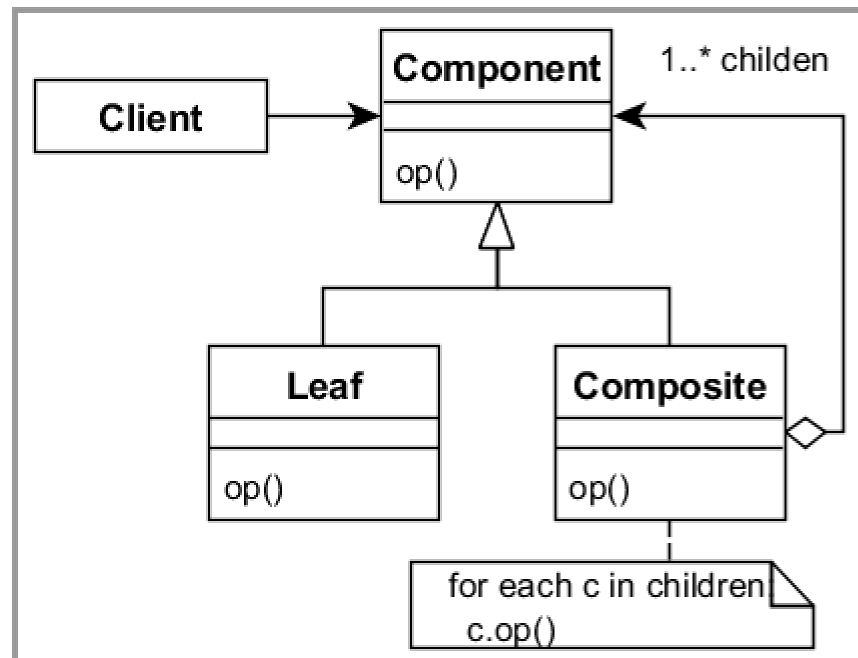


Factorial in ProgLang

- Left: syntax tree in form of an object diagram
- Utilizing UML composition is a natural way to build syntax trees; objects are connected to at most one aggregate; object diagrams with composition are acyclic
- Right: flow graphs in form of object diagram for 'compiled code' / execution
- Structuring control flow statements (if-then, if-then-else, while-do, repeat-until) have been represented by flows graphs
- Invariants (not shown) handle the connection between syntax and evaluation
- Operation unparseS() retrieves the source text from the syntax tree

Excursus: Composite pattern

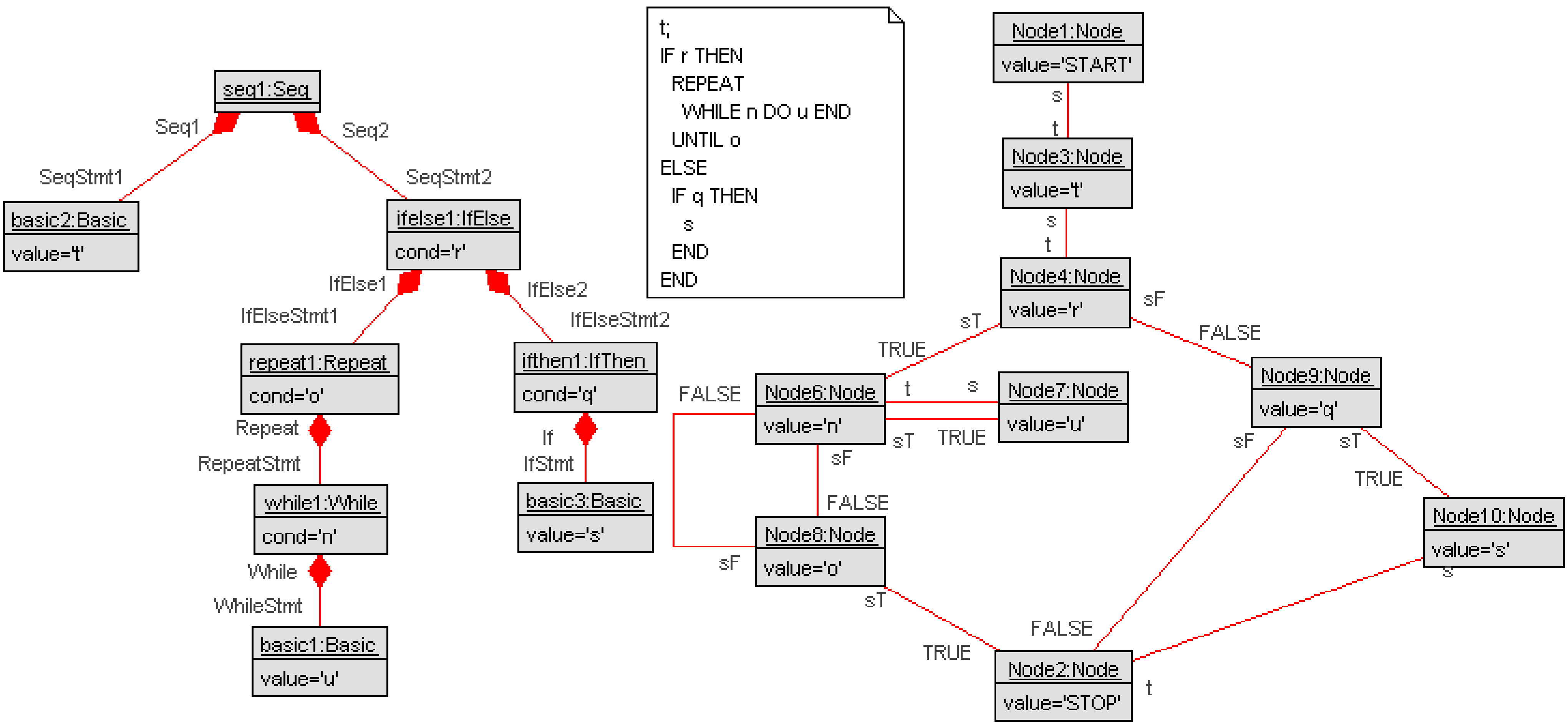
- The composite pattern is a partitioning design pattern
- The composite pattern describes a group of objects that is treated the same way as a single instance of the same type of object
- The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies
- Implementing the composite pattern lets clients treat individual objects and compositions uniformly
- The Client class does not refer to the Leaf and Composite classes directly (separately); instead, the Client refers to the common Component interface and can treat Leaf and Composite uniformly



Operation unparseS()

```
Stmt::unparseS():String = null

Basic::unparseS():String = self.value

Seq::unparseS():String = self.SeqStmt1.unparseS().concat('; ').
  concat(self.SeqStmt2.unparseS())

IfThen::unparseS():String =
  'IF '.concat(self.cond).concat(' THEN ').
  concat(self.IfStmt.unparseS()).concat(' END')

IfElse::unparseS():String =
  'IF '.concat(self.cond).concat(' THEN ').
  concat(self.IfElseStmt1.unparseS()).concat(' ELSE ').
  concat(self.IfElseStmt2.unparseS()).concat(' END')

While::unparseS():String =
  'WHILE '.concat(self.cond).concat(' DO ').
  concat(self.WhileStmt.unparseS()).concat(' END')

Repeat::unparseS():String =
  'REPEAT '.concat(self.RepeatStmt.unparseS()).
  concat(' UNTIL ').concat(self.cond)
```

Abstract example in ProgLang



Thanks for your attention!