ML2

MULTI-LEVEL MODELING LANGUAGE

HISTORY, THEORY AND PRACTICE

Structure

- People
- Background
- MLT*
- ML2 Language
- Example Model
- ML2 Editor
- Installing ML2
- Questions and Answers

People

Victorio Albani Carvalho

MLT and MLT*

João Paulo Andrade Almeida

Supervisor of the MLT team

Claudenir Morais Fonseca

ML2 and MLT*

Giancarlo Guizzardi

Co-supervisor of the MLT team

Fred Brasileiro

MLT for the Semantic Web

Background

MLT
MULTI-LEVEL
MODELING THEORY

- MLT was defined in 2015 with the work of Victorio Carvalho, at the time, PhD student at the Federal University of Espírito Santo (UFES) – Brazil.
- He worked under the supervision of João Paulo A. Almeida and Giancarlo Guizzardi in order to provide understanding of multi-level ontologies
- As result, he proposed the MLT theory as theoretical foundation for the comprehension of MLM

Background

MLT*
MULTI-LEVEL
MODELING THEORY

- MLT provided a solid foundation for MLM, organizing multi-level entities whose possible instances fall within a single instantiation order
- MLT* emerged as a generalized version MLT able to account for orderless entities, whose possible instances fall into different instantiation orders
- In that sense, MLT* is able to account for very general types, such as Entity or Thing

Background

ML2
MULTI-LEVEL
MODELING LANGUAGE

- The theory informed the design of a textual syntax to allow the specification of MLT* based models
  - ML2 is described in the M.Sc. Thesis of Claudenir Fonseca
- The ML2 language allows the user to define all sorts of entities and relations foreseen by the theory
- Additionally, other basic features of modeling languages are also provided, such as attributes, references and generalization sets

MLT*: Theoretical Basis for Multi-Level Conceptual Modeling

- Theory for interpreting multi-level domains
- Described in first-order logics
- Formalized (Alloy and TPTP)
- Relies solely on the *instance of* relation in order to build its theorems and definitions

Before language comes understanding

- To help us understand, accessible formalization

MLT
MULTI-LEVEL
MODELING THEORY

- Here we present only a non-temporal/non-modal version of the theory for simplicity

Basic Notions

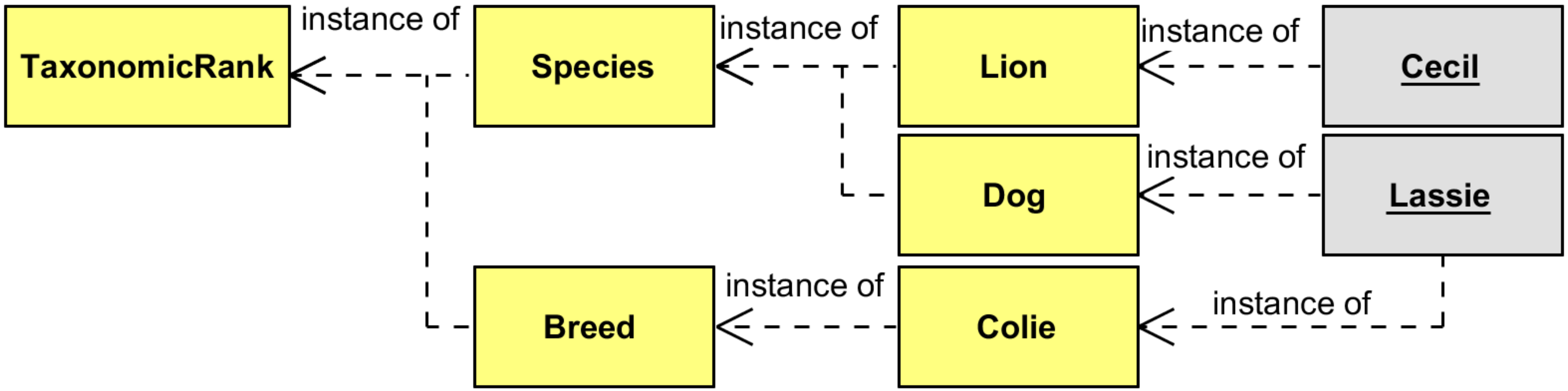
- Types and Individuals

  $\forall x(\text{individual}(x) \leftrightarrow \neg\exists y(\text{iof}(y,x)))$

  $\forall x(\text{type}(x) \leftrightarrow \exists y(\text{iof}(y,x)))$

- Well-founded types: MLT* types must have a possible instantiation chain that leads to some individual instance

  $\forall t(\text{type}(t) \rightarrow \exists x(\text{individual}(x) \land \text{iof}'(x,t)))$



Structural Relations

- Specialization

$$\forall t_1, t_2 \text{ specializes}(t_1, t_2) \leftrightarrow$$
$$\text{type}(t_1) \land \text{type}(t_2) \land \forall e(\text{iof}(e, t_1) \rightarrow \text{iof}(e, t_2))$$

- Proper specialization

$$\forall t_1, t_2 \text{ properSpecializes}(t_1, t_2) \leftrightarrow$$
$$(\text{specializes}(t_1, t_2) \land \neg(t_1 = t_2))$$

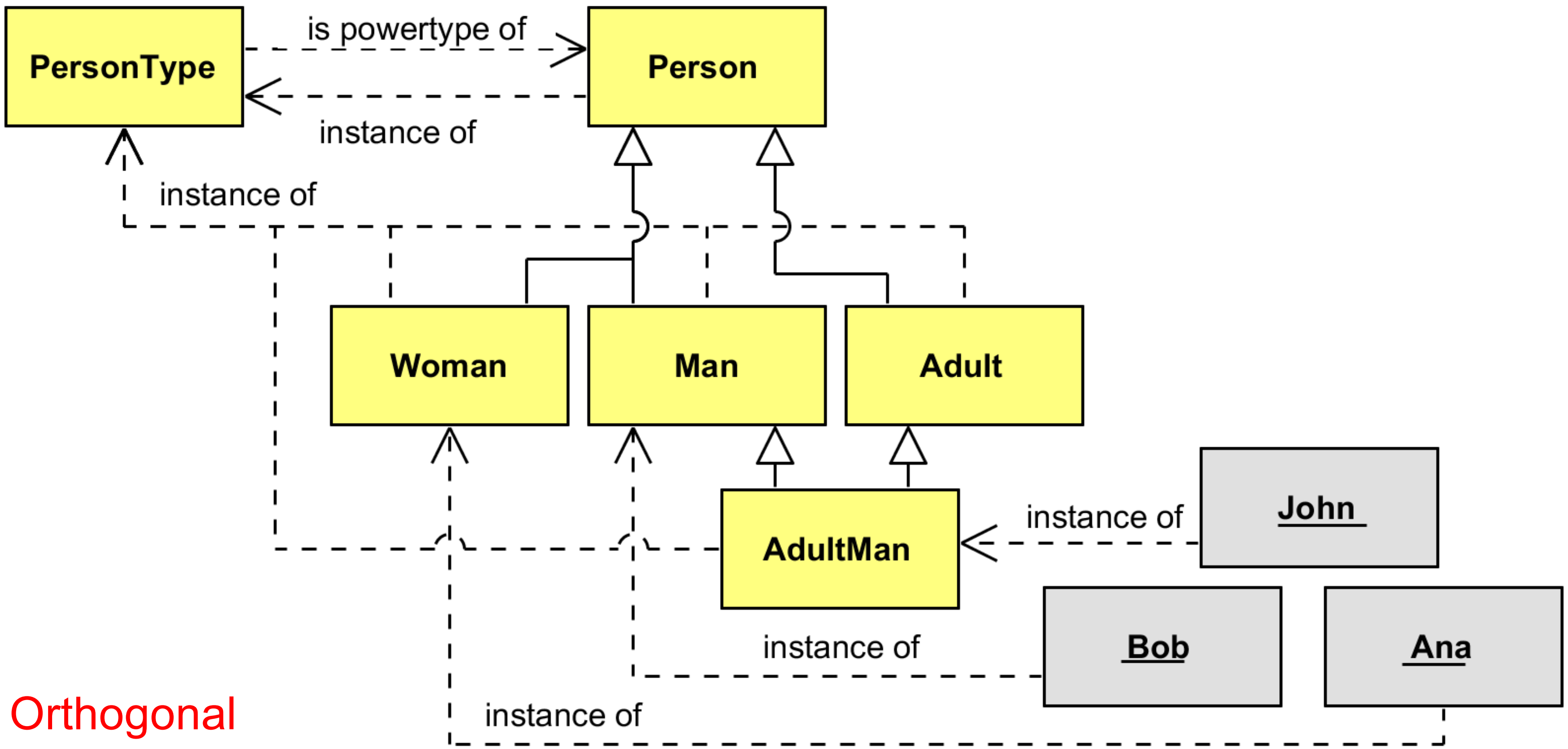
- Equality

$$\forall t_1, t_2((\text{type}(t_1) \land \text{type}(t_2)) \rightarrow$$
$$(t_1 = t_2) \leftrightarrow \forall x(\text{iof}(x, t_1) \leftrightarrow \text{iof}(x, t_2)))$$

- Powertype

$$\forall t_1, t_2 \text{ isPowertypeOf}(t_1, t_2) \leftrightarrow$$
$$\text{type}(t_1) \land \forall t_3(\text{iof}(t_3, t_1) \leftrightarrow \text{specializes}(t_3, t_2))$$

Structural Relations



Orthogonal specializations

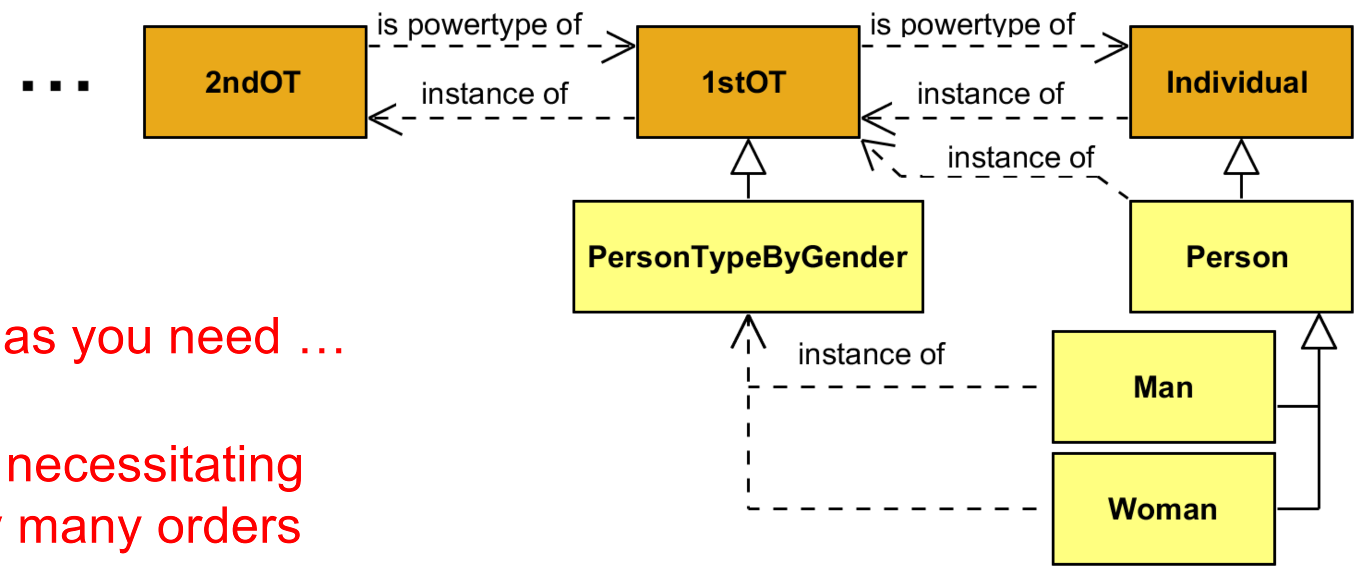
Accounting for Stratification

- The "Individual" constant

$$\forall t((t = \text{Individual}) \leftrightarrow \forall x(\text{individual}(x) \leftrightarrow \text{iof}(x, t)))$$

- Basic types

$$\forall b_i(\text{basictype}(b_i) \leftrightarrow (b_i = \text{Individual}) \lor \exists b_{i-1}(\text{basictype}(b_{i-1}) \land \text{isPowertypeOf}(b_i, b_{i-1}))))$$



As high as you need …

Without necessitating infinitely many orders

Beyond Stratification

- Ordered and Orderless Types

$$\forall x(\text{orderedtype}(x) \leftrightarrow \exists b(\text{basictype}(b) \land \text{specializes}(x, b)))$$

$$\forall x(\text{orderlesstype}(x) \leftrightarrow \text{type}(x) \land \neg\text{orderedtype}(x))$$

- Here you can decide whether you want to:
  - Commit to orderless types
    - Telos ω-properties
    - Cyc VariedOrderCollections
  - Commit to ordered types only (strictly stratified)
  - (or leave your theory general, so it encompasses both possibilities)

Special cases

- Two-level models:
- Just add an axiom stating that the only basic type is Individual

- Infinitely many orders:
- Just add an axiom stating that for every type there is a powertype

OrderedType, OrderlessType, Type, Entity

- Ordered and Orderless Types

  $\forall x(\text{orderedtype}(x) \leftrightarrow$
  $\exists b(\text{basictype}(b) \land \text{specializes}(x, b)))$

  $\forall x(\text{orderlesstype}(x) \leftrightarrow \text{type}(x) \land \neg\text{orderedtype}(x))$

- Theory constants

  $\forall t(t = \text{OrderedType} \leftrightarrow$
  $\forall x(\text{orderedtype}(x) \leftrightarrow \text{iof}(\mathrm{x}, \mathrm{t})))$

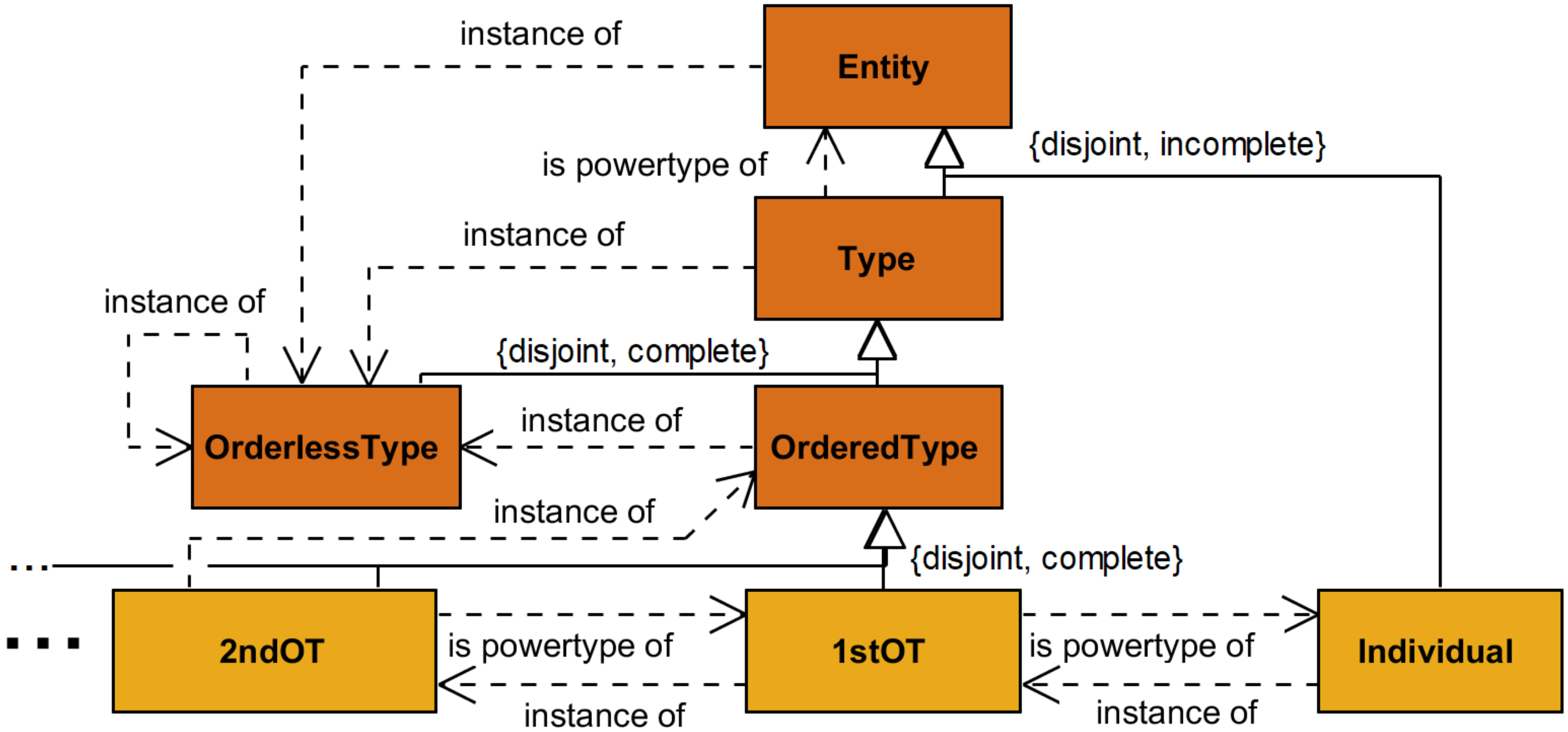
  $\forall \mathrm{t}(\mathrm{t} = \text{OrderlessType} \leftrightarrow$
  $\forall \mathrm{x}(\text{orderlesstype}(\mathrm{x}) \leftrightarrow \text{iof}(\mathrm{x}, \mathrm{t})))$

  $\forall \mathrm{t}((\mathrm{t} = \text{Type}) \leftrightarrow \forall \mathrm{x}(\text{type}(\mathrm{x}) \leftrightarrow \text{iof}(\mathrm{x}, \mathrm{t})))$

  $\forall \mathrm{t}(\mathrm{t} = \text{Entity} \leftrightarrow \forall \mathrm{x}(\text{iof}(\mathrm{x}, \mathrm{t})))$

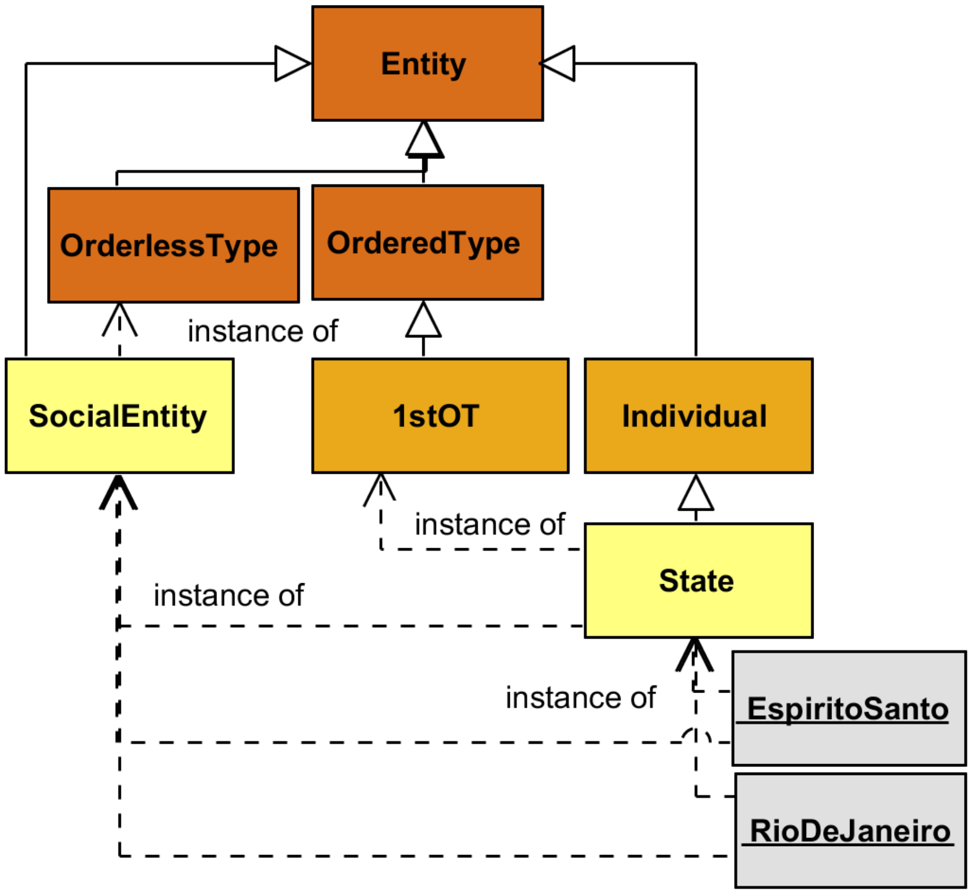
Beyond Stratification

- The constants of the theory build a top-level model that can be used for the interpretation of multi-level scenarios
- The relations among these entities are consequences of their very definitions



Beyond Stratification

- An example of a domain type that defies the stratified scheme is “Social Entity”, whose extension includes both individuals and other types



Structural Relations

- Categorization

$$\forall t_1, t_2(\text{categorizes}(t_1, t_2) \leftrightarrow (\neg\text{iof}(t_1, \text{Individual}) \land \forall t_3(\text{iof}(t_3, t_1) \rightarrow \text{properSpecializes}(t_3, t_2))))$$

- Complete Categorization

$$\forall t_1, t_2(\text{completelyCategorizes}(t_1, t_2) \leftrightarrow (\text{categorizes}(t_1, t_2) \land \forall e(\text{iof}(e, t_2) \rightarrow \exists t_3((\text{iof}(e, t_3) \land \text{iof}(t_3, t_1))))))$$

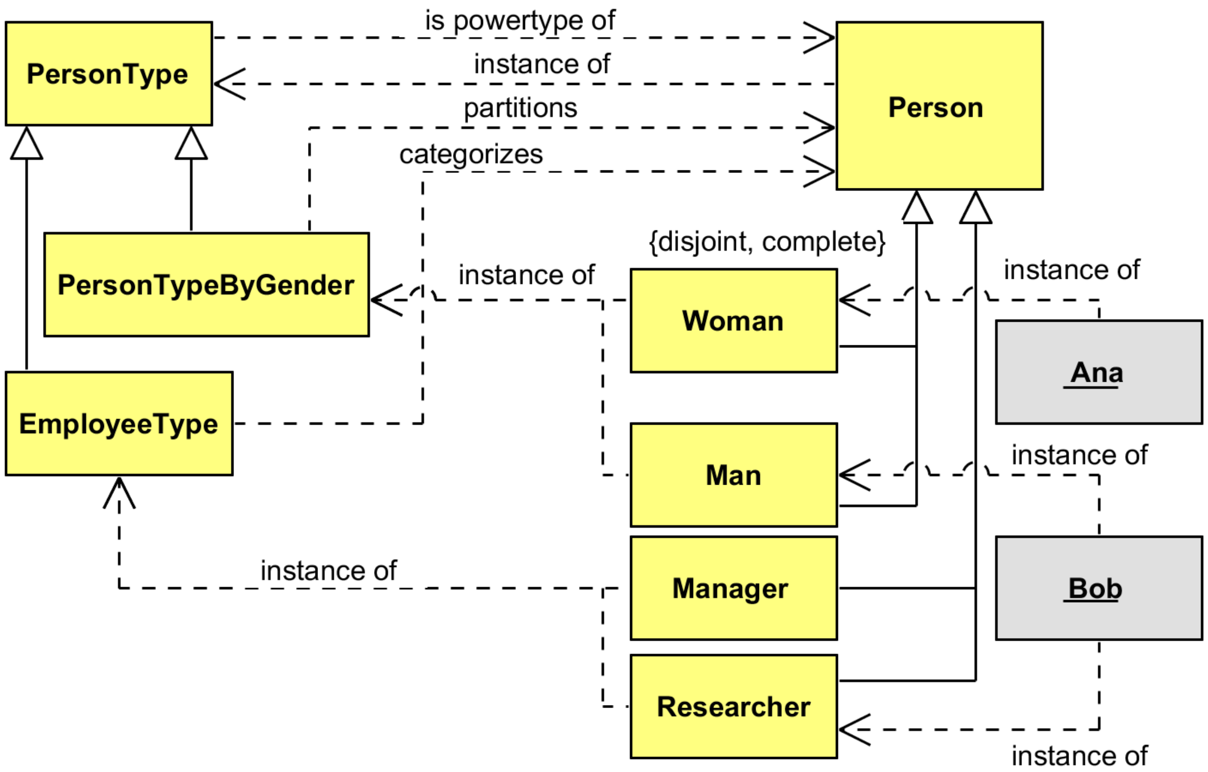
- Disjoint Categorization

$$\forall t_1, t_2(\text{disjointlyCategorizes}(t_1, t_2) \leftrightarrow (\text{categorizes}(t_1, t_2) \land \forall e, t_3, t_4((\text{iof}(t_3, t_1) \land \text{iof}(t_4, t_1) \land \text{iof}(e, t_3) \land \text{iof}(e, t_4)) \rightarrow t_3 = t_4)))$$

Structural Relations

- Partitions

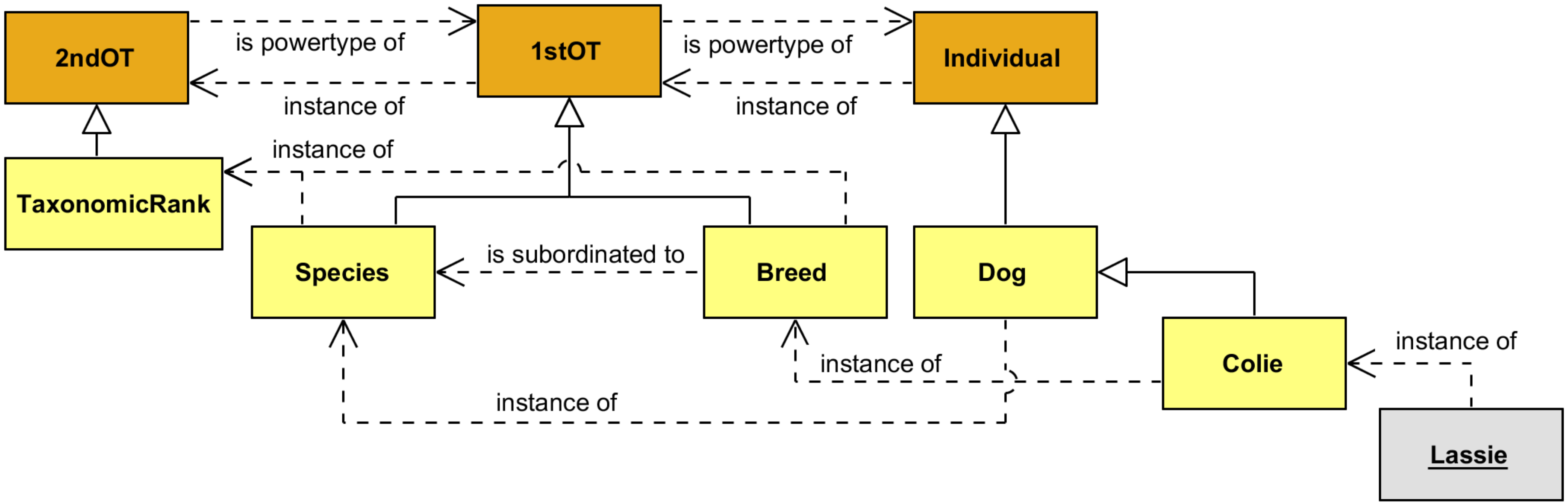
$$\forall t_1, t_2(\text{partitions}(t_1, t_2) \leftrightarrow (\text{completelyCategorizes}(t_1, t_2) \wedge \text{disjointlyCategorizes}(t_1, t_2)))$$



Structural Relations

- Subordination

$$\forall t_1, t_2(\text{isSubordinate}(t_1, t_2) \leftrightarrow (\neg\text{iof}(t_1, \text{Individual}) \land \forall t_3(\text{iof}(t_3, t_1) \rightarrow \exists t_4((\text{iof}(t_4, t_2) \land \text{properSpecializes}(t_3, t_4))))))$$



Structural Relations

- During the formalization of the theory, a set of theorems emerged as constraints on the properties of the relations, as well as for their domains and ranges

<table>
<tr><th>Relation (t → t')</th><th>Domain</th><th>Range</th><th>Constraint</th><th>Properties</th></tr>
<tr><td rowspan="3">specializes(t,t')</td><td>Orderless</td><td>Orderless</td><td rowspan="6">if t and t' are ordered types, they must be at the same type order</td><td rowspan="3">Reflexive, antissymetric, transitive</td></tr>
<tr><td>Ordered</td><td>Orderless</td></tr>
<tr><td>Ordered</td><td>Ordered</td></tr>
<tr><td rowspan="3">properSpecializes(t,t')</td><td>Orderless</td><td>Orderless</td><td rowspan="3">Irreflexive, antissymetric, transitive</td></tr>
<tr><td>Ordered</td><td>Orderless</td></tr>
<tr><td>Ordered</td><td>Ordered</td></tr>
<tr><td rowspan="2">isPowertypeOf(t,t')</td><td>Orderless</td><td>Orderless</td><td rowspan="2">t cannot be a first-order type if t and t' are ordered types, t must be at a type order immediately above the order of t'</td><td rowspan="2">Irreflexive, antissymetric, antitransitive</td></tr>
<tr><td>Ordered</td><td>Ordered</td></tr>
</table>

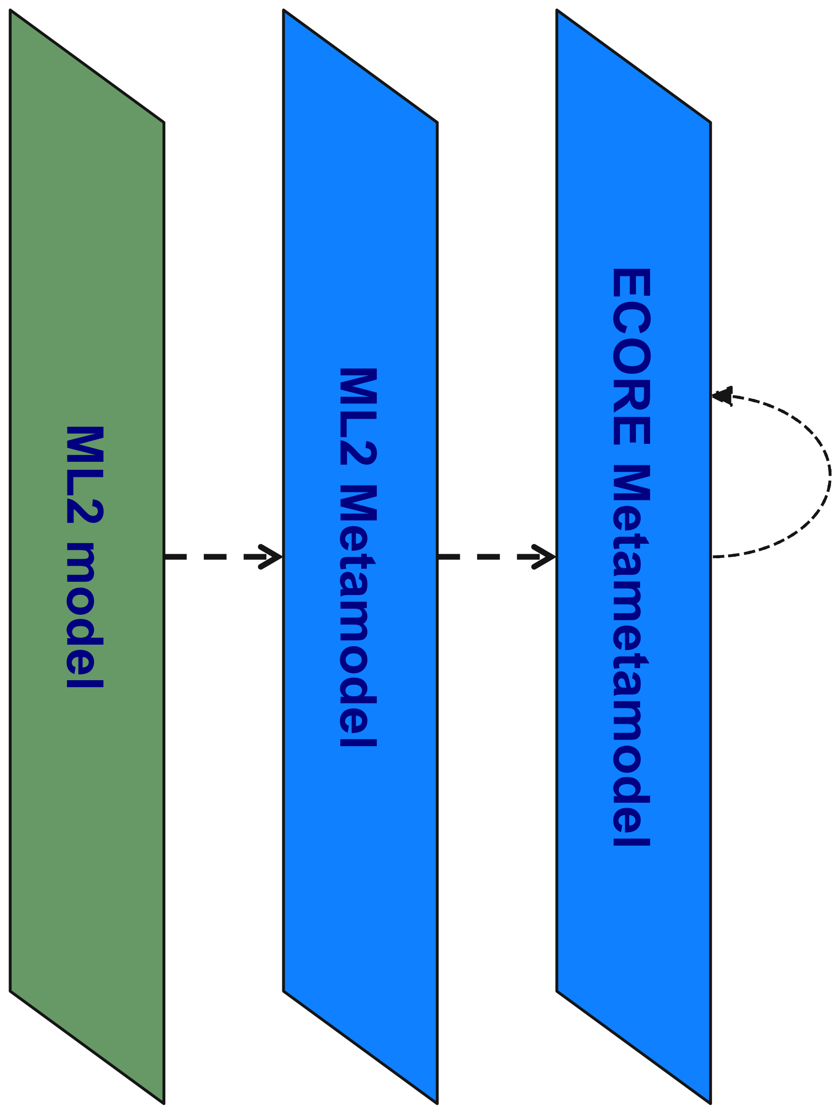
Structural Relations

- During the formalization of the theory, a set of theorems emerged as constraints on the properties of the relations, as well as for their domains and ranges

<table>
<tr><th>Relation (t → t')</th><th>Domain</th><th>Range</th><th>Constraint</th><th>Properties</th></tr>
<tr><td rowspan="3">categorizes(t,t')
disjointlyCategorizes(t,t')</td><td>Orderless</td><td>Orderless</td><td rowspan="5">t cannot be a first-order type
if t and t' are ordered types, t must be at a type order immediately above the order of t'</td><td rowspan="3">Irreflexive, antissymetric, nontransitive</td></tr>
<tr><td>Ordered</td><td>Orderless</td></tr>
<tr><td>Ordered</td><td>Ordered</td></tr>
<tr><td rowspan="2">completelyCategorizes(t,t')
partitions(t,t')</td><td>Orderless</td><td>Orderless</td><td rowspan="2">Irreflexive, antissymetric, antitransitive</td></tr>
<tr><td>Ordered</td><td>Ordered</td></tr>
<tr><td rowspan="3">isSubordinatedTo(t,t')</td><td>Orderless</td><td>Orderless</td><td rowspan="3">t and t' cannot be first-order types
if t and t' are ordered types, they must be at the same type order</td><td rowspan="3">Irreflexive, antissymetric, transitive</td></tr>
<tr><td>Ordered</td><td>Orderless</td></tr>
<tr><td>Ordered</td><td>Ordered</td></tr>
</table>

ML2 Language

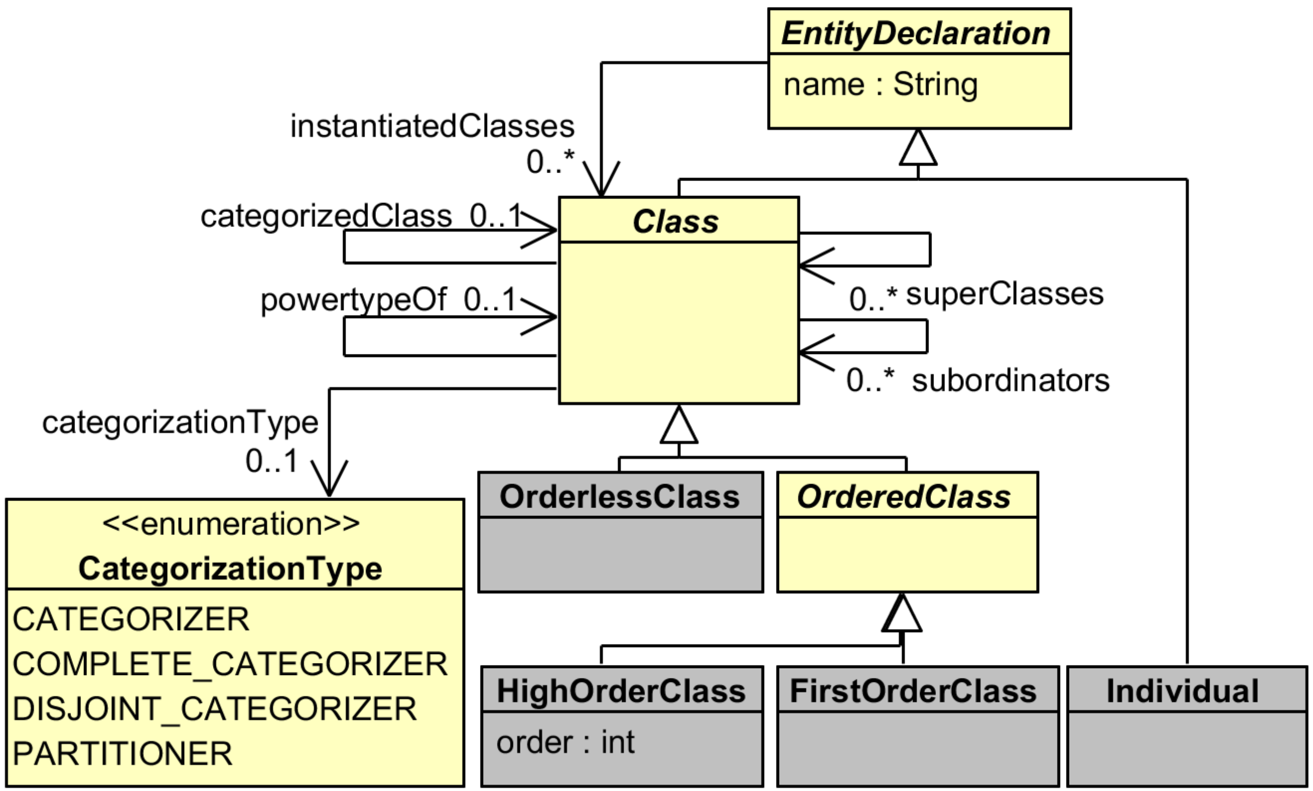
- Multi-Level Modeling Language
- Textual syntax
- Focused on the development of domain conceptual models
- Allows the specification of all sort of entities and relations foreseen by MLT*
- Incorporates MLT* rules as semantically-motivated languages constraints
- Support to other basic constructs of traditional modeling languages:
  - Attributes
  - References
  - Generalizations sets



ML2 Metamodel is quite
Similar to the MLT*

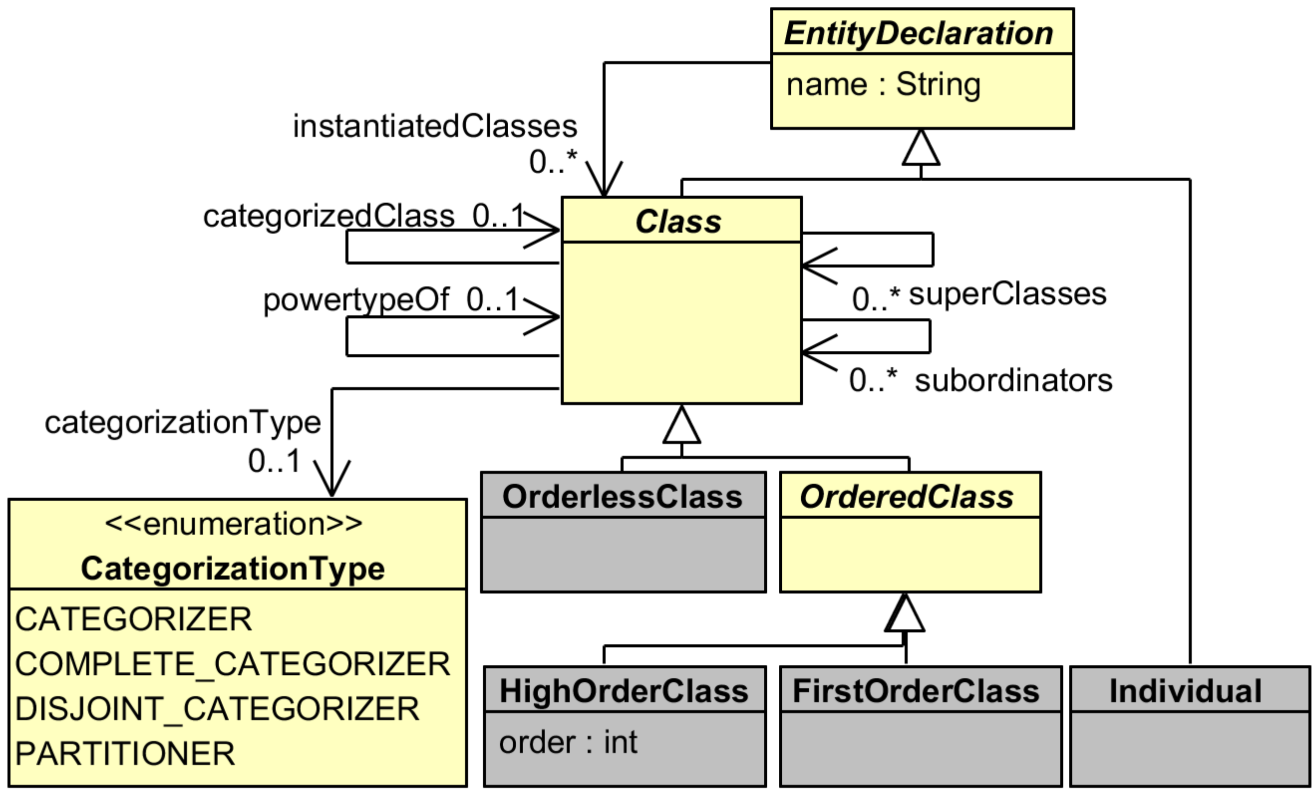
Core Concepts

- Core concepts of metamodel reflecting the theory constants
- Only metaclasses in gray can be instatiated



Core Concepts

- Classes and instances are handled both at the same level in regard to the metamodel
- The instantiation relation from ML2 is a common reference between two instances of the metamodel



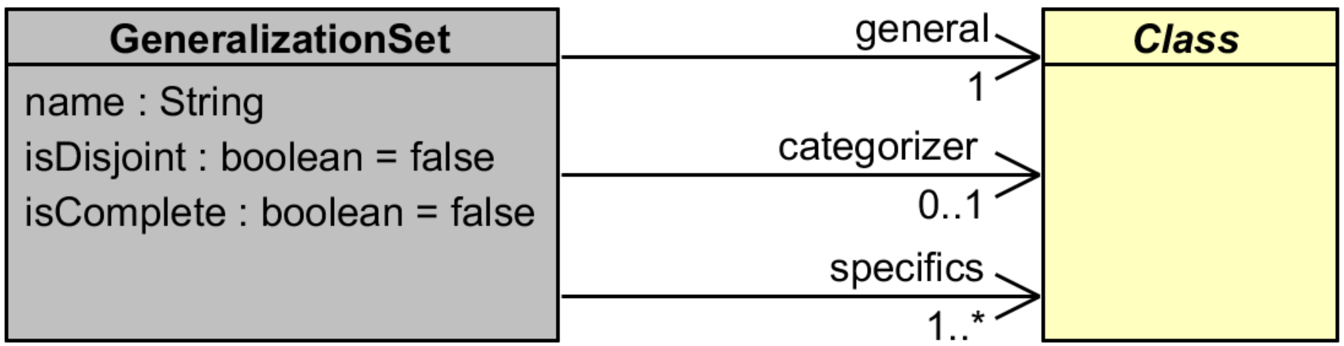
Core Concepts

- The simple syntax is design to improve readability
- Only high-order entities require the specification of an order
- For users of traditional two-level languages, the syntax syntax uses a familiar vocabulary for declaring common classes and instances

```
individual Eva : Entity , Person;

class Person : PersonType, Entity;

order 2 class PersonType : Entity isPowertypeOf Person;

orderless class Entity;
```

Generalization Sets

- Inspired on the UML usage of generalization sets
- Aggregates specializations of a common class that following the same criteria of definition
- Based on the powertype-pattern in UML, allows the identification of a categorizer class that represent the involved criteria
- Disjoint and complete constraints are also supported



Generalization Sets

- Both categorization relations and generalization sets affect the specializations of the base class
- Not all combinations of categorizations and disjoint/complete constraints are valid
- This aspect led to the definition of proper semantically-motivated constraints

Generalization Sets

- Syntactic constraints detect invalid combinations of generalization set constraints and categorization relations

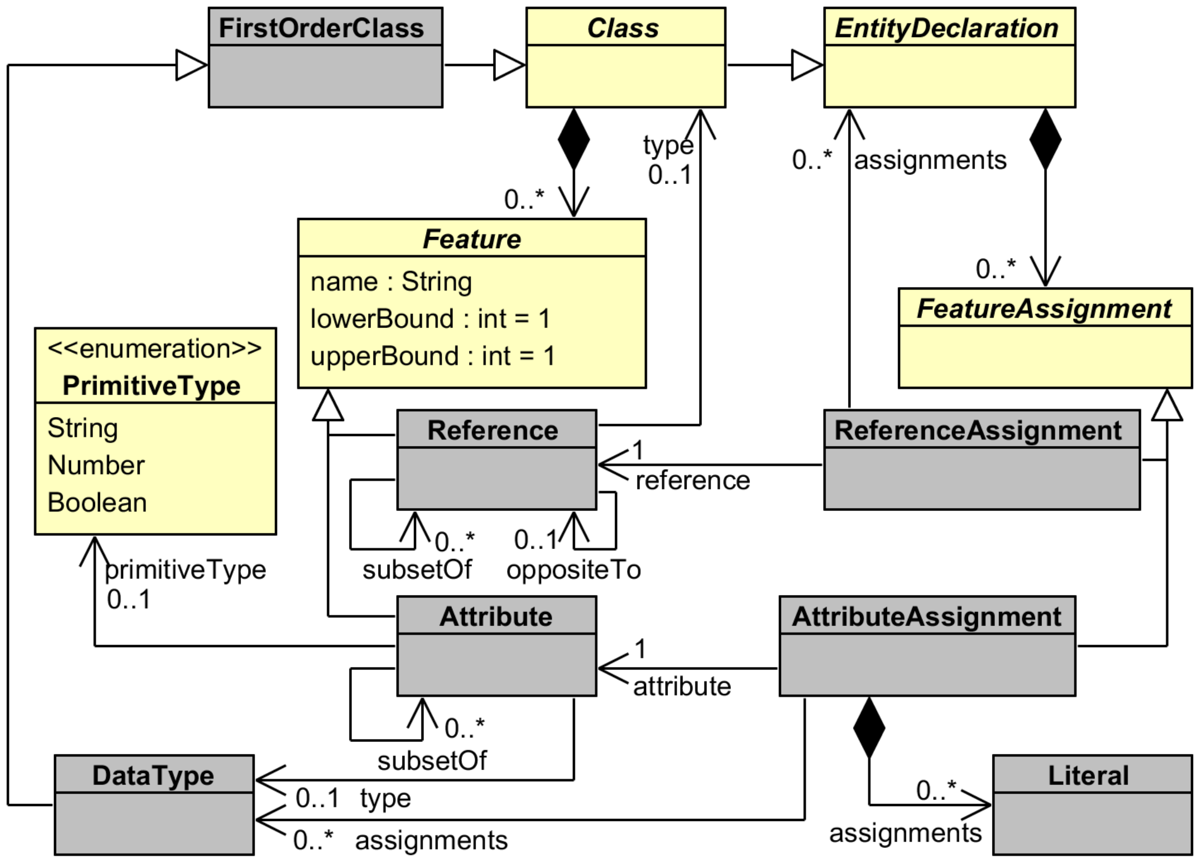
Categorization Relation	**Generalization Set Constraints**			
	Disjoint		***Overlapping***	
	Complete	***Incomplete***	***Complete***	***Incomplete***
Partitions	Enumerated	Not Enumerated	Invalid	Invalid
Disjoint Categorization	Invalid	Silent	Invalid	Invalid
Complete Categorization	Not Enumerated	Not Enumerated	Silent	Not Enumerated
Categorization	Invalid	Not Enumerated	Invalid	Silent

Generalization Sets

```
disjoint complete genset person_by_age
general Person
categorizer PersonTypeByAge
specifics Child, Teenager, Adult, Elder;


order 2 class PersonTypeByAge partitions Person;
class Person : PersonPowertype;


class Child : PersonTypeByAge specializes Person;
class Teenager : PersonTypeByAge specializes Person;
class Adult : PersonTypeByAge specializes Person;
class Elder : PersonTypeByAge specializes Person;
```

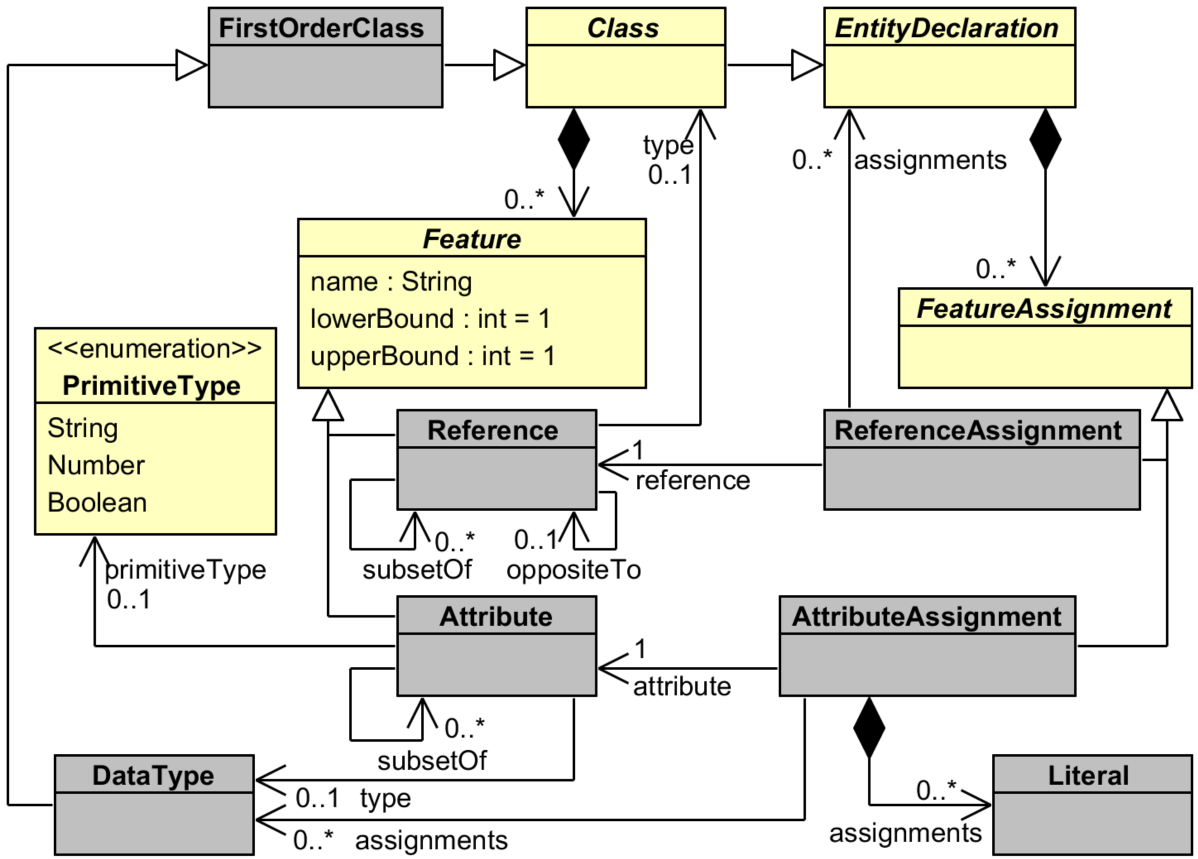
Features and Assignments

- ML2 supports the definition of features and assignments
- Features and assignments must be either attributes or references



Features and Assignments

- A reference's type can be any given class
- An attribute's type must be a primitive type (String, Number or Boolean) or some complex DataType



Features and Assignments

- Features and features assignments are handled at the same implementation level, allowing assignments for entities in any given order

```
orderless class Entity : Entity {
name : String
name = "Entity"
};
class Person : Entity {
name = "Person"
};
individual Elvis : Entity {
name = "Elvis Presley"
};
```

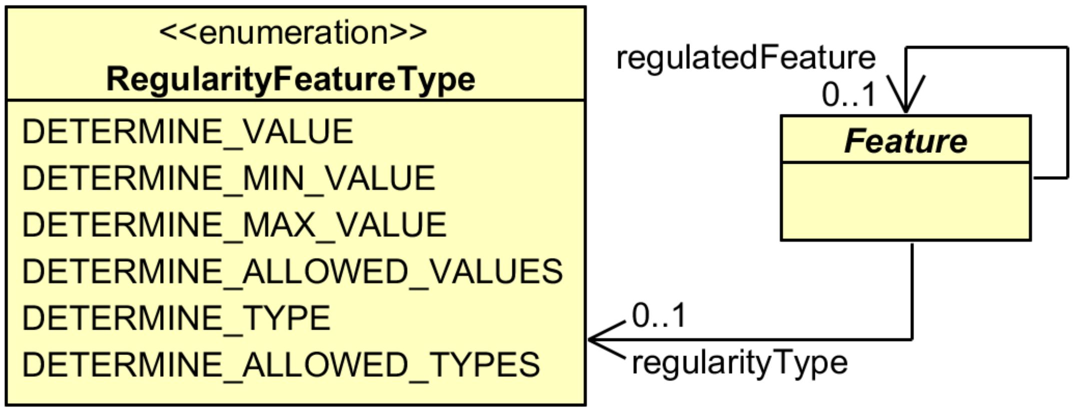
Features and Assignments

- ML2 features also support other common mechnisms in modelling
  - Cardinalities
  - Subsetting
  - Opposite references

```
orderless class Artifact {
    ref isCreatedBy : [0..*] Agent isOppositeTo creator
};
class Agent {
    ref creator : [0..*] Artifact isOppositeTo isCreatedBy
};
class Designer specializes Agent {
    ref designed : [0..*] Artifact subsets creator
};
```

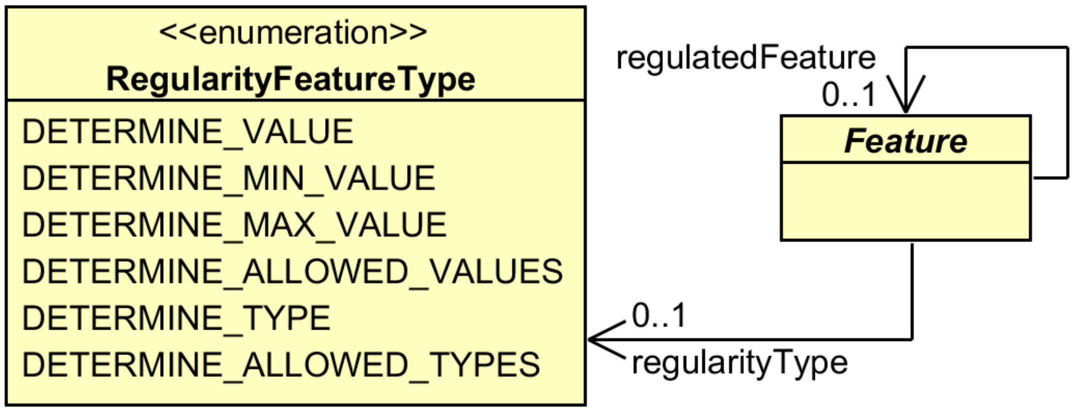
Regularity Features

- In addition to shallow instantiation, ML2 also supports deep instantiation through regularity features
- This mechanism allows features of higher order to regulate the assignments of others at a lower order



Regularity Features

- ML2 considers six type of regularities
  - Minimum Value
  - Maximum Value
  - Determined Value
  - Allowed Values
  - Determined Type
  - Allowed Types



Regularity Features

- Minimum and Maximum Values
  - The regularity feature determines the limits of that can be assigned to the regulated one

```
order 2 class CellphoneModel categorizes Cellphone {
regularity maximumStorageCapacity : Number
determinesMaxValue storageCapacity
regularity minimumStorageCapacity : Number
determinesMinValue storageCapacity
};
class Cellphone { storageCapacity : Number };
class IPhone5 : CellphoneModel specializes Cellphone {
maximumStorageCapacity = 64
minimumStorageCapacity = 16
};
```

Regularity Features

- Determine Value
  - The regularity feature determines the actual values that can be assigned to the regulated one
  - Assignment of the regularity features may add enough information to the model (see "Device321")

```
class Cellphone { screenSize : Number };
order 2 class CellphoneModel categorizes Cellphone {
regularity instancesScreenSize : Number determinesValue screenSize
};
class IPhone5 : CellphoneModel specializes Cellphone {
instancesScreenSize = 4.1
};
individual Device123 : IPhone5 { screenSize = 4.1 };
individual Device321 : IPhone5;
```

Regularity Features

- Allowed Values
  - The regularity feature determines the possible values to be assigned to the regulated one

```
datatype Color { red:Number green:Number blue:Number };
individual White:Color { red=255 green=255 blue=255 };
individual Red:Color { red=255 green=0 blue=0 };

class Cellphone { color : Color };
order 2 class CellphoneModel categorizes Cellphone {
regularity availableColors : [1..*] Color
determinesAllowedValues color
};
class IPhone5 : CellphoneModel specializes Cellphone { availableColors = {White,Red} };
individual WhiteDevice : IPhone5 { color=Red };
individual RedDevice : IPhone5 { color=White };
```

Regularity Features

- Determine Type
  - The regularity feature determines the actual type of entity that can be assigned to the regulated one

```
order 2 class ProcessorModel categorizes Processor;
class Processor;
class A6 : ProcessorModel specializes Processor;

order 2 class CellphoneModel categorizes Cellphone {
regularity ref compatibleProcessor : ProcessorModel
determinesType installedProcessor
};
class Cellphone { ref installedProcessor : Processor };
class IPhone5 : CellphoneModel specializes Cellphone {
ref compatibleProcessor = A6
};
```

Regularity Features

- Allowed Types
  - The regularity feature determines the possible types of entities that can be assigned to the regulated one

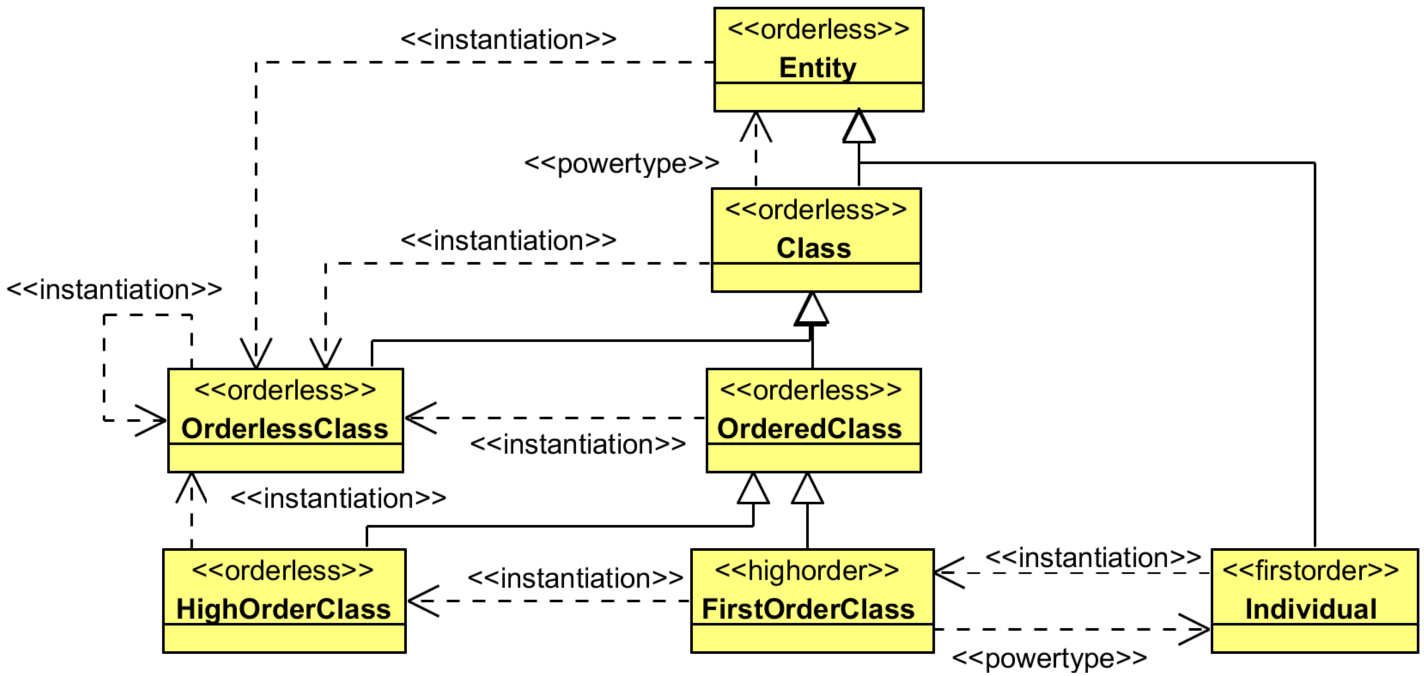
```
class CellphoneCharger;
order 2 class CellphoneChargerModel;
class UKCellphoneCharger : CellphoneChargerModel specializes CellphoneCharger;
class USACellphoneCharger : CellphoneChargerModel specializes CellphoneCharger;
class Cellphone { ref bundledCharger : CellphoneCharger };
order 2 class CellphoneModel categorizes Cellphone {
    regularity ref availableChargerModels : [0..*] CellphoneChargerModel
        determinesAllowedTypes bundledCharger
};
class IPhone5 : CellphoneModel specializes Cellphone {
    ref availableChargerModels = { UKCellphoneCharger, USACellphoneCharger}
};
individual Charger321 : UKCellphoneCharger;
individual Device321 : IPhone5 { ref bundledCharger=Charger321 };
```

Example Model

- With ML2 when are able to build very general conceptualization
- A quick example model in ML2, is the conceptualization of it’s own foundation theory, MLT*
- We can describe the theory constant as elements in a ML2 model

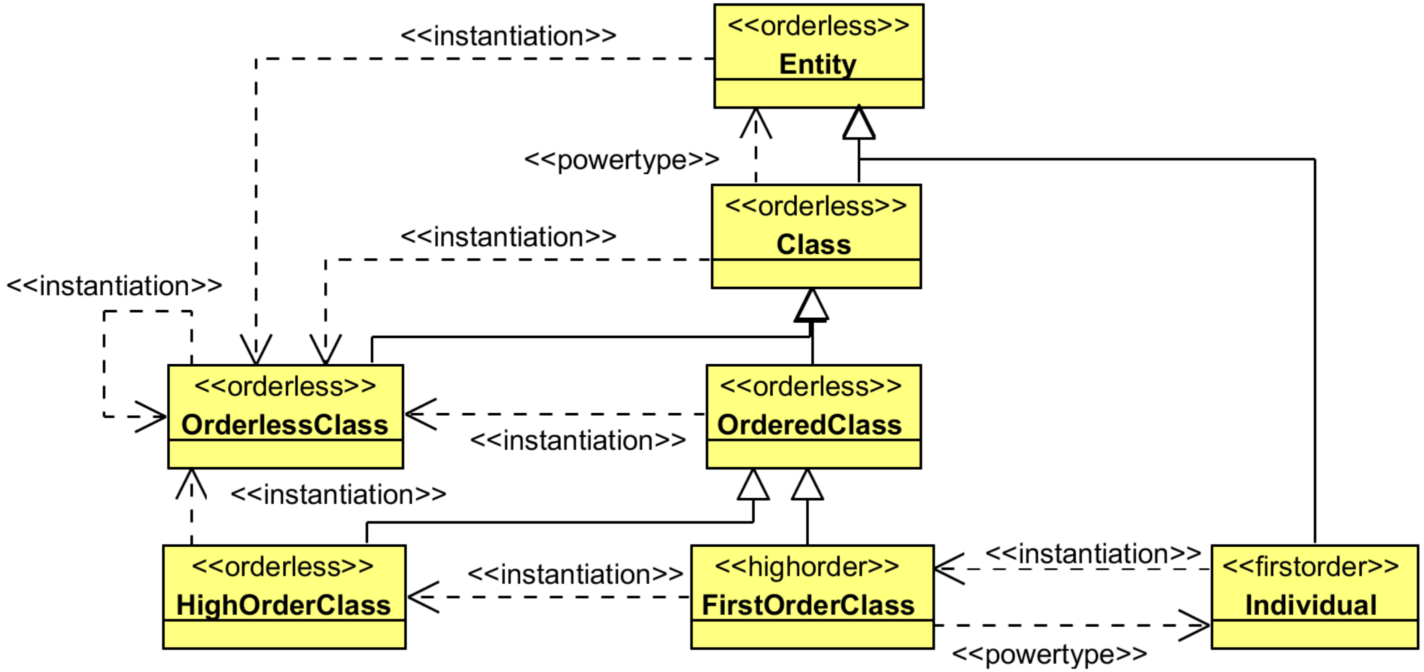
- In the next slides we are going to use an UML-based representation of the models in order to improve the presentation, however, the definition of a visual syntax for ML2 is still topic of an future research

Example Model



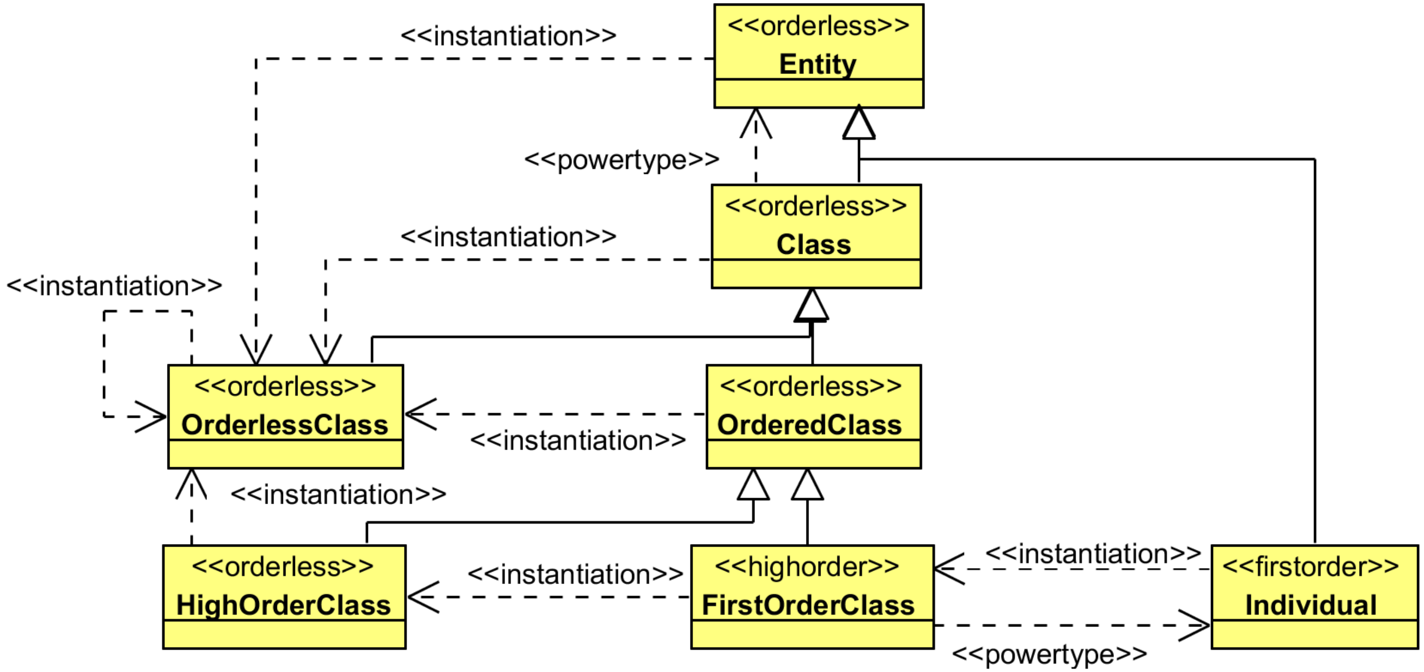

```
orderless class Entity : OrderlessClass;
orderless class Class : OrderlessClass specializes Entity isPowertypeOf Entity;
class Individual : FirstOrderClass specializes Entity;
disjoint complete genset has_instances
    general Entity
    specifics Class, Individual;
```

Example Model




```
orderless class OrderlessClass : OrderlessClass specializes Class;
orderless class OrderedClass : OrderlessClass specializes Class;
disjoint complete genset fixed_order
    general Class
    specifics OrderedClass, OrderlessClass;
```

Example Model

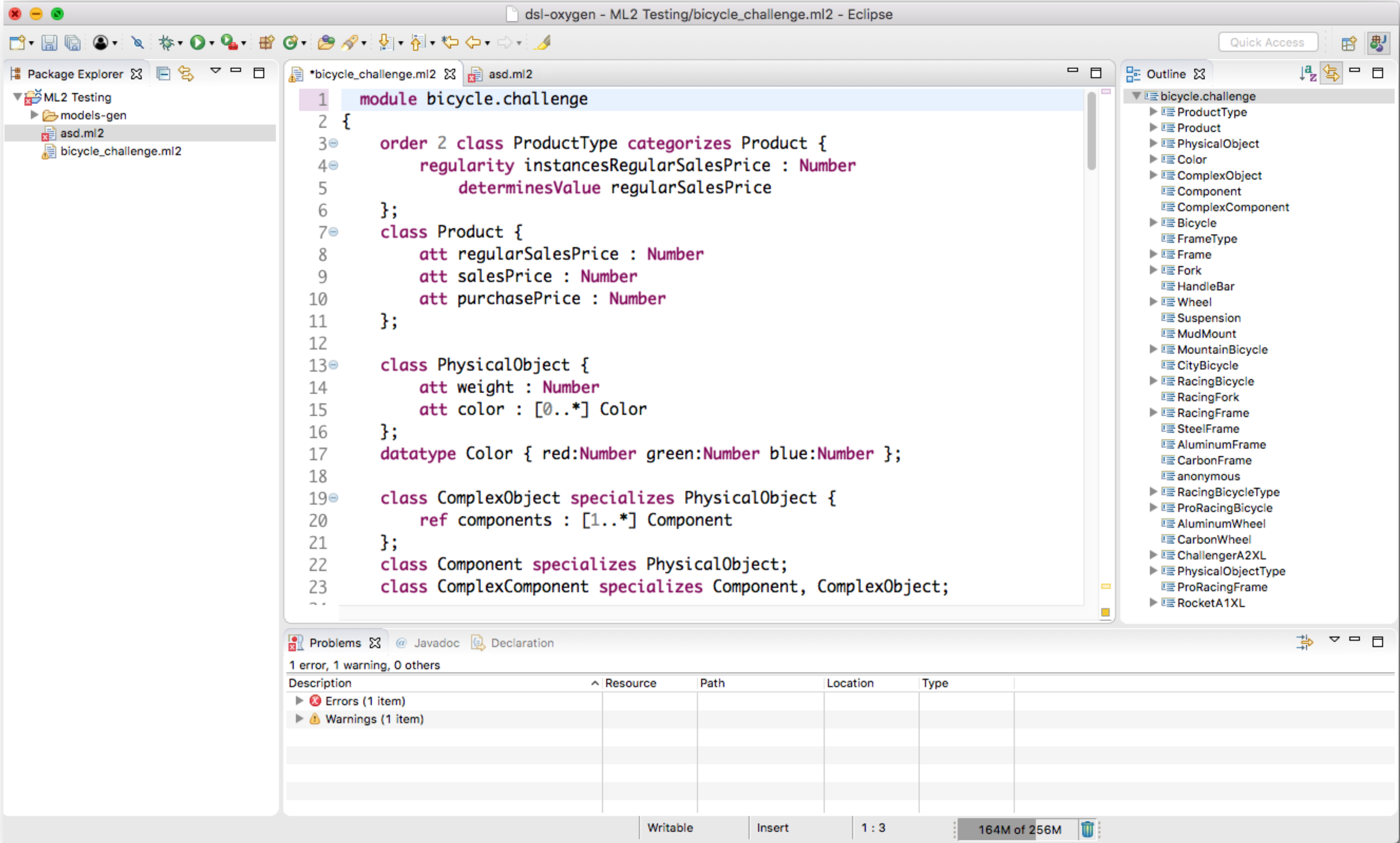



```
order 2 class FirstOrderClass : HighOrderClass specializes OrderedClass
    isPowertypeOf Individual;
orderless class HighOrderClass : OrderlessClass specializes OrderedClass;
disjoint complete genset high_order
    general OrderedClass
    specifics FirstOrderClass, HighOrderClass;
```

ML2 Editor

- The ML2 Editor is an Eclipse-based IDE for the development of ML2 models
- Built with the Xtext framework
- Provides the basic features of an traditional IDE for an conceptual modeling language
- Validation of semantically-motivated syntactical rules

ML2 Editor


dsl-oxygen - ML2 Testing/bicycle_challenge.ml2 - Eclipse
Quick Access
Package Explorer
ML2 Testing
models-gen
asd.ml2
bicycle_challenge.ml2
*bicycle_challenge.ml2
asd.ml2
module bicycle.challenge
{
order 2 class ProductType categorizes Product {
regularity instancesRegularSalesPrice : Number
determinesValue regularSalesPrice
};
class Product {
att regularSalesPrice : Number
att salesPrice : Number
att purchasePrice : Number
};
class PhysicalObject {
att weight : Number
att color : [0..*] Color
};
datatype Color { red:Number green:Number blue:Number };
class ComplexObject specializes PhysicalObject {
ref components : [1..*] Component
};
class Component specializes PhysicalObject;
class ComplexComponent specializes Component, ComplexObject;
Outline
bicycle.challenge
ProductType
Product
PhysicalObject
Color
ComplexObject
Component
ComplexComponent
Bicycle
FrameType
Frame
Fork
HandleBar
Wheel
Suspension
MudMount
MountainBicycle
CityBicycle
RacingBicycle
RacingFork
RacingFrame
SteelFrame
AluminumFrame
CarbonFrame
anonymous
RacingBicycleType
ProRacingBicycle
AluminumWheel
CarbonWheel
ChallengerA2XL
PhysicalObjectType
ProRacingFrame
RocketA1XL
Problems
Javadoc
Declaration
1 error, 1 warning, 0 others
Description
Resource
Path
Location
Type
Errors (1 item)
Warnings (1 item)
Writable
Insert
1 : 3
164M of 256M

ML2 Editor

- Syntax coloring
- Hover information and in-code documentation
- Error checking

```
order 2 class A isPowertypeOf B;
class B;
class C : A;
```

Missing specialization of B, base type of A.

```
order 2 class A;
orderless class B specializes A;
```

Invalid specialization of A.

```
order 2 class A subordinatedTo B;
order 2 class B subordinatedTo A;
```

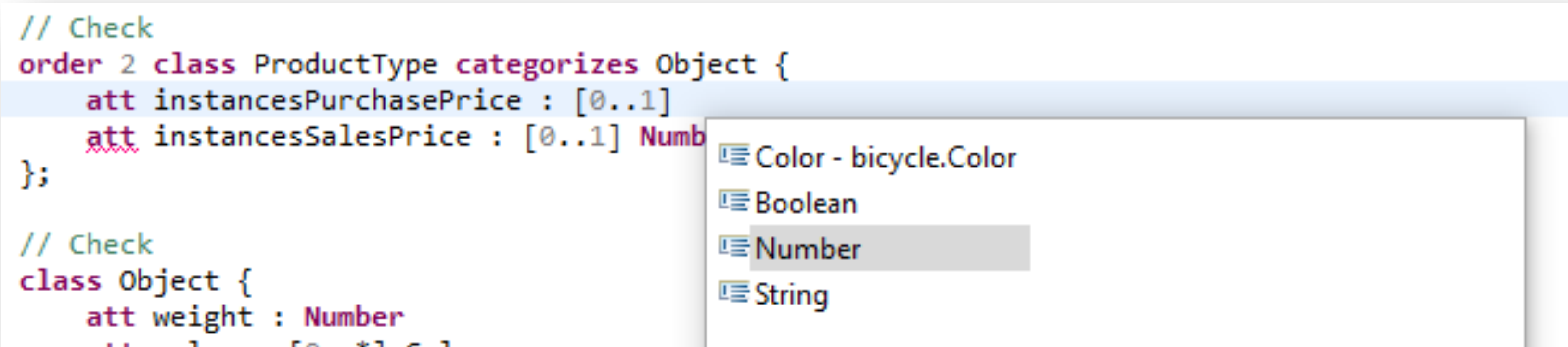
B is in a invalid subordination cycle with A.

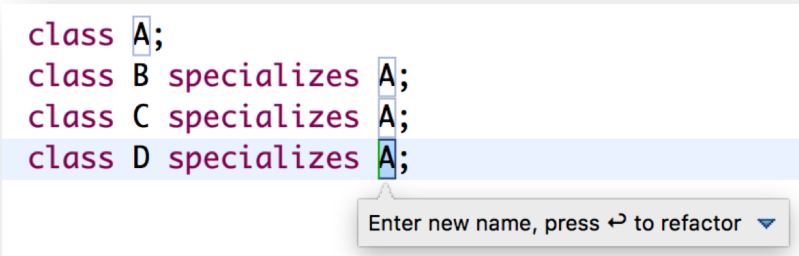
```
class A;
class B : A;
```

Invalid instantiation of A

ML2 Editor

- Auto-completion
- Go to declaration
- Rename refactoring
- Find references





ML2 Editor

- The table bellow presents some of the syntax rules checked by the ML2 Editor
- These rules are lively checked

Type	Syntactic Rules
Class	Specializations can only occurs between entities of same order or orderless classes.
Class	Ordered classes can only be powertype of classes in the order immediately below.
Class	Classes cannot be in subordination cycles.
Class	An instance of a subordinated class must specialize some instance of the related subordinator class.
GeneralizationSet	The categorizer class must categorize the general class.
Feature	Regularity types of "maximum value" and "minimum value" applies only to number attributes.
Feature	Regularity types of "determined types" and "allowed type" applies only to references.
Feature	A regulated feature assignment must conform to the regularity feature assignment.
FeatureAssignment	A feature assignment must conform to the multiplicity and type of its associated feature.

Installing the ML2 Editor

- Go to https://github.com/claudenirmf/ML2-Editor
- Download the compressed file in the release and extract it in your computer
- On an instance of the Eclipse IDE, go to *Help* > *Install New Software…*
  - We suggest you to use the *Eclipse IDE for Java and DSL Developers* since it offers the minimum set of tools required for the ML2 Editor
- Click on *Add*, enter the path to the folder you extracted to your computer (i.e. *../repository*) and click on *Ok*

Installing the ML2 Editor

- The ML2 plugin for Eclipse should appear on the list of available software now. Select it and proceed its installation
- In the end, you will required to restart your Eclipse in order to activate the ML2 plugin

Creating a Project

- On your Eclipse, create a *General Project*
- Within this project you should create your ".ml2" files, and the editor you consider the models in that project sharing a common context
- When the first ".ml2" file is opened, a message will appear on the screen asking to active the Xtext capabilities in the project. Please select "Yes".
- Now you can write your ML2 models and reference entities between the different models

Questions and Answers

MLT

MULTI-LEVEL

MODELING THEORY

ML2

MULTI-LEVEL

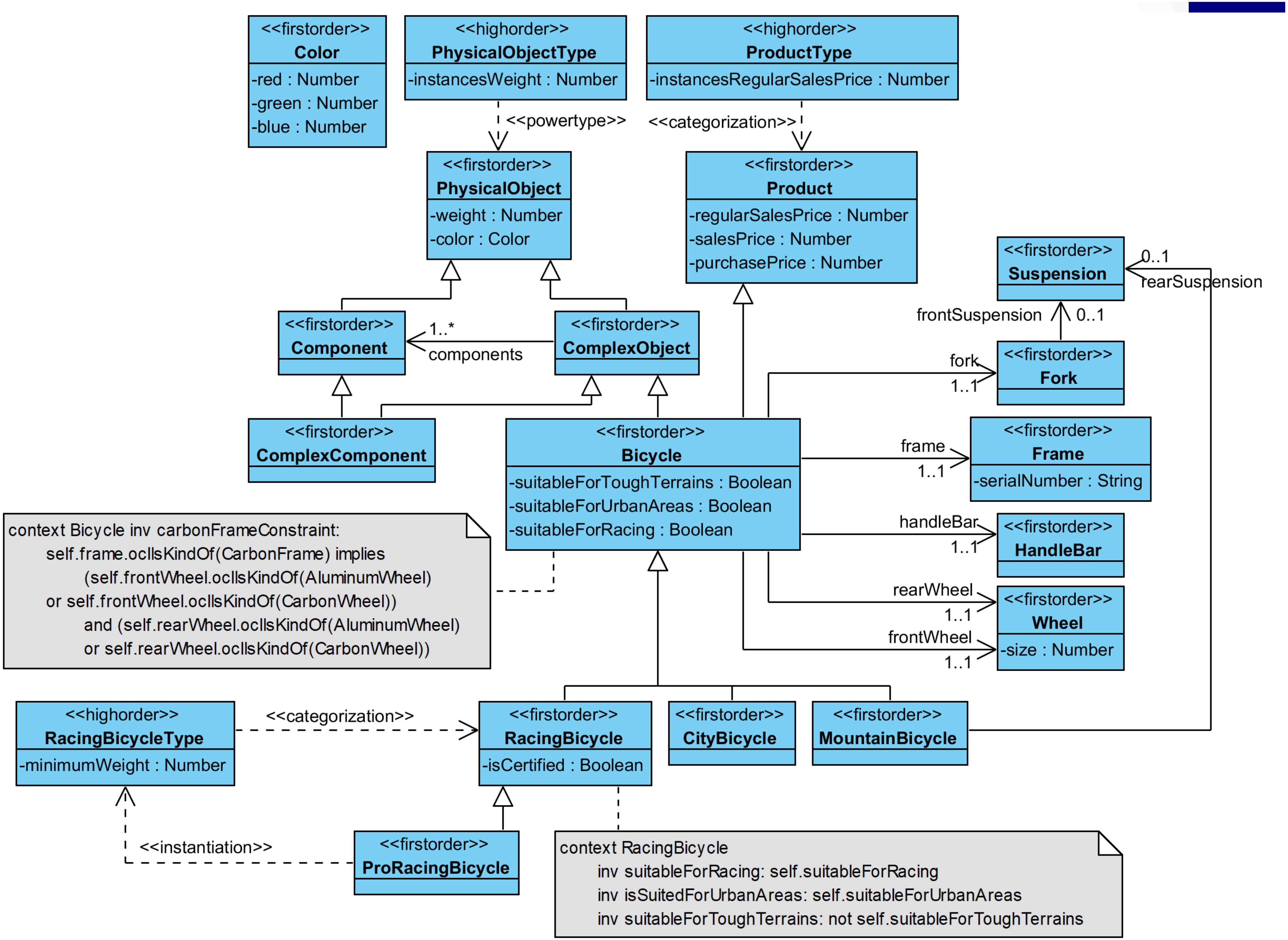
MODELING LANGUAGE

Thank you!

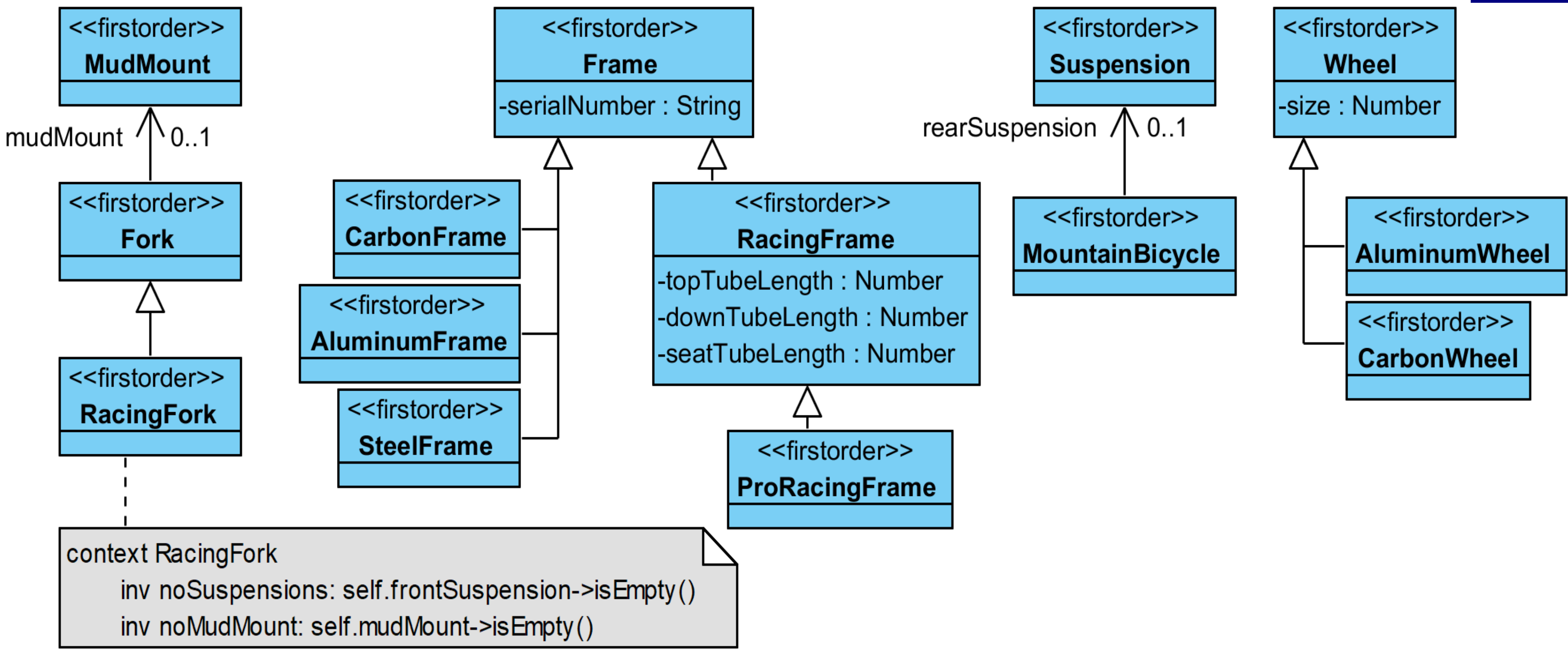
unibz KRDB nemo UFES UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO INSTITUTO FEDERAL Espírito Santo

See also

- M.Sc. Thesis Claudenir Fonseca (includes an ML2 model for the bicycle challenge of MULTI 2017)



<<firstorder>>
Color
-red : Number
-green : Number
-blue : Number
<<highorder>>
PhysicalObjectType
-instancesWeight : Number
<<highorder>>
ProductType
-instancesRegularSalesPrice : Number
<<powertype>>
<<categorization>>
<<firstorder>>
PhysicalObject
-weight : Number
-color : Color
<<firstorder>>
Product
-regularSalesPrice : Number
-salesPrice : Number
-purchasePrice : Number
<<firstorder>>
Suspension
0..1
rearSuspension
frontSuspension
0..1
<<firstorder>>
Component
1..*
components
<<firstorder>>
ComplexObject
fork
1..1
<<firstorder>>
Fork
<<firstorder>>
ComplexComponent
<<firstorder>>
Bicycle
-suitableForToughTerrains : Boolean
-suitableForUrbanAreas : Boolean
-suitableForRacing : Boolean
frame
1..1
<<firstorder>>
Frame
-serialNumber : String
handleBar
1..1
<<firstorder>>
HandleBar
rearWheel
1..1
<<firstorder>>
Wheel
-size : Number
frontWheel
1..1
context Bicycle inv carbonFrameConstraint:
self.frame.oclIsKindOf(CarbonFrame) implies
(self.frontWheel.oclIsKindOf(AluminumWheel)
or self.frontWheel.oclIsKindOf(CarbonWheel))
and (self.rearWheel.oclIsKindOf(AluminumWheel)
or self.rearWheel.oclIsKindOf(CarbonWheel))
<<highorder>>
RacingBicycleType
-minimumWeight : Number
<<categorization>>
<<firstorder>>
RacingBicycle
-isCertified : Boolean
<<firstorder>>
CityBicycle
<<firstorder>>
MountainBicycle
<<instantiation>>
<<firstorder>>
ProRacingBicycle
context RacingBicycle
inv suitableForRacing: self.suitableForRacing
inv isSuitedForUrbanAreas: self.suitableForUrbanAreas
inv suitableForToughTerrains: not self.suitableForToughTerrains



<<firstorder>>
MudMount
mudMount
0..1
<<firstorder>>
Fork
<<firstorder>>
RacingFork
<<firstorder>>
Frame
-serialNumber : String
<<firstorder>>
CarbonFrame
<<firstorder>>
AluminumFrame
<<firstorder>>
SteelFrame
<<firstorder>>
RacingFrame
-topTubeLength : Number
-downTubeLength : Number
-seatTubeLength : Number
<<firstorder>>
ProRacingFrame
<<firstorder>>
Suspension
rearSuspension
0..1
<<firstorder>>
MountainBicycle
<<firstorder>>
Wheel
-size : Number
<<firstorder>>
AluminumWheel
<<firstorder>>
CarbonWheel
context RacingFork
inv noSuspensions: self.frontSuspension->isEmpty()
inv noMudMount: self.mudMount->isEmpty()

```
class PhysicalObject { att weight : Number };

class ComplexObject specializes PhysicalObject {
ref components : [1..*] Component };

class Component specializes PhysicalObject;
class ComplexComponent specializes Component, ComplexObject;

class Bicycle specializes ComplexObject {
          ref frame : Frame subsets components
          ref fork : Fork subsets components
          ref handleBar : HandleBar subsets components
          ref frontWheel : Wheel subsets components
          ref rearWheel : Wheel subsets components
};

class Frame specializes Component;
class Fork specializes ComplexComponent;
class HandleBar specializes Component;
class Wheel specializes Component;
class Suspension specializes Component;
class MudMount specializes Component;
```

```
class PhysicalObject {
            att weight : Number
            att color : [0..*] Color
};
datatype Color { red:Number green:Number blue:Number };

order 2 class ProductType categorizes Product {
            regularity instancesRegularSalesPrice : Number
determinesValue regularSalesPrice
};
class Product {
            att regularSalesPrice : Number
            att salesPrice : Number
            att purchasePrice : Number
};

class Bicycle specializes PhysicalObject, ComplexObject, Product {
ref frame : Frame subsets components
            ref fork : Fork subsets components
            ref handleBar : HandleBar subsets components
            ref frontWheel : Wheel subsets components
            ref rearWheel : Wheel subsets components
};
class Frame specializes Component, Product {
            att serialNumber : String
};
```

```
class Bicycle specializes PhysicalObject, ComplexObject, Product {
        att suitableForToughTerrains : Boolean
        att suitableForUrbanAreas : Boolean
        att suitableForRacing : Boolean
};

class CityBicycle specializes Bicycle;
class MountainBicycle specializes Bicycle {
        ref rearSuspension : [0..1] Suspension subsets components
};
class RacingBicycle : RacingBicycleType specializes Bicycle;
```

```
class RacingBicycle specializes Bicycle { att isCertified : Boolean };

class RacingFrame specializes Frame {
        att topTubeLength : Number
        att downTubeLength : Number
        att seatTubeLength : Number
};

class SteelFrame specializes Frame;
class AluminumFrame specializes Frame;
class CarbonFrame specializes Frame;

disjoint genset
        general Frame
        specifics SteelFrame, AluminumFrame, CarbonFrame;
```

```
order 2 class RacingBicycleType categorizes RacingBicycle {
         regularity minimumWeight : Number determinesMinValue weight
         regularity ref allowedFrameTypes : [0..*] FrameType
                  determinesAllowedTypes frame
};

class ProRacingBicycle :RacingBicycleType specializes RacingBicycle {
         att minimumWeight  = 5.200
         ref allowedFrameTypes =  {AluminumFrame, CarbonFrame}
};

class AluminumWheel specializes Wheel;
class CarbonWheel specializes Wheel;
```

```
class ChallengerA2XL :RacingBicycleType, ProductType specializes ProRacingBicycle {
        att instancesRegularSalesPrice = 4999.00
        ref frame : RocketA1XL subsets frame
};

order 2 class PhysicalObjectType isPowertypeOf PhysicalObject {
        att instancesWeight : [0..1] Number
};

class ProRacingFrame specializes RacingFrame;
class RocketA1XL :ProductType specializes ProRacingFrame {
        att instancesWeight = 0.920
};
```