

# INTERFACES and DATA MODELING in MELODY ADVANCE

<b>1 Document purpose and origins .....</b>	<b>4</b>
1.1 Document purpose.....	4
1.2 Change record .....	4
<b>2 Introduction .....</b>	<b>5</b>
2.1 Why Data Modeling?.....	5
2.2 Main Concepts Overview .....	5
<b>3 Basic Constructs : data model .....</b>	<b>8</b>
3.1 Overview .....	8
3.2 Simple Type .....	8
3.2.1 NumericType .....	9
3.2.2 PhysicalQuantity and Unit.....	10
3.2.3 Enumeration .....	13
3.2.4 BooleanType.....	13
3.2.5 StringType .....	14
3.3 Complex Type.....	15
3.4 Class .....	17
3.4.1 Class Editor .....	18
3.4.2 Property .....	19
3.4.3 Key.....	19
3.4.4 Cardinality.....	20
3.4.5 Other Property basic features.....	21
3.4.5.1 Default Value .....	22
3.4.5.2 Null Value .....	22
3.4.5.3 Min/Max .....	22
3.4.5.4 Min. Length/Max. Length .....	22
3.4.6 Property ordering .....	23
3.4.7 Operation .....	23
3.4.8 Abstract Class.....	23
3.4.9 isPrimitive .....	25
3.5 Values .....	25
3.5.1 Literal Numeric Value .....	25
3.5.2 Enumeration Literal.....	26
3.5.3 Literal Boolean Value.....	27
3.5.4 Literal String Value .....	28
3.6 Association.....	29
3.6.1 Role .....	29
3.6.2 Cardinality.....	30
3.6.3 Navigability .....	32
3.6.4 Name .....	33
3.6.5 Associations in Melody Advance Project Explorer.....	35
3.6.6 Aggregation, composition .....	36
3.7 Generalization.....	38
3.8 Package .....	39
3.8.1 Definition.....	39
3.8.2 Dependencies .....	40
3.9 CDB (Class Diagram Blank) .....	42
3.9.1 Filters .....	42
3.9.1.1 Hide Properties.....	43
3.9.1.2 Hide Operations.....	43

3.9.1.3	Hide Associations .....	43
3.9.1.4	Hide Generalizations .....	44
3.9.1.5	Show Full Path .....	44
3.9.1.6	Hide Derived Properties .....	45
3.9.2	Toolbox .....	45
3.9.2.1	Insert / Remove Types .....	45
<b>4</b>	<b>Basic Constructs : communication model .....</b>	<b>47</b>
4.1	Overview of Main Communication Concepts .....	47
4.2	Exchange Item .....	48
4.2.1	Event.....	50
4.2.2	Operation .....	51
4.2.3	Flow .....	52
4.2.4	Shared Data.....	52
4.3	Exchange Item Element.....	54
4.4	Interface .....	59
4.4.1	Interface Definition Blank (IDB) diagram .....	60
4.4.2	CEI and CDI diagrams.....	60
4.4.3	Interface Scenario (Vs. Functional Scenario and Exchange Scenario) .....	61
<b>5</b>	<b>Advanced Constructs .....</b>	<b>68</b>
5.1	Class and Type advanced features .....	68
5.1.1	isDiscrete .....	68
5.1.2	isDerived .....	69
5.1.3	isOrdered .....	70
5.1.4	isUnique.....	71
5.1.5	Is Final .....	72
5.1.6	Is Read Only .....	72
5.1.7	Pattern .....	73
5.1.8	Visibility.....	74
5.1.9	Super .....	74
5.2	Expression .....	76
5.2.1	Unary Expression .....	76
5.2.2	Binary expression .....	77
5.3	Collection .....	79
5.4	Union / Variant .....	81
5.5	Values and References.....	83
5.5.1	Simple Values .....	83
5.5.2	Complex Values.....	83
5.5.3	Collection Values .....	83
5.5.4	References .....	83
5.6	Modeling Accelerators .....	84
5.6.1	Exchange Items Propagation to Function Ports .....	84
5.6.2	Interface Generation from Allocated Functions .....	85
<b>6</b>	<b>Methodological Recommendations.....</b>	<b>87</b>
6.1	Data Model and Arcadia Abstraction Levels .....	87
6.2	Data Normalization .....	88
6.2.1	1NF .....	88
6.2.2	2NF .....	88
6.2.3	3NF .....	88
6.3	Validating the interface and data model.....	89
6.4	Basic Best Practices .....	91
6.4.1	Creating new model elements .....	91
6.4.2	Naming Conventions .....	91
6.4.3	Package Structuration .....	92
6.4.4	Diagram Layout .....	92
6.4.5	Using color codes on diagrams .....	92
6.4.6	Document and annotate the model.....	92

<b>7 Appendix A – Car data model example.....</b>	<b>93</b>
7.1 Package Dependencies .....	93
7.2 Domain Concepts .....	93
7.3 Complex and Simple Types .....	94

## 1 DOCUMENT PURPOSE AND ORIGINS

### 1.1 Document purpose

The objective of this document is to provide a clear status on how interface and data modeling can be supported by Melody Advance.

It details the concepts available in Melody Advance and also proposes some methodological guidance to be taken into account while performing data modeling.

It has been initially developed in collaboration with several Thales entities : TAS, TCS and TGS.

### 1.2 Change record

DATE	Version	Author	Change record
21/11/2014	V0	Pascal Roques (PRFC)  <b>Major contributors :</b> Nathalie Millet (TAS) Michel Raoux (TCS) Stéphanie Cheutin (TAS) Pierre-Marie Perchoc (TCS)	Initial Version  MA 2.5.4 and 3.0

## 2 INTRODUCTION

### 2.1 Why Data Modeling?

A significant part of system engineering consists in ensuring proper definition and coherency between data & information inside the system, and those exchanged with external actors (including interface & I/O management).

In order to describe unambiguously what is exchanged between functions, activities, components, and external actors, a specific formalization of data, information, material flow... used in the system is usually performed.

Beyond this description need, one important engineering task consists in avoiding multiple definitions of a same data in different places of the system. Hence the need to indicate that several exchanges should carry the same kind of data, without having to re-define these common data for each exchange.

Rationalizing and mastering data definition and use is a major stake of engineering;

- to structure them in an intelligible manner, reducing complexity of their definition and use,
- to bring out a semantics coherency, independently from their use – or common to all their uses,
- to avoid ambiguity, redundancy, incoherency in their different uses.

The main benefits derived from this improved data definition are:

- better consistency and completeness of component interfaces,
- detection of hidden dependencies between components (for instance when the same piece of data is produced or consumed by several components),
- global architecture optimization (to remove dependencies),
- less rework during IVVQ,
- easier impact analysis in the case of engineering evolutions.

An efficient means to ensure these properties is to formalize a 'data model' (also known as 'information model' in information processing dominant systems), describing the kind of data to be used by the system; at least, for each kind of data:

- data name and semantics,
- their definition, contents and properties (unit, value domain...),
- their relationships with other data ('is composed of', 'is kind of', 'uses', semantic reference ...).

### 2.2 Main Concepts Overview

Interfaces, Exchanges and Exchange Items are contracts specifying how components can interact with each other. Exchange Items specify the communications between components and Interfaces or Exchanges allow structuring these communications. Interfaces and/or Exchanges are defined by grouping (referencing) Exchange Items: they can share Exchange Items definitions.

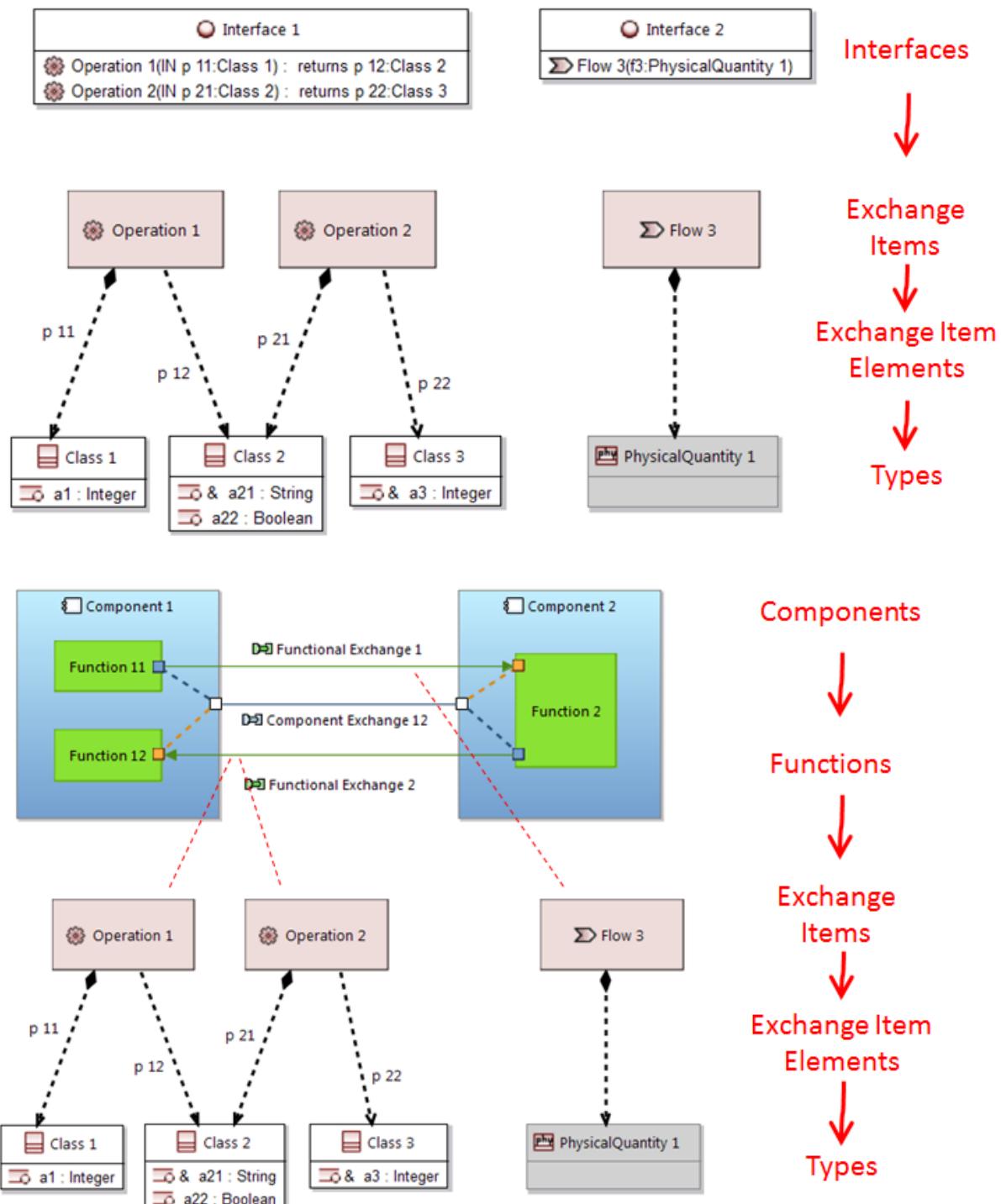
An Exchange Item defines a communication media and a set of Data semantically coherent with regards to their usage in a given context:

- same communication principles

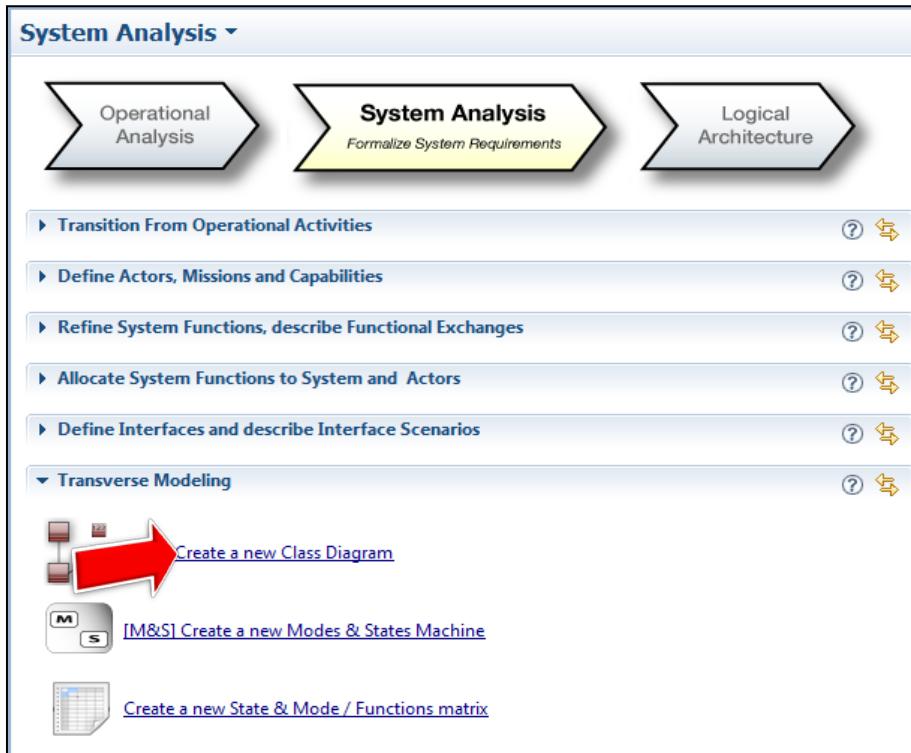
- simultaneity of transportation
- same non functional properties (e.g. security level, integrity requirement, expected performance...)
- indivisibility (an Exchange Item is atomic)

Possible communication mechanisms for Exchange Items are: OPERATION, EVENT, FLOW, and SHARED DATA.

Exchange Items are structured through Exchange Items Elements in the same way as classes are structured in Properties. These elements are in turn defined by classes, complex types and simple types.



Definitions of Interfaces, Exchange Items, Exchange Item Elements and Types, are mostly done in Melody Advance through Class Diagrams. These CDBs (Class Diagram Blank) are available at each Arcadia abstraction level, and can be found under the *Transverse Modeling* topic.

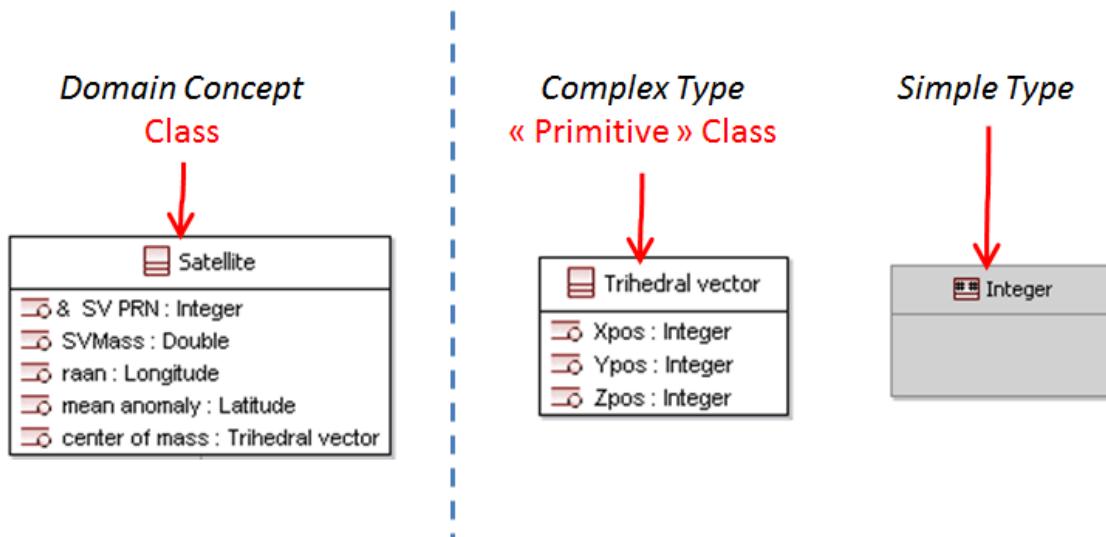


### 3 BASIC CONSTRUCTS : DATA MODEL

#### 3.1 Overview

There are three kinds of classifiers in Melody: classes (including unions and collections), complex types and simple types.

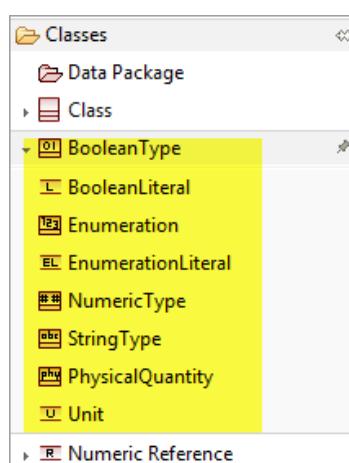
A data type is a special kind of classifier, similar to a class. It differs from a class in that instances of a data type are identified only by their value. All copies of an instance of a data type and any instances of that data type with the same value are considered to be equal instances. There are two kinds of data types: simple or [complex types](#).



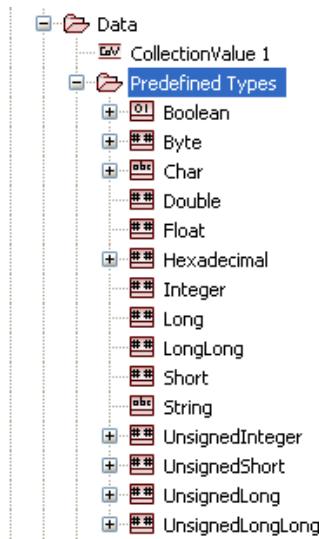
Note that Melody does not provide a way to distinguish primitive classes from non-primitive classes in class diagrams.

#### 3.2 Simple Type

Melody provides five different families of simple data types: [Boolean Type](#), [Enumeration](#), [Numeric Type](#), [String Type](#) and [Physical Quantity](#).



By default a new Melody project comes with a set of predefined simple data types. These predefined types can be completely redefined, removed or replaced. The best practice being to build-up a library from which all these data types would be reused. Libraries are available from Melody 3.0.



Melody provides advanced editors for specifying simple data types. For example, it is possible to define [min, max, default and null values](#), to specify whether a simple data type is discrete (countable) or not, etc.

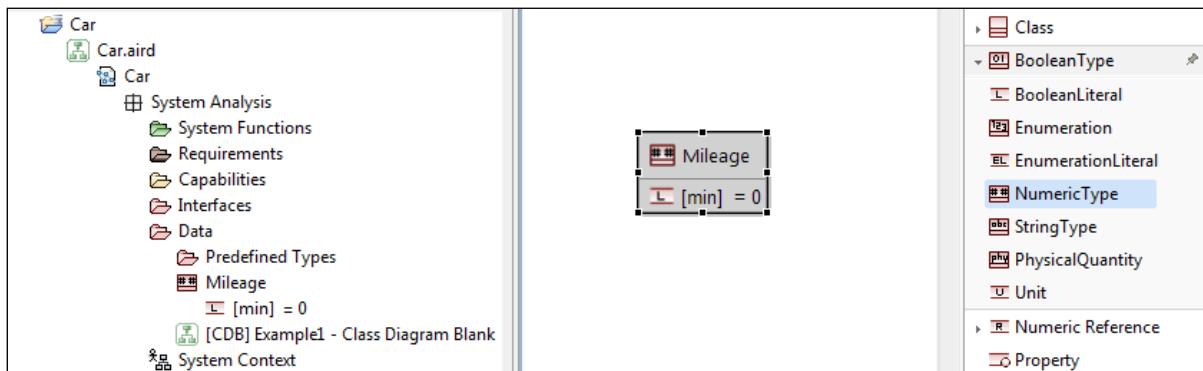
We will use the simple example of a car to further illustrate data modeling constructs.

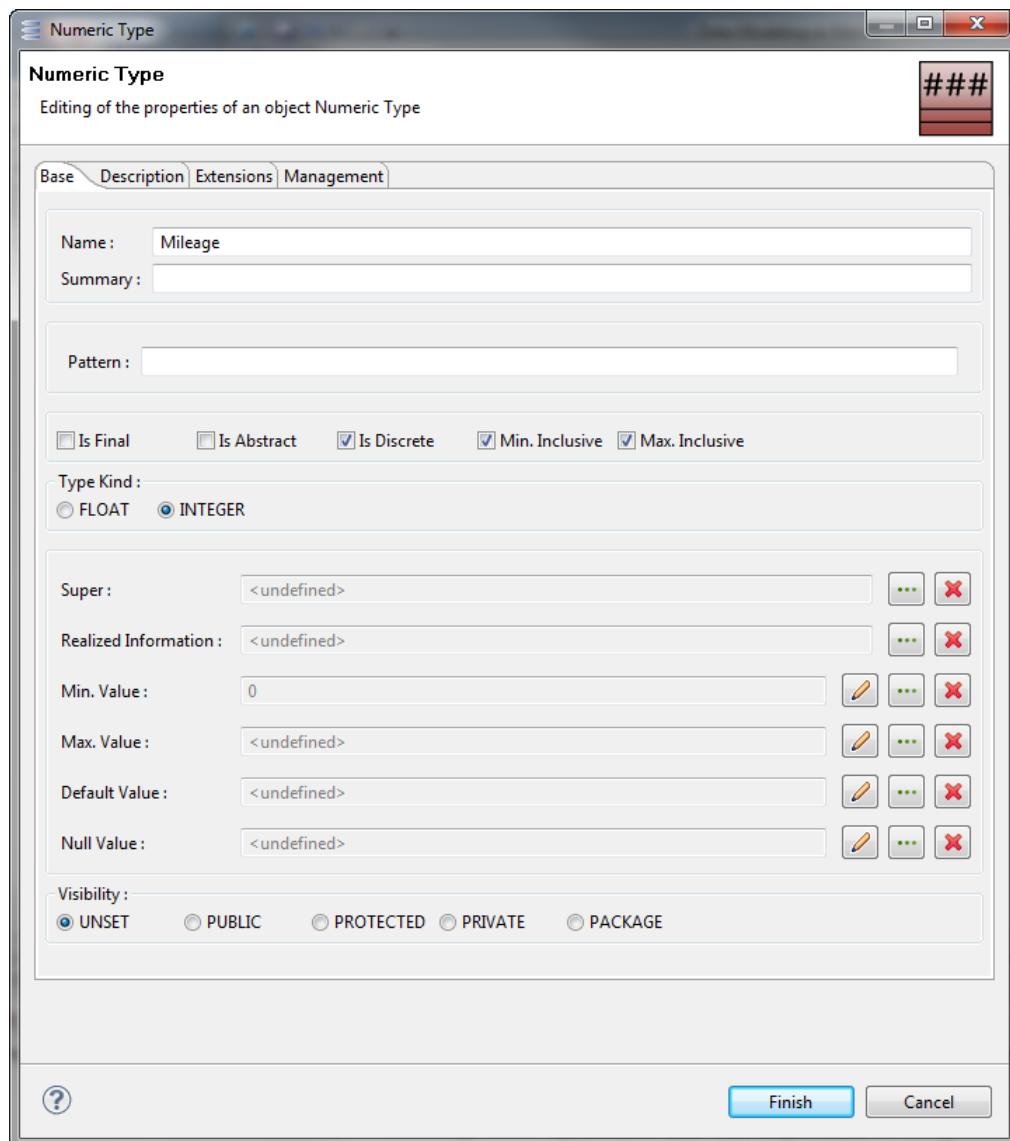
### 3.2.1 NumericType

Numeric Type is a kind of simple data type, representing numbers.

Numeric Types can be of Kind Float or Integer; their min and max Values define the range of the possible Values for the Numeric Type.

Let us create a first Numeric Type of Kind Integer called Mileage, for our car description.



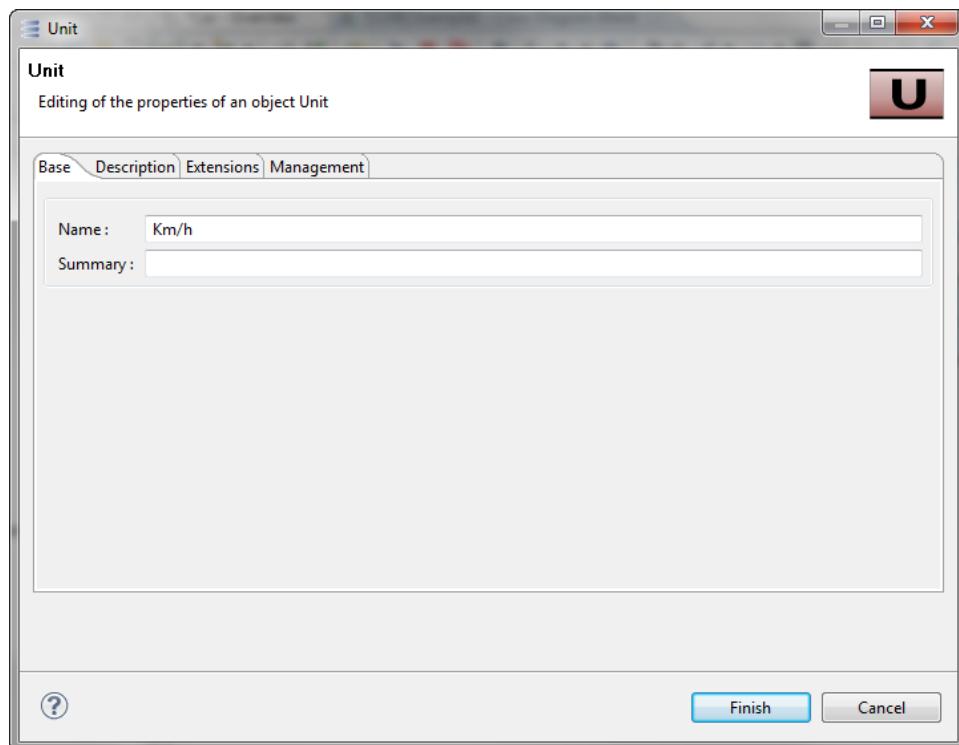


### 3.2.2 PhysicalQuantity and Unit

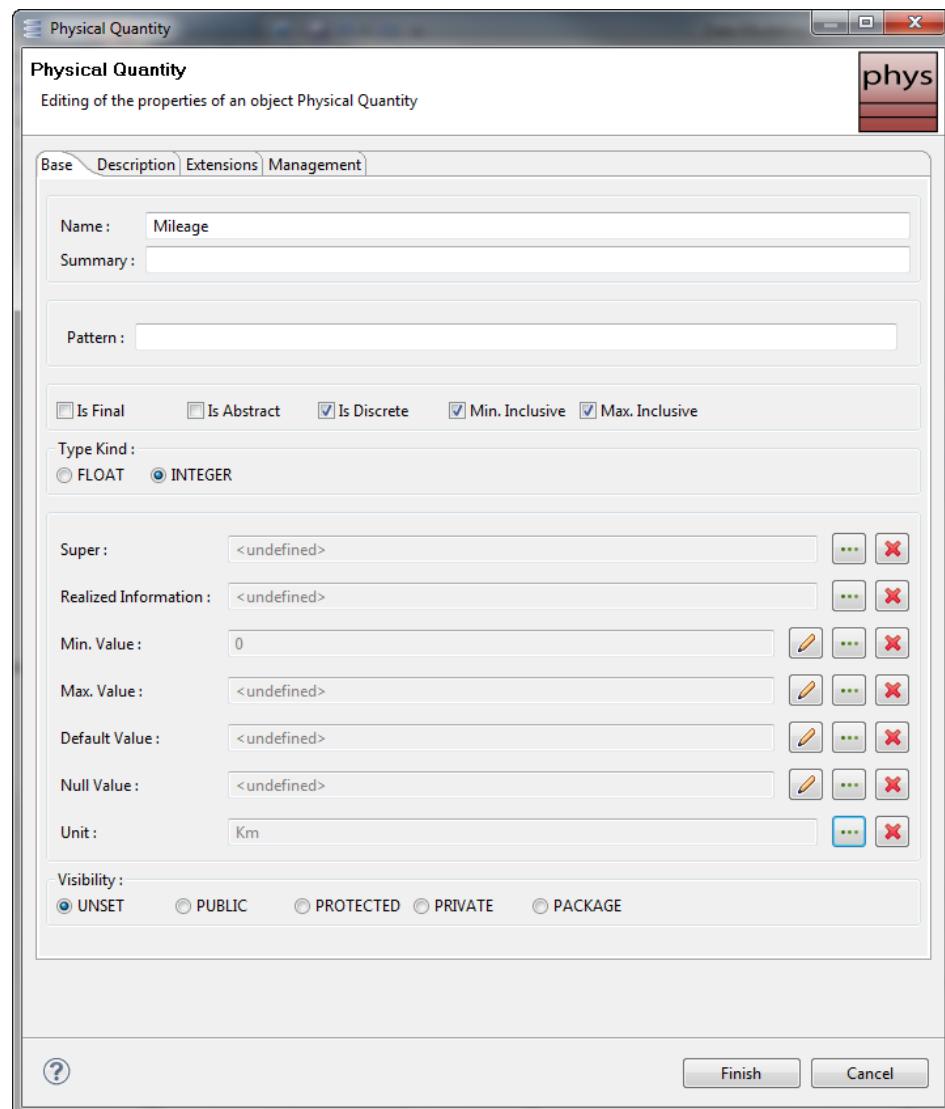
**Physical Quantity** is a specific kind of Numeric Type. It represents a physical “dimension” and is associated to Unit (meter, gram, volt, etc.).

A **Unit of measurement** is a definite magnitude of a physical quantity, defined and adopted by convention and/or by law, that is used as a standard for measurement of the same physical quantity. Units are used in Physical Quantity data types and [Numeric Values](#).

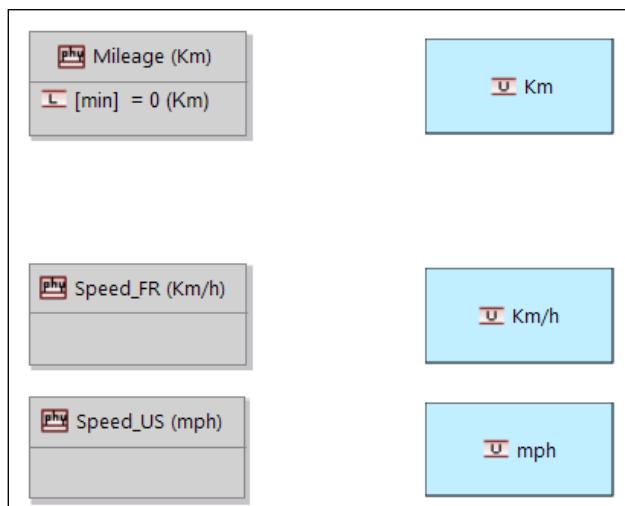
The speed of a car, for instance, can be expressed in terms of km/h (or mile/h, depending on the country).



The Mileage of a car is more precisely expressed in terms of kilometers or miles. If we wish to add this Unit concept, we need to use a Physical Quantity Type instead of a Numeric Type.



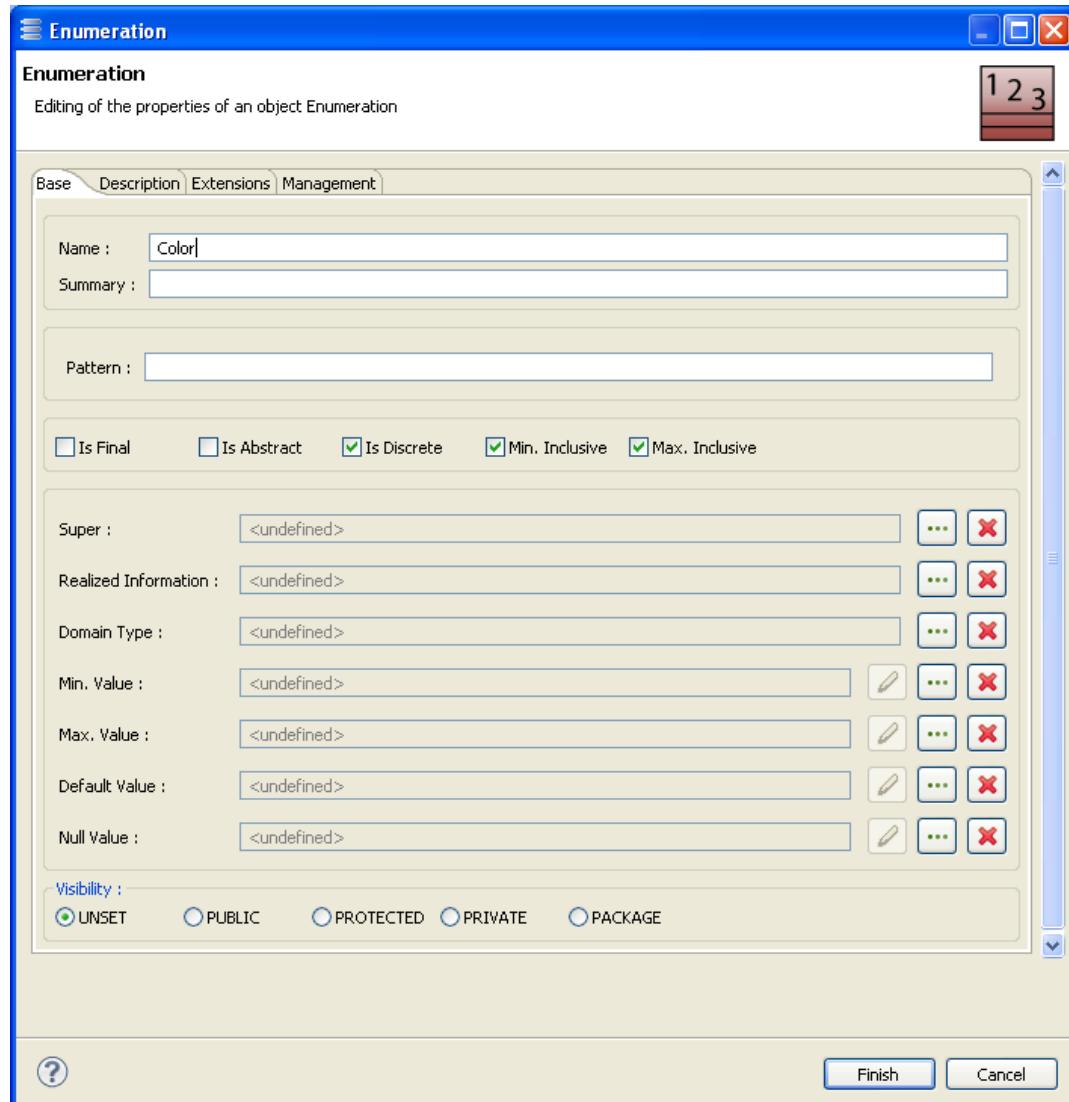
Speed can also be a Physical Quantity, with another Unit. We could even define two different Speed types, with different units...



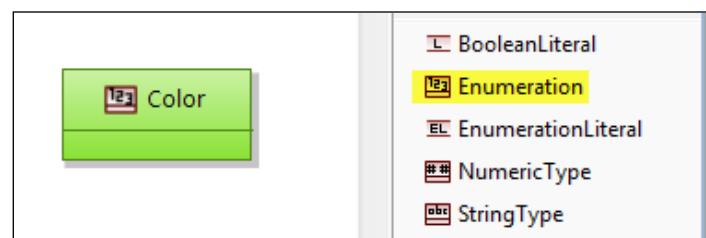
### 3.2.3 Enumeration

An Enumeration is a simple data type whose possible values are enumerated in the model as Enumeration Literals.

For instance, when you buy a car, you can choose its color in a predefined set.

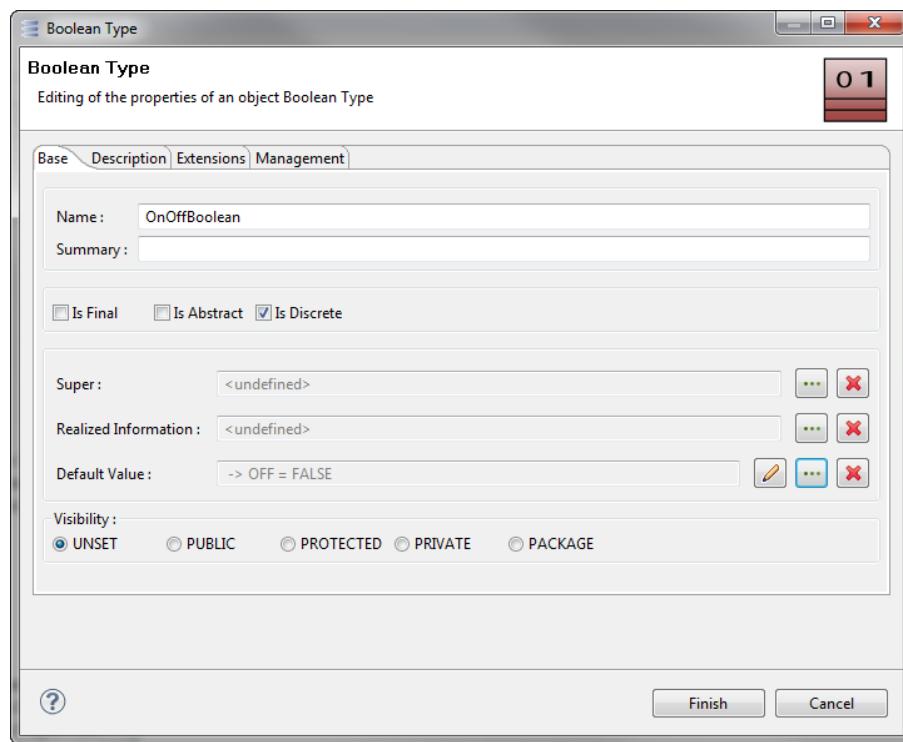


Note that the “Domain Type” Field is a new feature of version 3 (it does not appear in version 2.x).



### 3.2.4 BooleanType

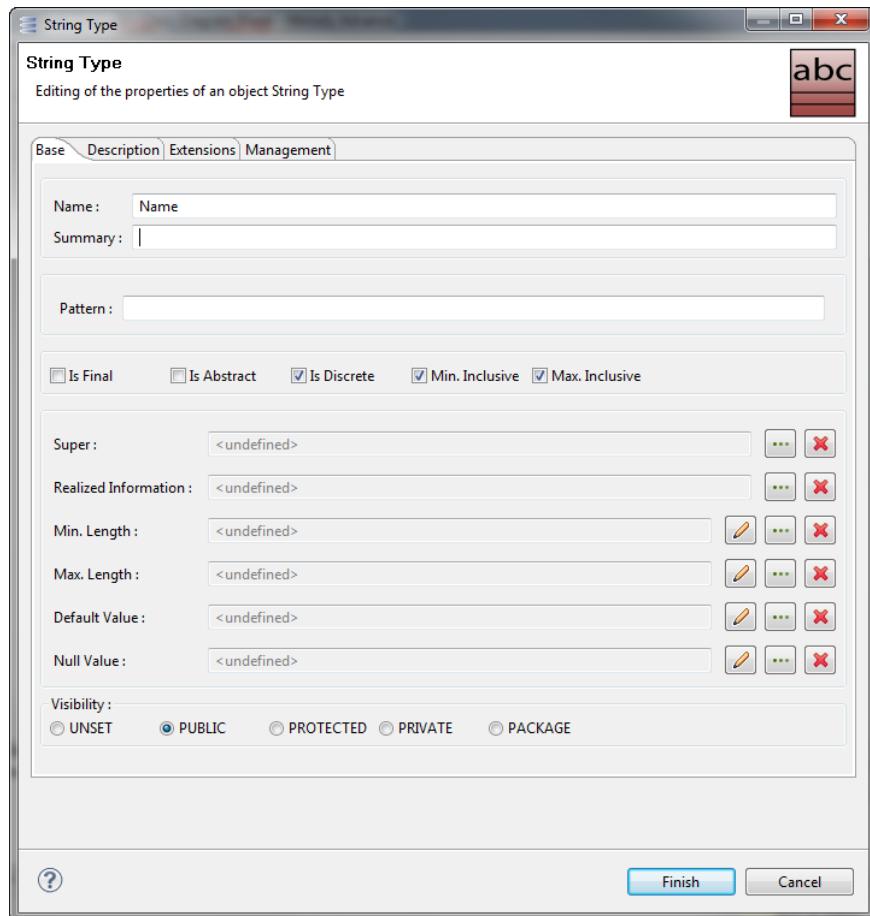
A Boolean Type is a kind of simple data type. A Boolean type defines two named values which hold the semantics of truth and falsehood.



### 3.2.5 StringType

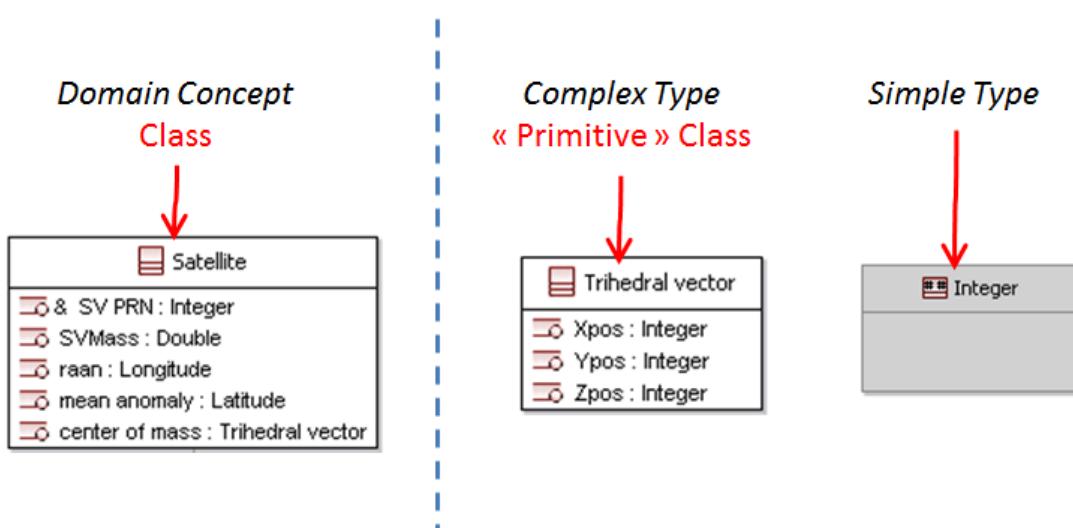
A String Type is a kind of simple data type, representing a sequence of characters. A String defines a piece of text. The semantics of the string itself depends on its purpose; it can be a comment, computational language expression, OCL expression, etc.

In our example, the owner of the car could be a person, with a first name and a last name. The first name and the last name are of kind String Type. Hence we define a specific String Type named "Name".



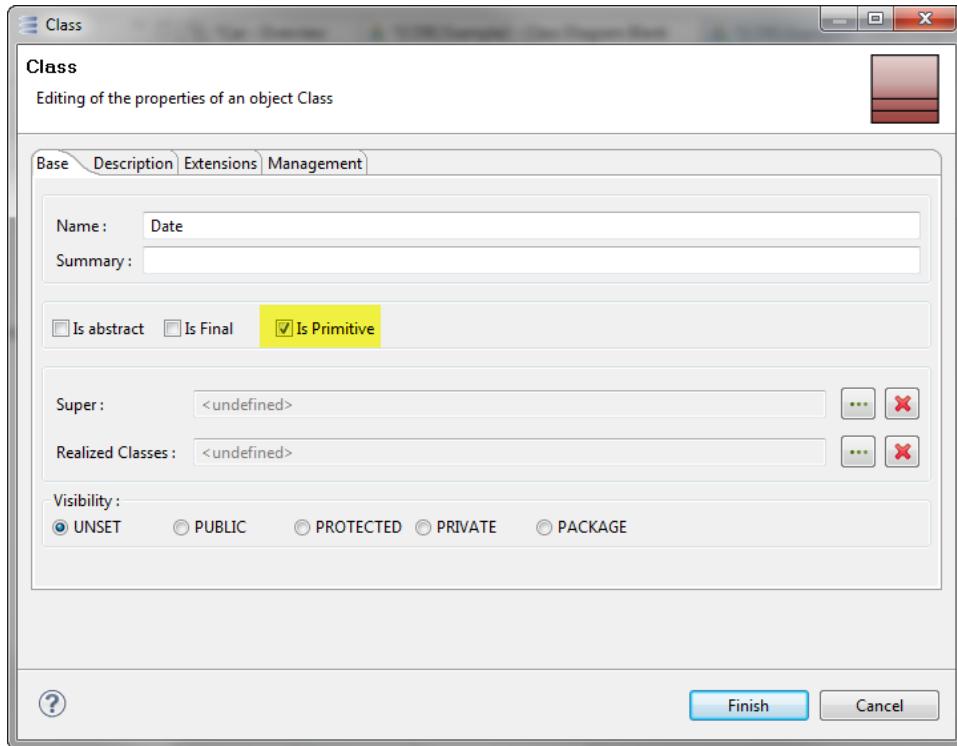
### 3.3 Complex Type

In Melody, simple types cannot own [Properties](#). So if we need structured data types, we have to create Classes, but then we have to specify that these classes are “[primitive](#)”. These primitive classes play the role of complex types.

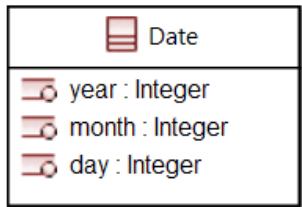


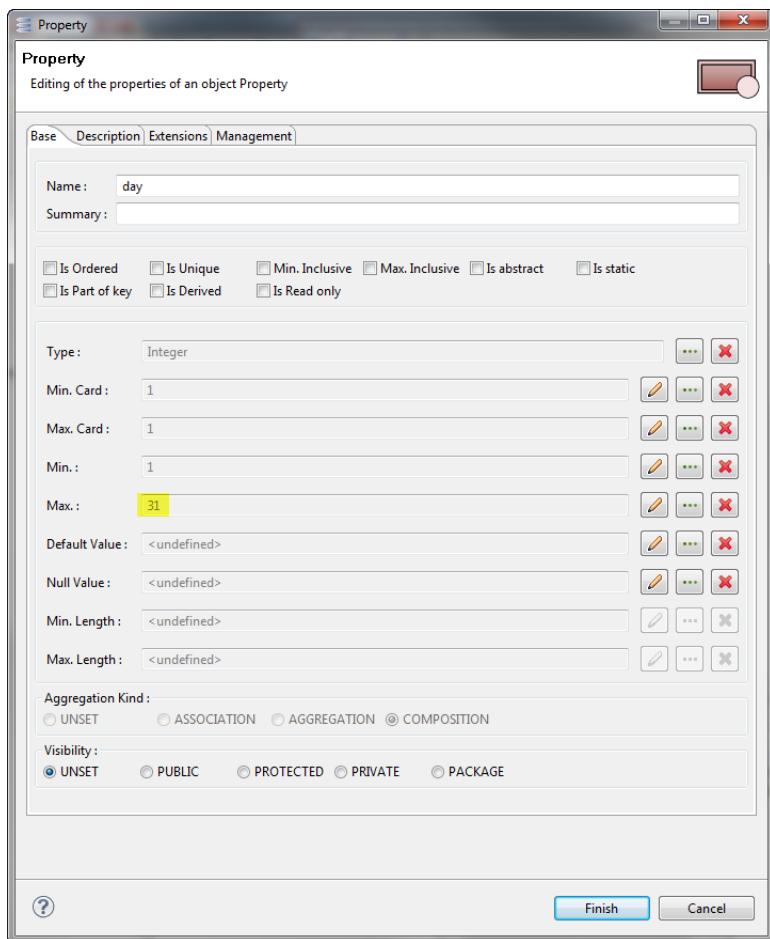
Primitive Classes can be used to type other Class Properties, but cannot be the source or target of [associations](#). Primitive means instances of the class have no identity in the context of the system.

Let us give an example. If we need a Date type containing three properties like year, month, and day, we have to create a “primitive” class.



A simple solution consists in creating three properties with the Integer type. We can further specify that Min and Max values for month are 1 and 12, and that Min and Max values for day are 1 and 31.





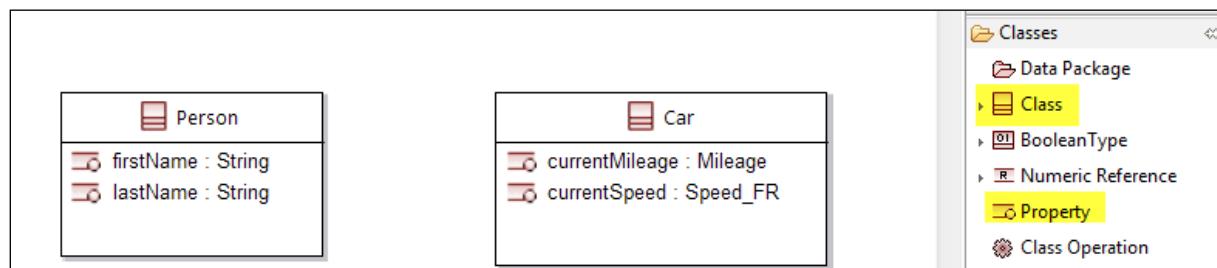
Two instances of the Date Class with the same property values cannot be distinguished ; A date does not have any intrinsic identity, contrary to a car or a person, it is a « primitive » class or complex type.

### 3.4 Class

A Class is a complex classifier.

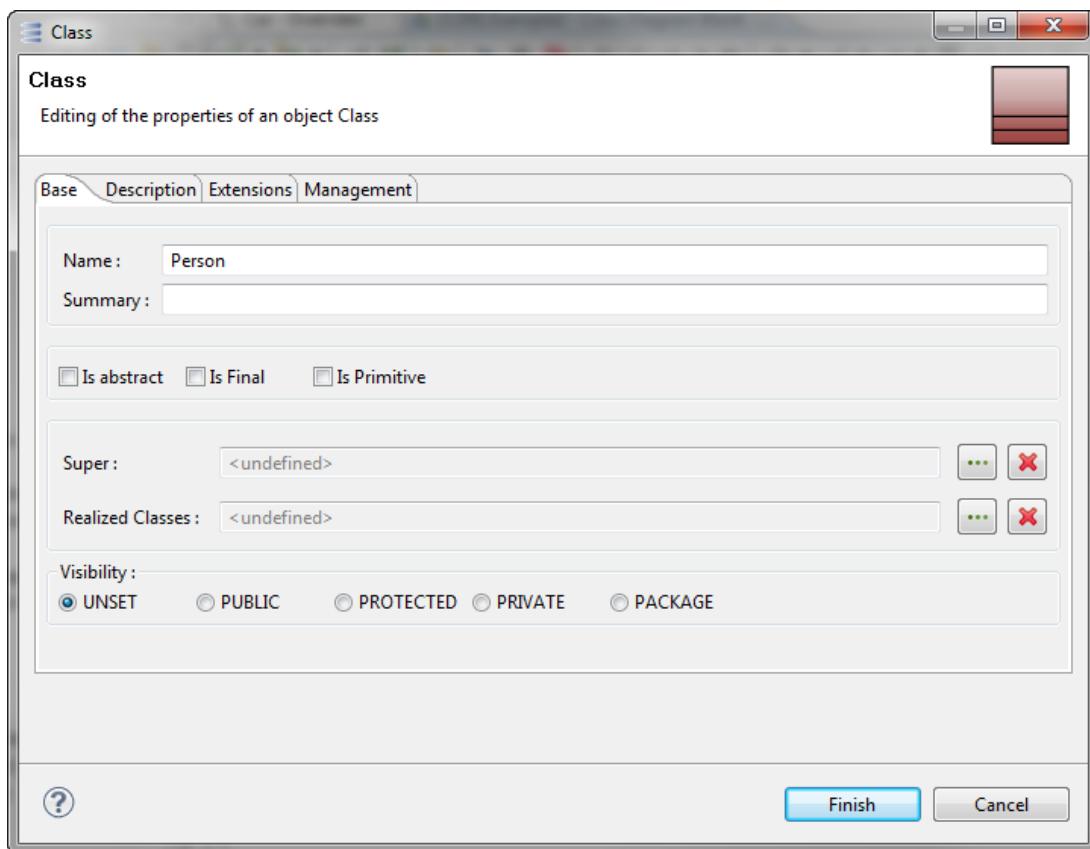
The purpose of a Class is to specify a classification of objects and to specify the features that characterize the structure and behavior of those objects.

Classes have properties, typed by other Classes, [Unions](#), [Collections](#), data types, etc. Objects of a class must contain values for each property of that class, in accordance with the characteristics of the property, for example its type and multiplicity.



Classes are the Melody declination of the UML concept of “Class”. So, many possibilities given by the UML Class Diagram are available here: classes can be associated and generalized, etc.

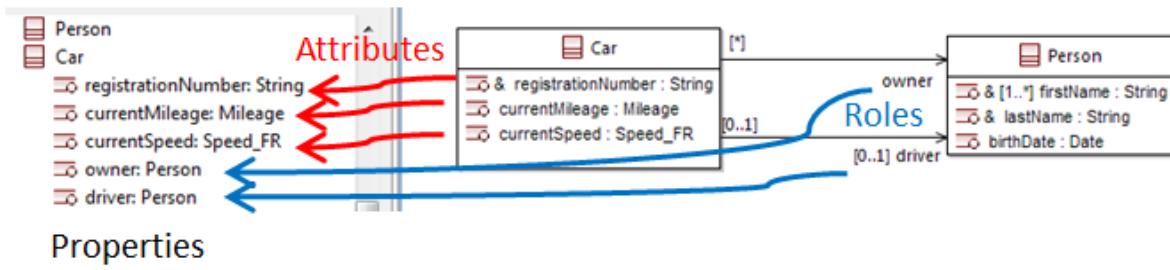
### 3.4.1 Class Editor



Field	Description	Default
Name	The name of the current Element.	<generic name>
Summary	A short headline about the role of the current Element.	<blank>
Super	One or several classes the current Class inherits. An option in the preferences allows specifying whether multiple inheritance is possible or not.	<blank>
Realized Classes	One or several classes in the previous engineering phase (for example System Analysis if the current Class belongs to Logical Architecture) refined by the current Class.	<blank>
isFinal	Specifies whether the current Class can be specialized or not. A final Class cannot be specialized by other Classes.	False
isAbstract	Specifies whether the current Class is abstract or not. An abstract Class is a Class that cannot be instantiated; it is expected to be specialized by one or several concrete Classes.	False
isPrimitive	Primitive Classes specify complex data types that can be used to type Properties held by a class, but cannot be the source or target of associations. Primitive means instances of the class have no identity in the context of the system.	False

### 3.4.2 Property

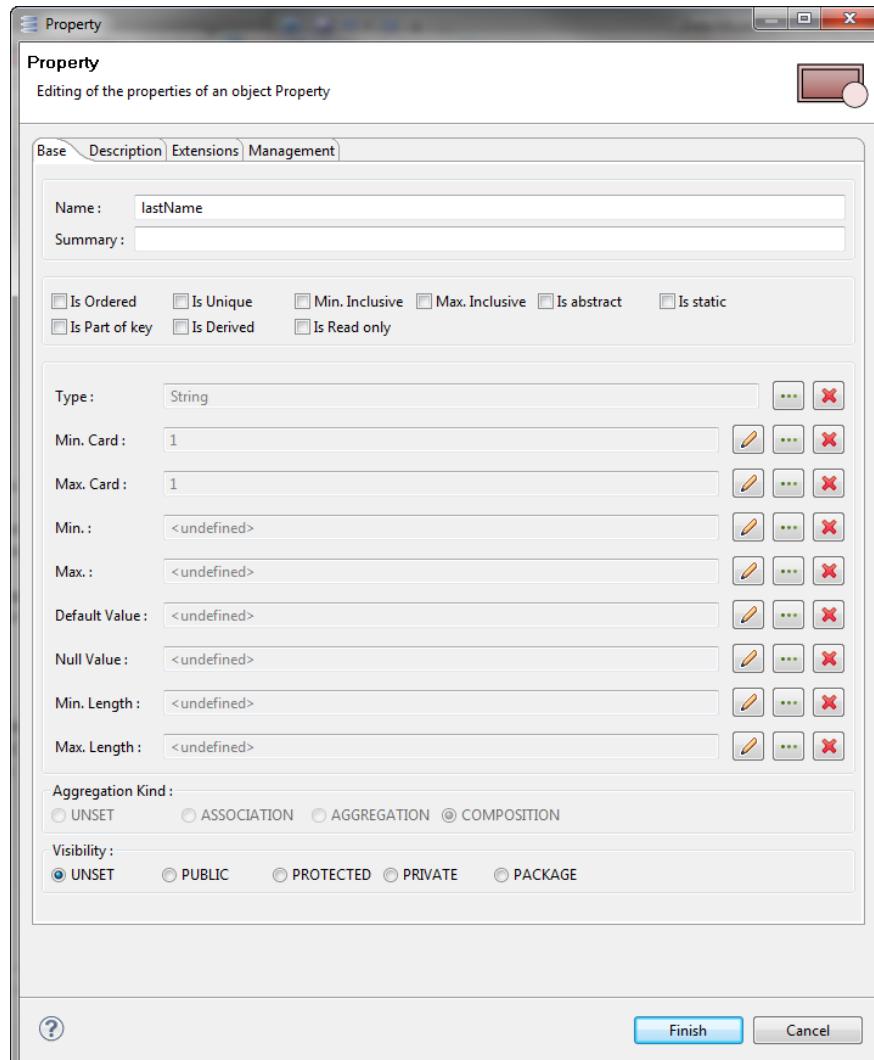
Properties of a class can be held directly by the class itself or by the navigable [role](#) of an [association](#) from this class. When held by a class, a property is also called attribute.



#### Properties

Properties are constituent members of structured elements such as Classes and Unions.

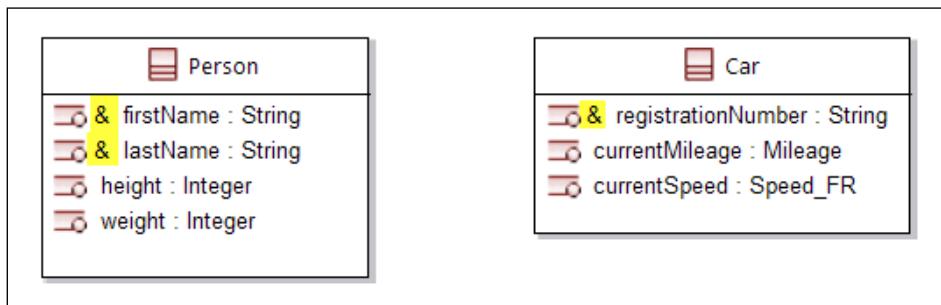
A Property has mainly a name and a type. A property is complex or simple depending whether its type is complex (class or complex data type) or simple (simple data type).



### 3.4.3 Key

A Property (attribute or association role) can be part of key. It means that the values of this property enable to distinguish different instances of the Class.

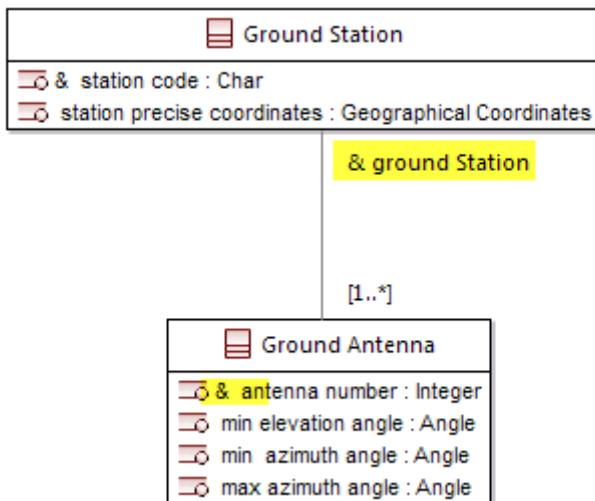
For instance, assuming there are no homonyms, we could say that both properties (firstName + lastName) make the key for the class Person. Of course, weight, height or birthday, would not be valid keys. But this is highly domain-dependent.



Key properties are noted with a “&” prefix.

Note that a key is said to be composite when it is made of several parts or a unique complex part, whereas simple keys are specified by a unique simple property.

Let us give a simple example of an association role being part of a key: a Ground Antenna is both identified by its own antenna number and the Ground Station it is related to (Ground Antennas are numbered by: Ground Antenna 1, 2 3 ... of Ground Station STA1).

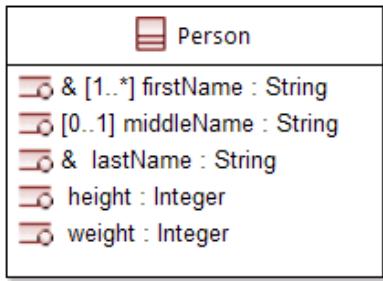


### 3.4.4 Cardinality

By default, each instance of a Class will bear exactly one value for each Property of the Class: Min. Card = Max. Card = 1.

We can specify that a property is optional, by setting its Min. Card to “0” instead of “1”. We can also specify that a property may have several values by setting the Max. Card to more than 1.

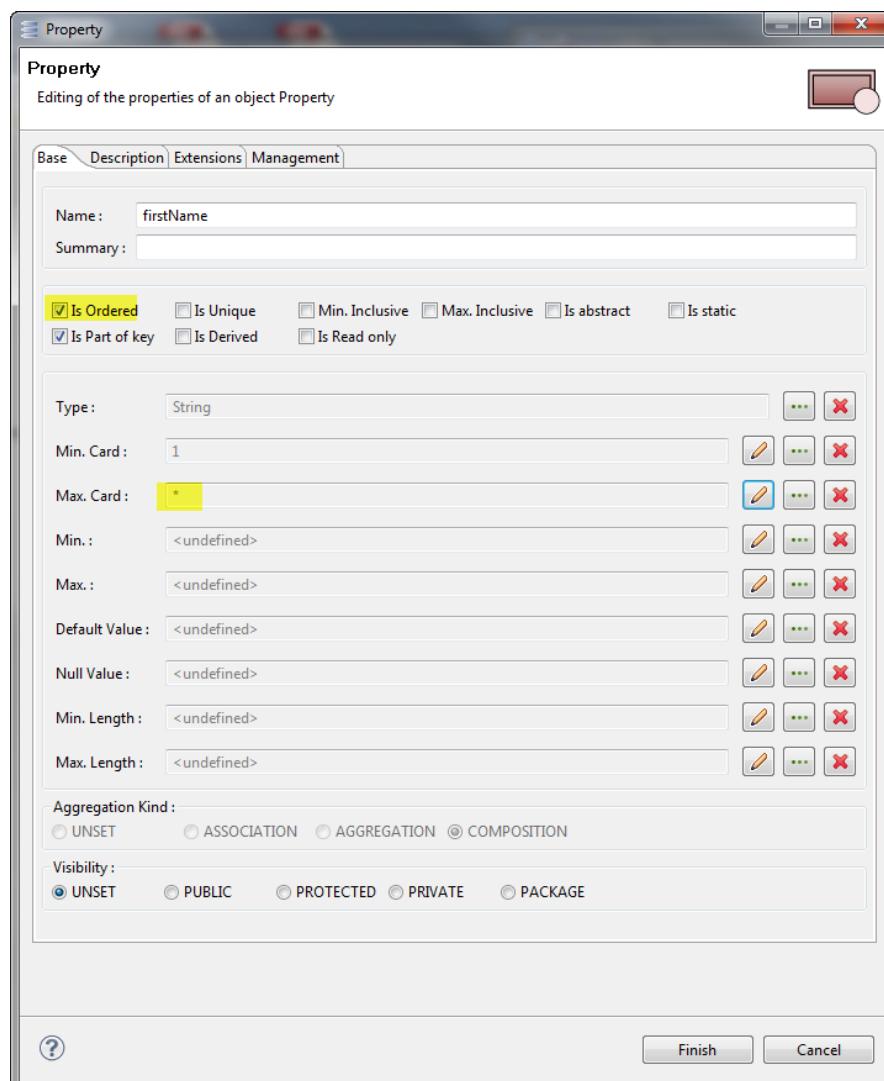
In France, for instance, when a child is born and declared, the parents must give him/her at least one first name, but there is no official number limit. The order of declaration is important, and the first one is the official first name. Let us also specify an optional middle name ...



The cardinality bounds are typically shown in the format:

<lower-bound> ‘..’ <upper-bound>

where <lower-bound> is an integer and <upper-bound> is an unlimited natural number. The star character (\*) is used as part of a cardinality specification to represent the unlimited (or infinite) upper bound.



Note that for a multi-valued property to be part of a key, it shall be ordered.

The usual practice is to allow only properties with [1..1] multiplicity as being part of a key.

### 3.4.5 Other Property basic features

The following table synthetizes the available features of a property according to the different simple type kinds.

	Boolean	Enumeration	String	Numeric	Phys. Quan.
Min / Max	No	No	No	Yes	Yes
Default Value	Yes	Yes	Yes	Yes	Yes
Null Value	Yes	Yes	Yes	Yes	Yes
Min. / Max. Length	No	No	Yes	No	No

These features, when specified for a property, overwrite those, if any, defined for the property's type.

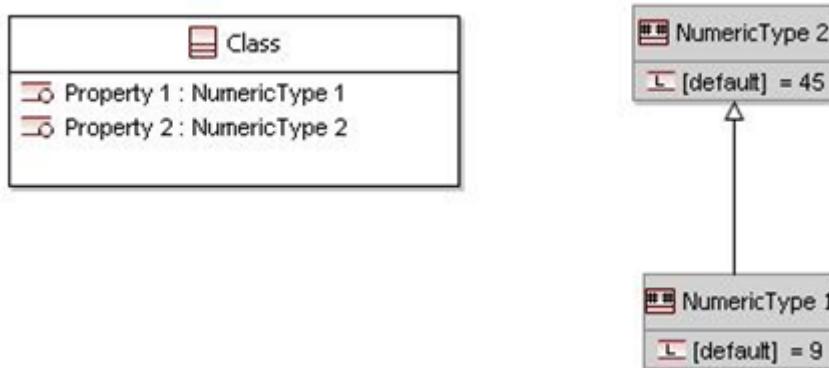
As a good practice, defining these features in types should be preferred to defining them in properties. When features are defined in types the model is clearer. A drawback of this practice is that it leads to defining a larger number of types.

### 3.4.5.1 Default Value

When an object is instantiated in a class, for every attribute of the class that has a specified default, if an initial value of the attribute is not specified explicitly for the instantiation, then the default value specification is evaluated to set the initial value of the attribute for the object [UML].

The default value can be overwritten in a subtype.

Example: Numeric Type 1 redefines to 9 the default value of Numeric Type 2 (45).



### 3.4.5.2 Null Value

If the null value is defined for a type, wherever a value of this type is expected, the null value shall be provided to actually specify "no value". The null value of a type is the concrete (physical) value to specify "no value" from a logical point of view.

The null value can be overwritten in a subtype.

### 3.4.5.3 Min/Max

Minimum and maximum values for range.

If pertinent, *Min. / Max. Inclusive* properties indicate if min/max values are inclusive in the range.

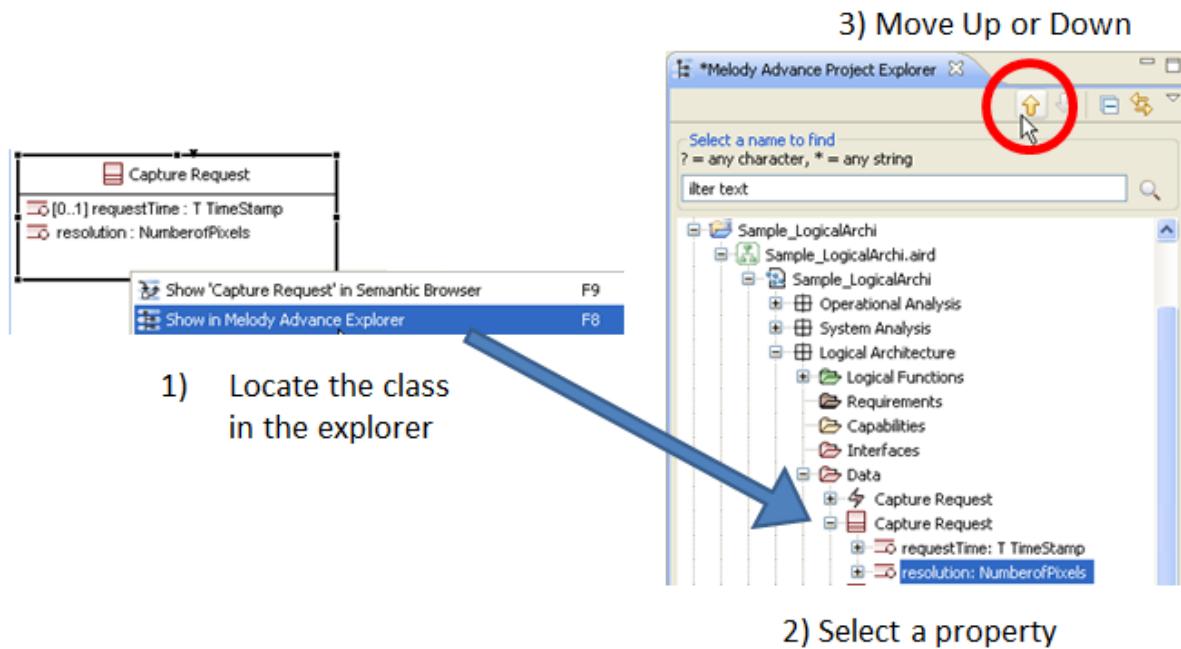
### 3.4.5.4 Min. Length/Max. Length

Minimum and maximum values for String length.

### 3.4.6 Property ordering

Property ordering may be considered for data detailed definition.

Note that, with Melody, ordering can be changed inside the model explorer, but not directly from diagrams.



### 3.4.7 Operation

An operation is a behavioral feature of a class that specifies the name, type, parameters, and constraints for invoking an associated behavior.

An operation belonging to a class should not be confused with an [operation](#) belonging to a component, available as an Exchange Item in the Communication Model.

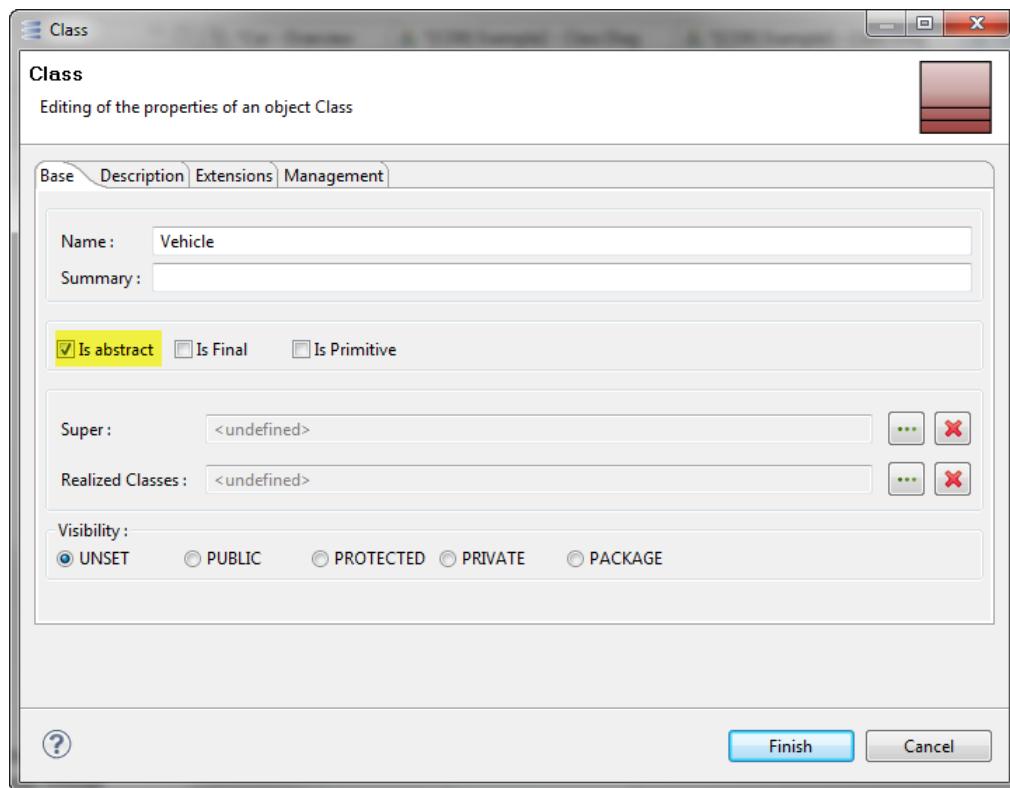
From a system engineering perspective, unless specific needs, use of operations in classes is not recommended.

### 3.4.8 Abstract Class

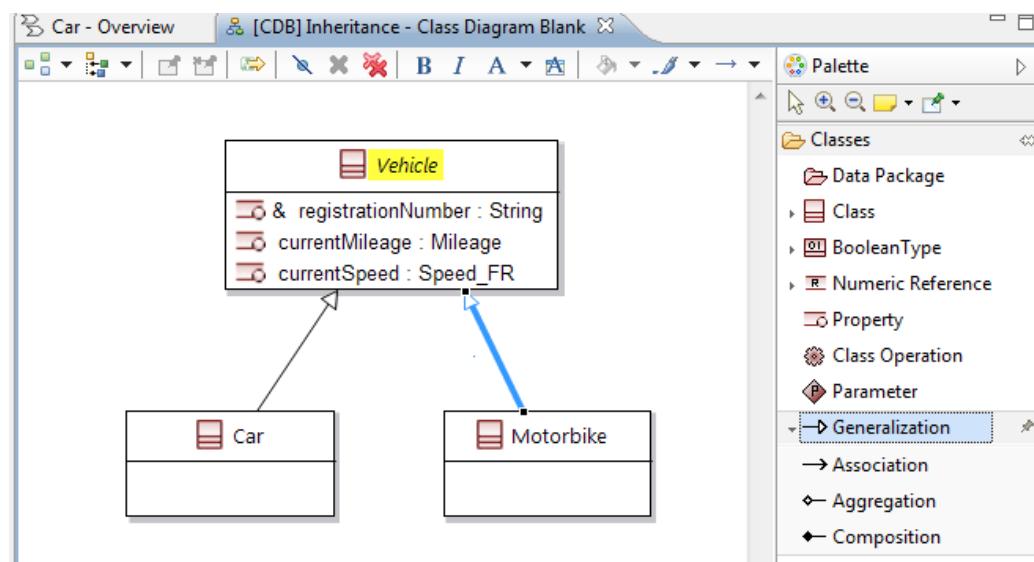
An abstract Class is a class that cannot be instantiated; it is expected to be specialized by one or several concrete classes (subclasses).

Abstract class names are displayed in italics in diagrams (look at *Vehicle* in the following CDB).

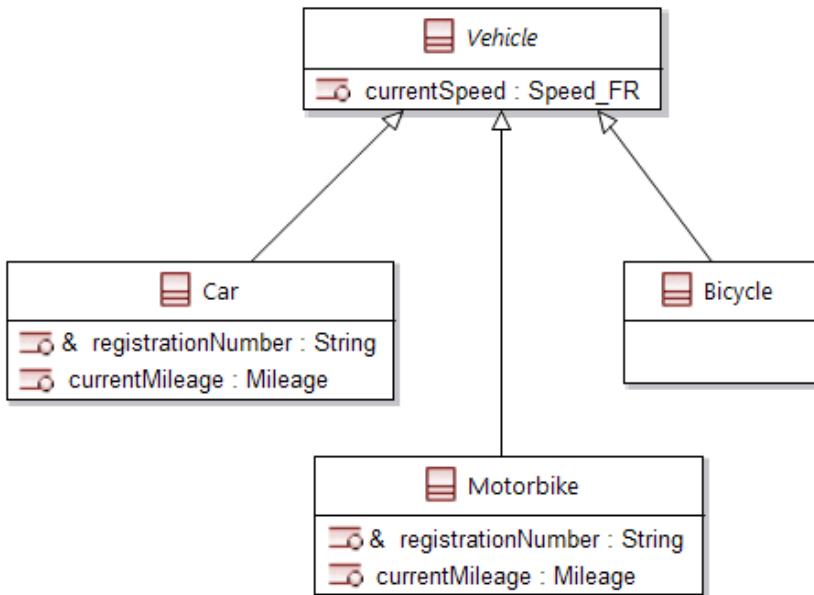
In our example, let us speak also about motorbikes. Cars and motorbikes are “vehicles”. We can create an abstract Class “Vehicle” that will factorize all the common properties for cars and motorbikes.



*isAbstract*: Boolean. Default value is false.



Take care that all properties of the generalized class (super-class) must be meaningful for all the subclasses! If we wish to add the bicycles to our model, it would not be a good idea just to add a third subclass, as it would imply that bicycles have also registration numbers and mileage. A better model would be the following one:



### 3.4.9 isPrimitive

When a class is tagged “isPrimitive”, it becomes a [Complex Type](#).

Primitive Classes can be used to type other Class Properties, but cannot be the source or target of [associations](#). Primitive means instances of the class have no identity in the context of the system.

Warning: as a consequence, converting a class to a primitive might delete all its associations. It depends on your Melody preferences.

## 3.5 Values

Values are instances of classes and data types and allow defining fixed data values. There are named values and anonymous values.

Only named values can be reused through value references to specify other model elements. In this case they have the semantics of a constant in the system and can be referenced to build other data values or to define:

- default/null/max/min value of a numeric DataType or property,
- default/null value and max/min length of a String DataType or property
- default/null value of an enumeration DataType or property,
- default value of a boolean DataType or property,
- min/max cardinality of a property

An anonymous Data Value cannot be referenced.

### 3.5.1 Literal Numeric Value

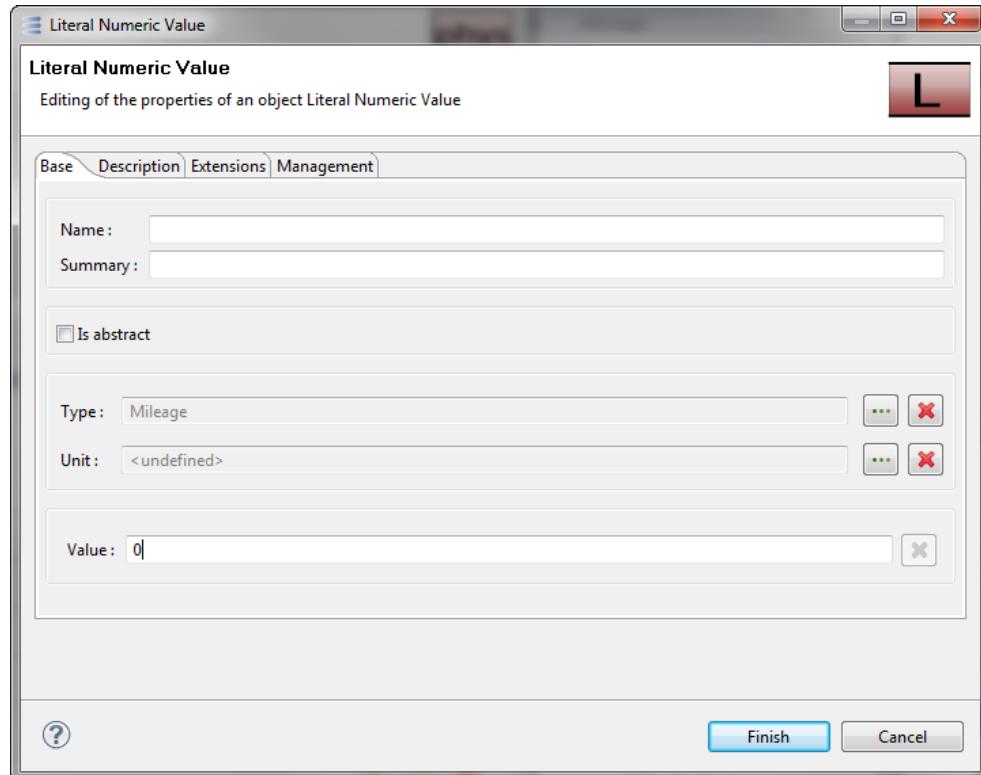
Literals allow defining fixed Data Values.

Examples of Numeric Literals for a Numeric Type of Kind FLOAT:

- 25
- +6.34

- 0.5
- 25e-03
- -1

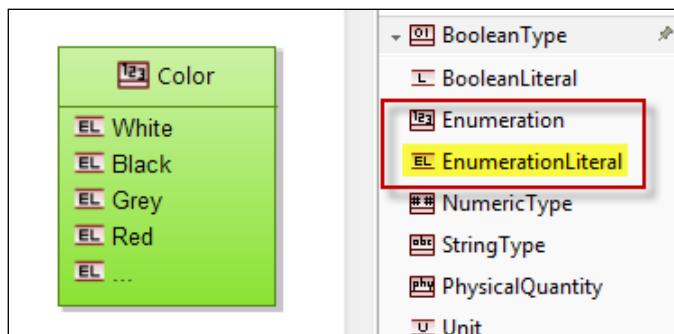
In our example, we just wanted to express that the minimum mileage value is "0".

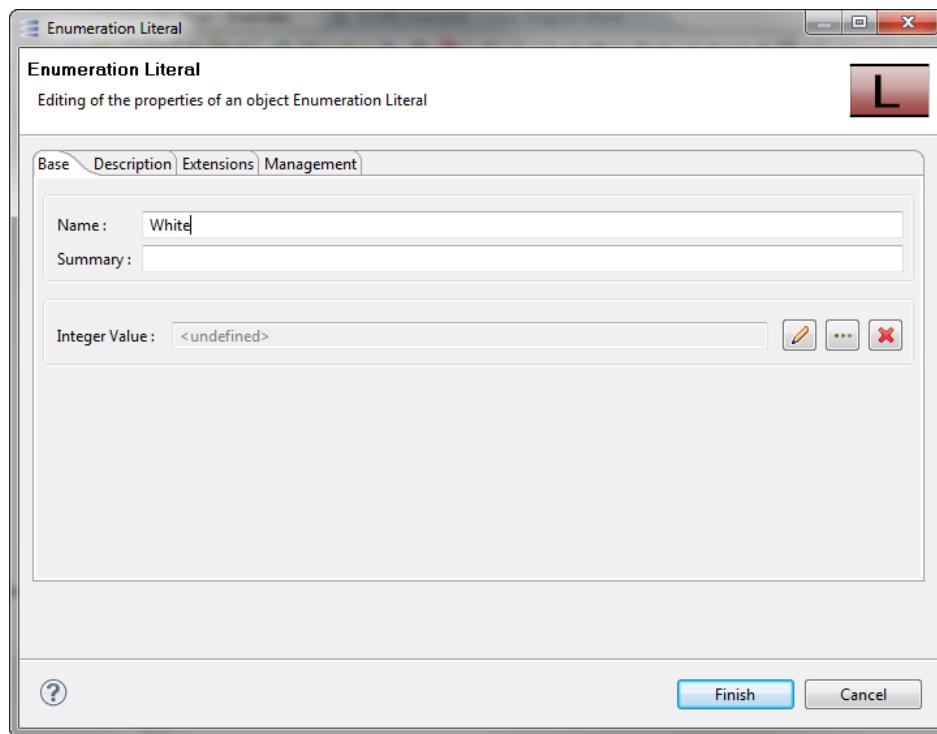


### 3.5.2 Enumeration Literal

An Enumeration Literal is a user-defined possible [Enumeration](#) instance.

We can specify the precise list of available colors for cars. Let us begin...

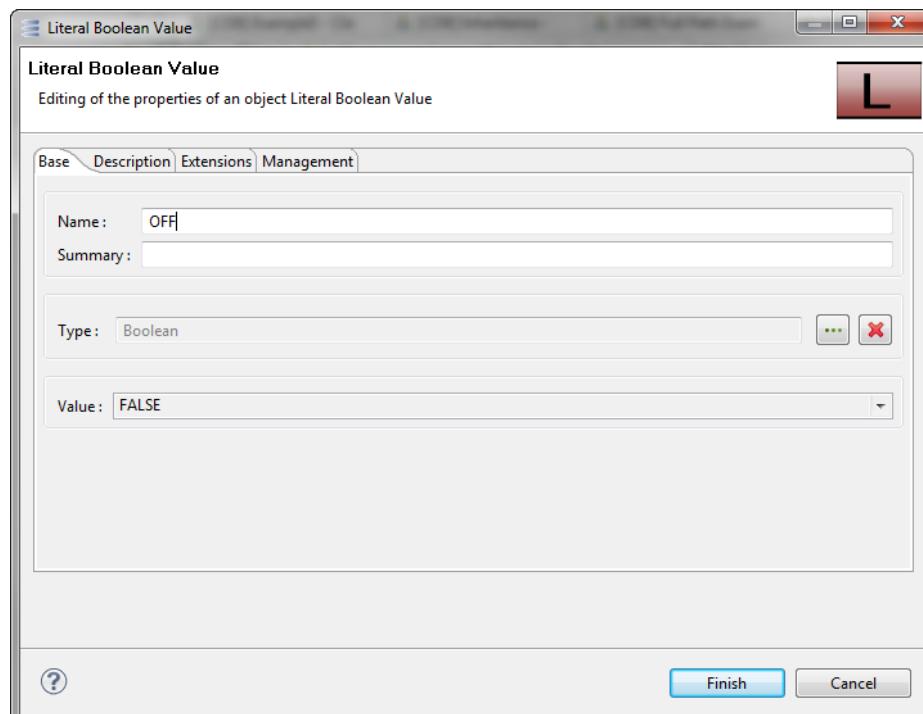




Each Enumeration literal can be associated optionally with an Integer value, enabling to treat them as an ordered set (next, previous). In Melody 3.0, the *Integer Value* field is replaced by a more general *Domain Value* field which should be consistent with the Domain Type declared in the owning Enumeration data type.

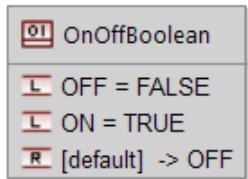
### 3.5.3 Literal Boolean Value

Literal Boolean values are used to define the names of true and false values of a Boolean type.



Examples of Boolean Literals:

- YES, NO
- ON, OFF



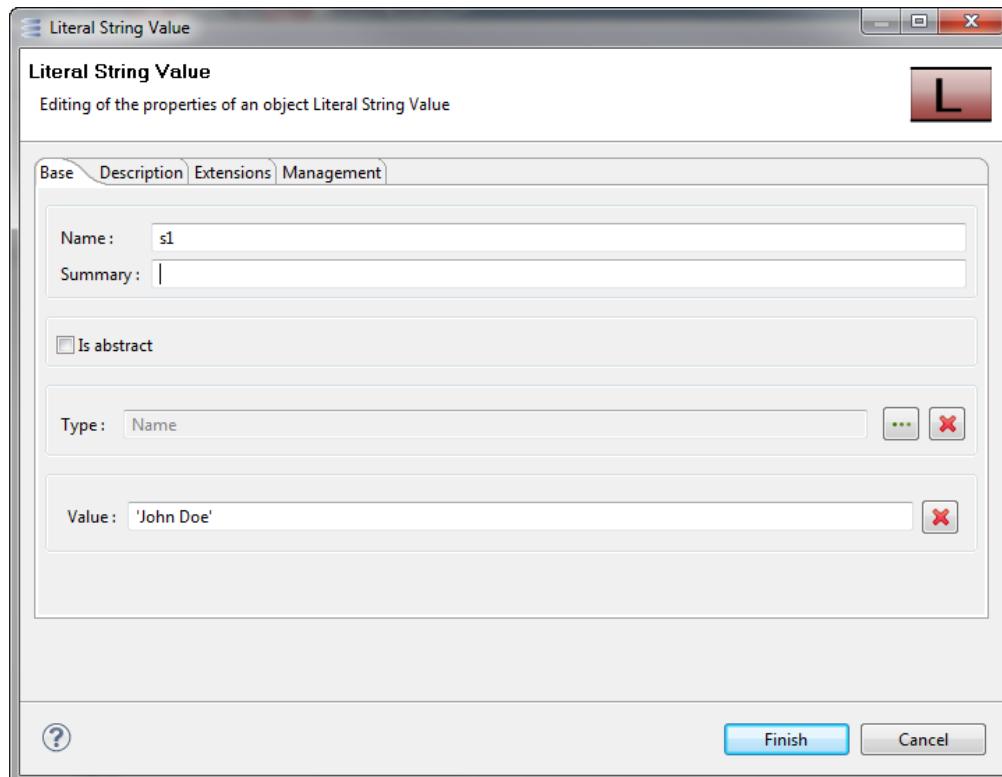
A good practice is to have only one Boolean type in a model. If there is a need of defining other values (Yes/No, etc.), it is strongly recommended to create an Enumeration simple type owning two Enumeration literals, and then to associate each of the literals to either false or true value of the Boolean type. This feature is available from Melody version 3.0 (by choosing Boolean as Domain Type of the Enumeration).

### 3.5.4 Literal String Value

Literals allow defining fixed Data Values.

Examples of String Literals:

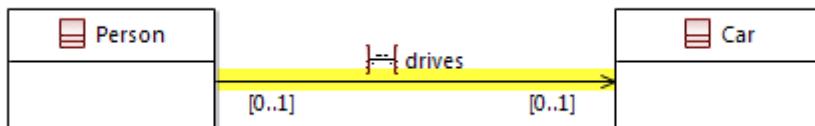
- 'NONE'
- 'January 2014'
- 'John Doe'



These String Literals can then be used in default and null values of properties typed by a String data type.

## 3.6 Association

An Association is a semantic relationship between two Classes (or Unions and Collections). “Association” represents a relationship shared among the instances of two classes.



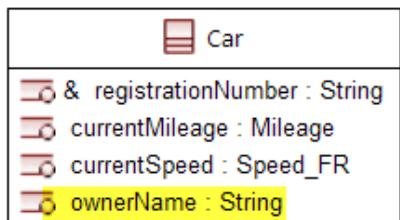
An association is normally drawn as a solid line connecting two types, or a solid line connecting a single classifier to itself (the two ends are distinct).

Note: Distinction between “attribute” and “association” properties should be made upon the basis of the type of the element linked to the class:

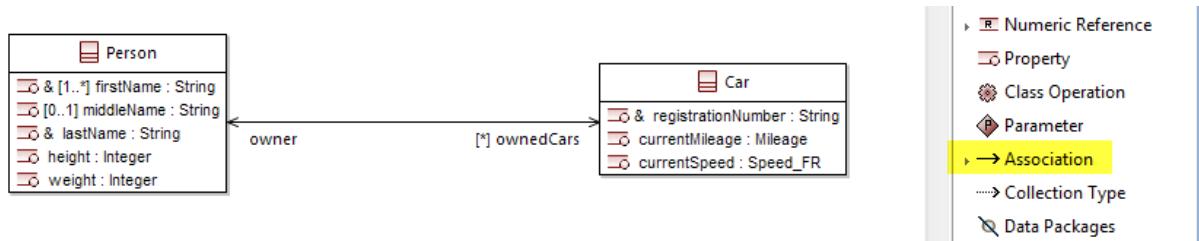
- Relationship towards a “Data type” or a “primitive Class” leads to the creation of a single class property.
- Relationship towards a non-primitive “Class” is achieved through the creation of an association between the two classes.

As a consequence, it is neither possible to create an association towards a Datatype / primitive Class, neither possible to type a single Property with a non-primitive Class.

In our example, imagine we just need the last name for the owner of a car, assuming there are no homonyms, the following simple model would be sufficient.

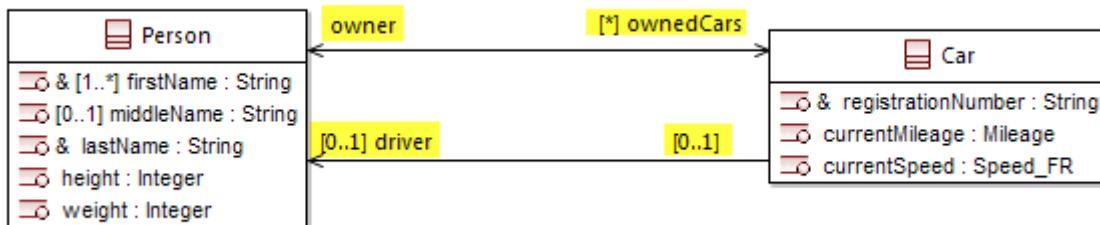


But as soon as we need more information on the owner, the best is to add a Class Person, and create an association between Car and Person, as in the next figure.



### 3.6.1 Role

An association end is the connection between the line depicting an association and the box depicting the connected class. A name string may be placed near the end of the line to show the name of the association end. The name is optional and suppressible.



There can be several associations between two classes, each representing a specific relationship (with its own properties). For instance, let us assume that a car always has one and exactly one owner, whereas it can have zero or one driver at a time. The Person class plays two roles with respect to class Car: owner and driver roles. Conversely, a person can own several cars simultaneously, but cannot drive more than one car.

### 3.6.2 Cardinality

Cardinality is a definition of an interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound.

- isOrdered : Boolean

For a multivalued multiplicity (Max. Card > 1), this attribute specifies whether the values in an instantiation of this element are sequentially ordered. Default is false. See § [isOrdered](#).

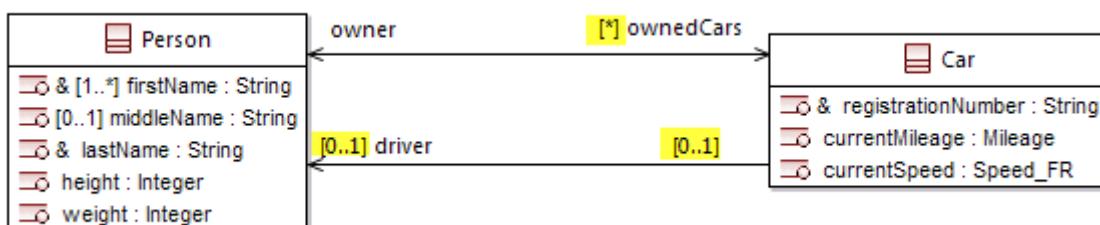
- isUnique : Boolean

For a multivalued multiplicity (Max. Card > 1), this attribute specifies whether the values in an instantiation of this element are unique. Default is true. See § [isUnique](#).

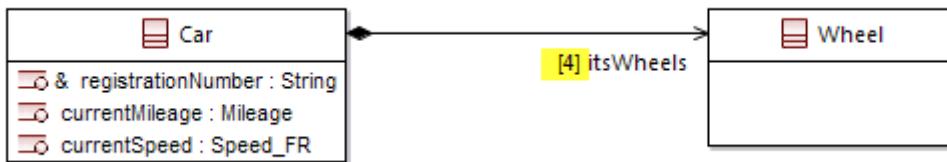
Typical cardinalities are:

- [0..1] = optional
- [1..\*] = at least one
- [n..\*] = at least n
- [n..m] = at least n, but not more than m
- [0..\*] = [\*] = any number
- [1..1] = [1] = exactly one

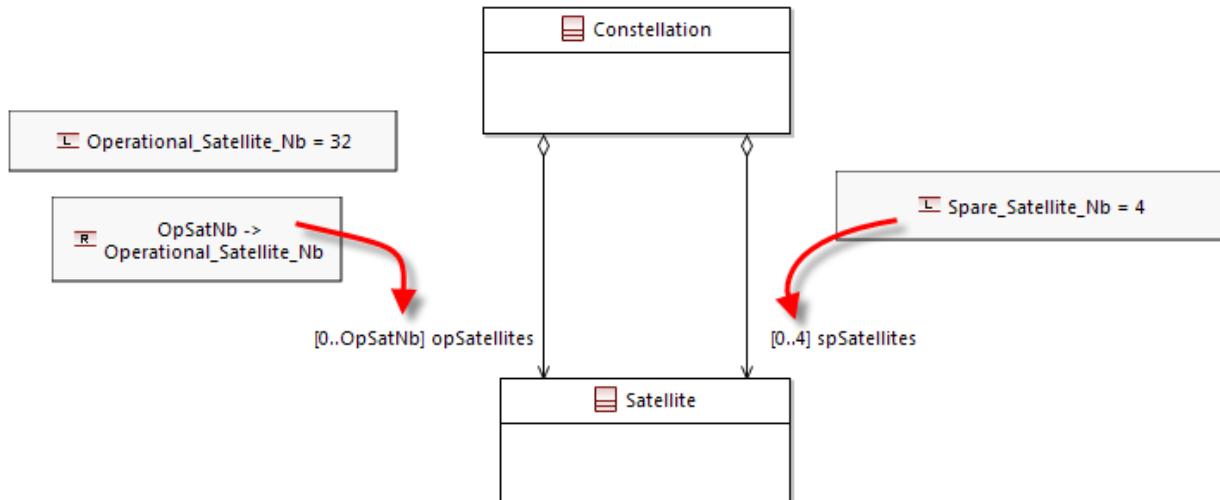
A cardinality with zero as the lower bound and an unspecified upper bound may use the alternative notation containing a single star "\*" instead of "0..\*". In Melody, as for Properties, the default is supposed to be "1..1", and is not displayed.



If the lower bound is equal to the upper bound, then an alternate notation is to use the string containing just the upper bound. For example, "4" is semantically equivalent to "4..4" (exactly 4).



The lower and upper bounds for cardinality may be specified by [Literal values](#) or [References](#) (it is also true for attribute cardinalities).



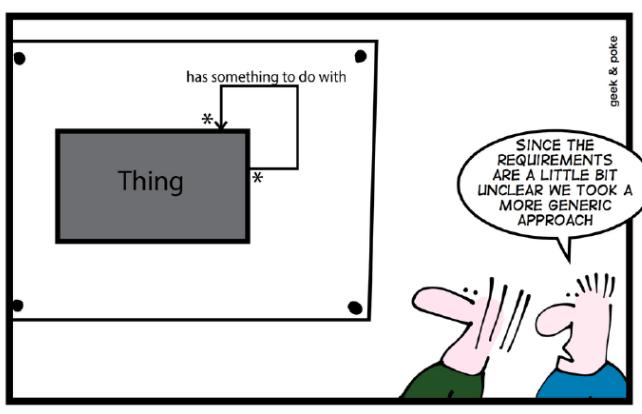
For instance in our example, the Constellation contains both operational and spare satellites. To be more flexible, we do not want to fix the number of operational and spare satellites once and for all in the cardinality ranges. We prefer to refer to named constants or references, so that if they appear in multiple diagrams, we only have to change the unique definitions.

Notice the difference:

- On the right, we used directly a [Literal Numeric Value](#), called Spare\_Satellite\_Nb. It is the value of this Literal (4) that is displayed inside the cardinality;
- On the left, we used a Numeric Reference, called OpSatNb, referencing a Literal Numeric Value, called Operational\_Satellite\_Nb. This time, the name of the reference is displayed, and not the numeric value (32).

A good practice to name classes that are not collections is to always use singular. Do not forget that cardinalities are there to express multiplicity!

Take care also not to abuse of [0..\*] or [\*] cardinalities: they are never wrong but often too imprecise!

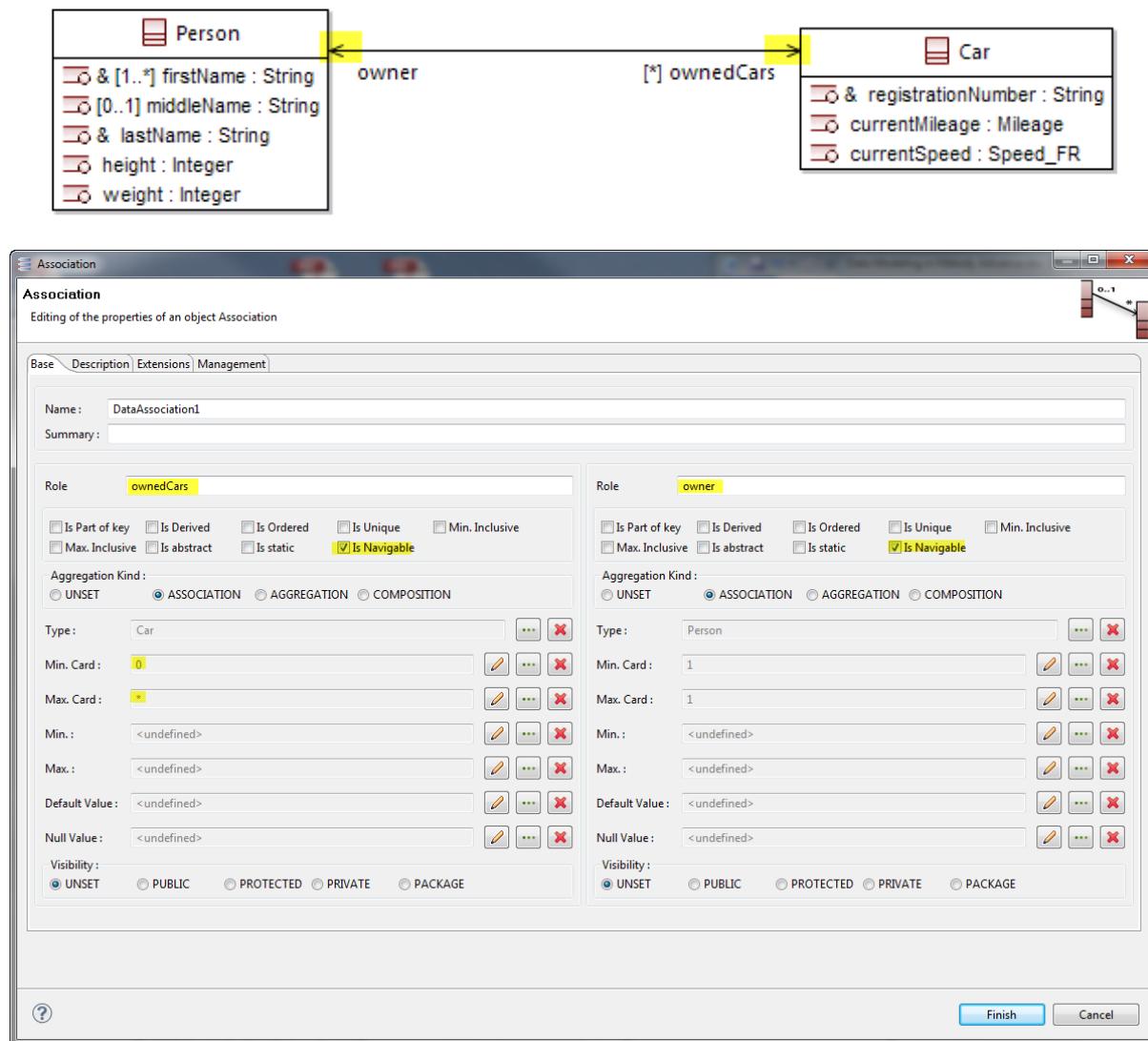


(Source: Geek and Poke, July, 2013)

### 3.6.3 Navigability

Navigability means instances participating in links at runtime (instances of an association) can be accessed efficiently from instances participating in links at the other ends of the association. The precise mechanism by which such access is achieved is implementation specific. If an end is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient.

An open arrowhead on the end of an association indicates the end is navigable. In the following example, the Person can access directly to his/her owned cars, and the Car can access to its owner.



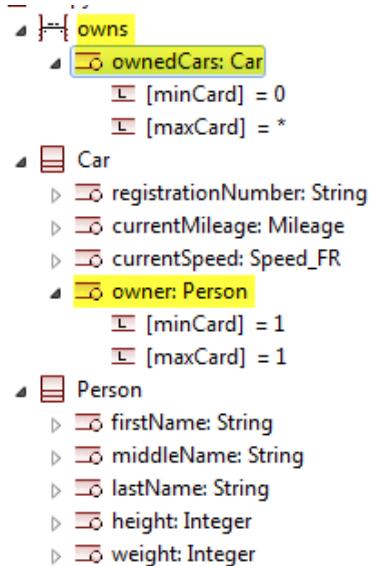
Specially when a modeling objective is code generation, it is strongly advised to reduce the navigability of associations to only one direction to minimize coupling. In the following diagram, for instance each Car knows directly its owner, but a person cannot access directly to ownedCars. This does not mean that the system does not have the capability to access the cars owned by a person, but it will not be performed directly from the person side.

If the role is not navigable, its name is optional, and is not displayed on the diagram. Conversely, the role name must be filled in the navigable role side.



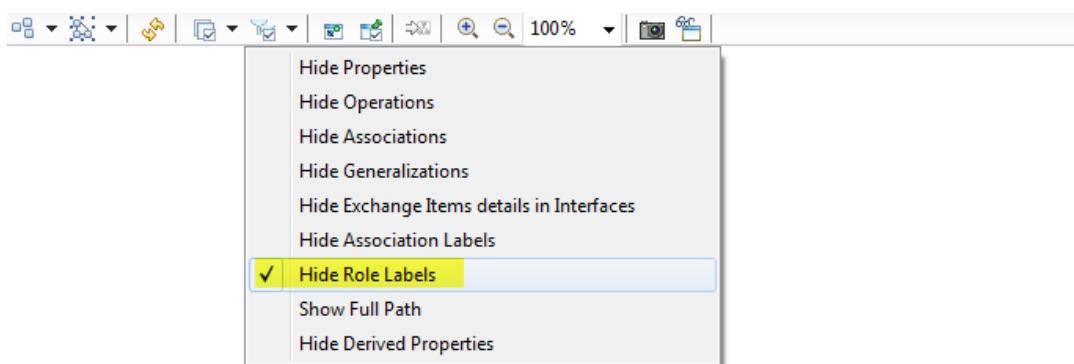
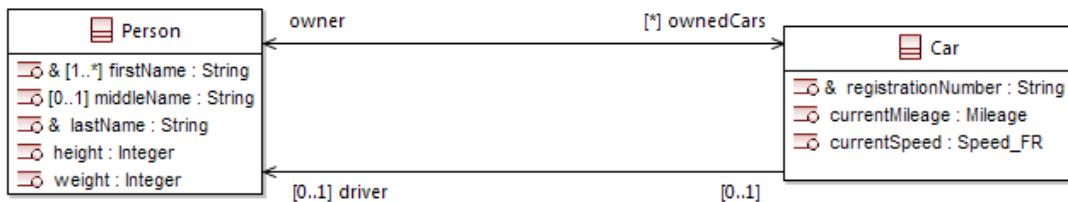
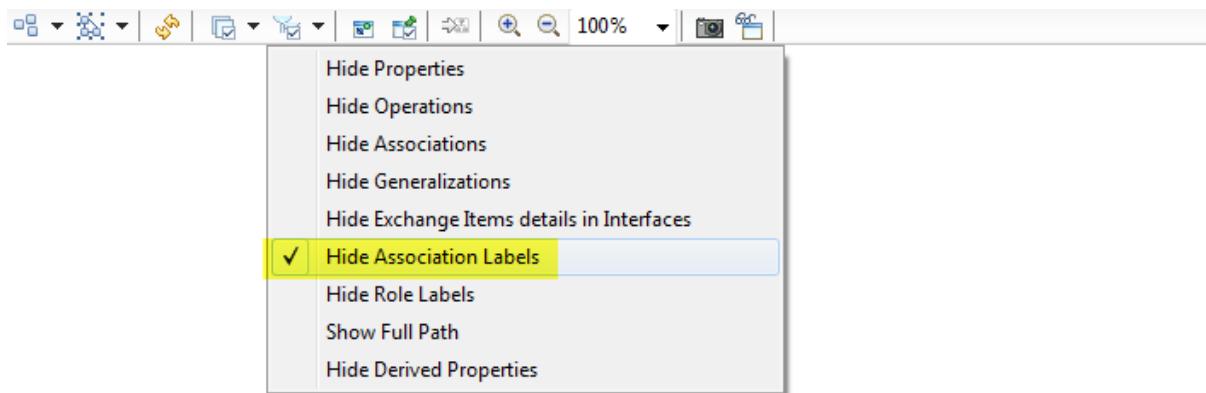
Note also that:

- navigable roles appear as properties of the opposite class (as the owner of type Person in class Car) in Melody project Explorer ,
- whereas non navigable roles appear as properties of the association itself



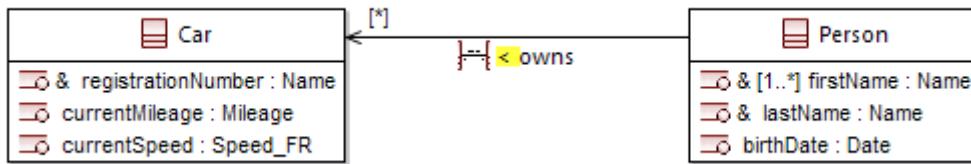
### 3.6.4 Name

It is also possible to give a name to the association itself, instead of or in addition to the role names. Take care: by default, the name of the association does not appear on the diagram! You have to deactivate the default filter called “association label”.



Take care that if you change the layout of the diagram, for instance if you put Car on the left and Person on the right, the verbs will read in the wrong direction! Very often, the reading direction of each verb is obvious from the domain knowledge, but it may not be the case. Two possibilities are available: either prefer the role names, or use additional symbols such as < or ^ (but you will have to modify them if you change again the layout).

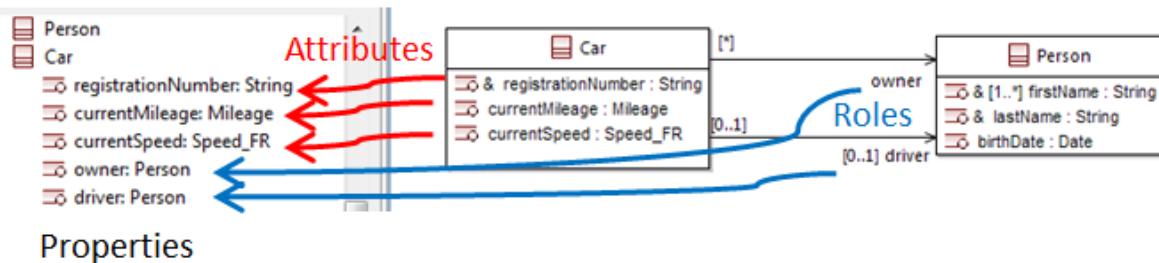
And do not forget that navigability (arrows on associations) and direction of the verb naming the association are completely distinct notions !



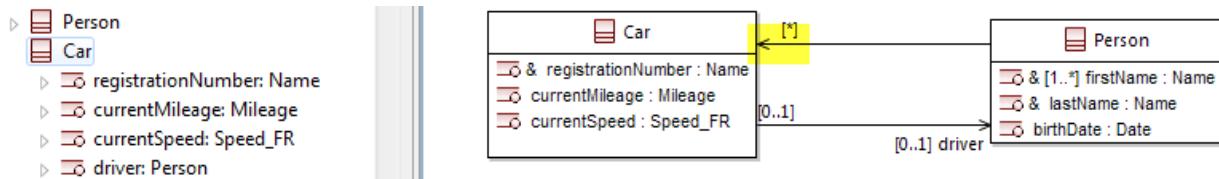
### 3.6.5 Associations in Melody Advance Project Explorer

Properties of a class (attributes or navigable roles) are shown in the same way under their owning class in Melody Advance Project Explorer.

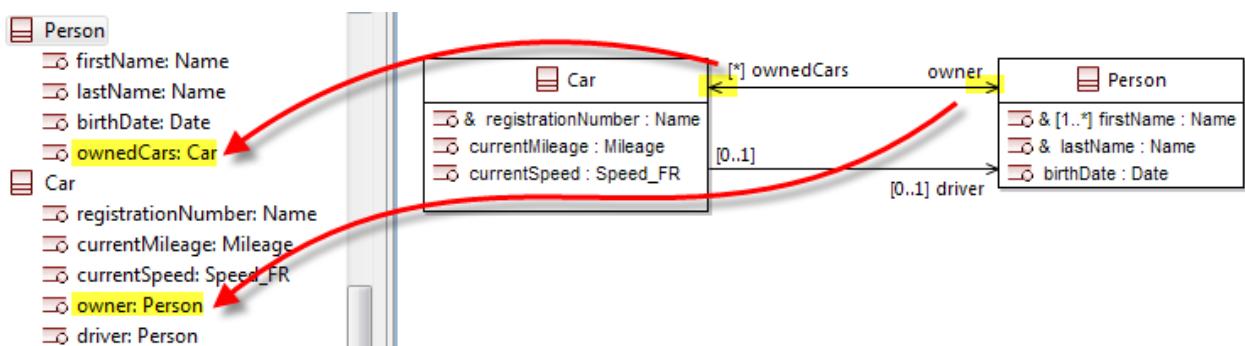
A first example is given by the next figure, where the class Car is shown in the Project explorer with 3 attributes and two navigable roles.



As soon as we reverse the navigability of the first association, one property disappears (owner: Person).



When an association is navigable in both directions, one property for each navigable role is visible under each class.



Remember that non navigable roles appear as properties of the association itself.

Take care also that if you ever change the name of a Class, the names of the roles pointing to it will not be changed automatically!

### 3.6.6 Aggregation, composition

When Aggregation Kind of a role is ASSOCIATION, the association is a simple association. For this role, the origin class has a reference to the target class which shall own or inherit a key.

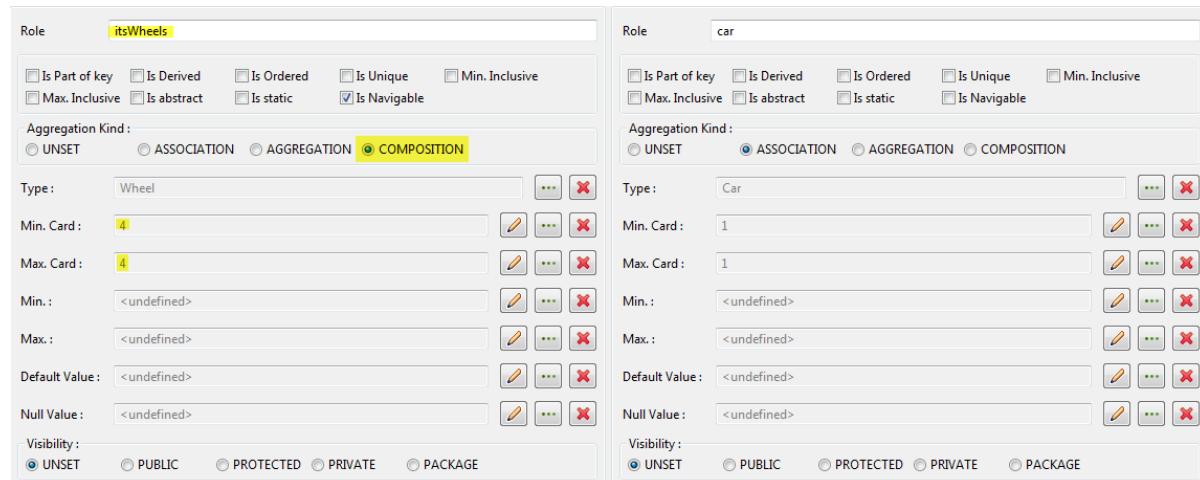
"Aggregation" is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-of relationship.



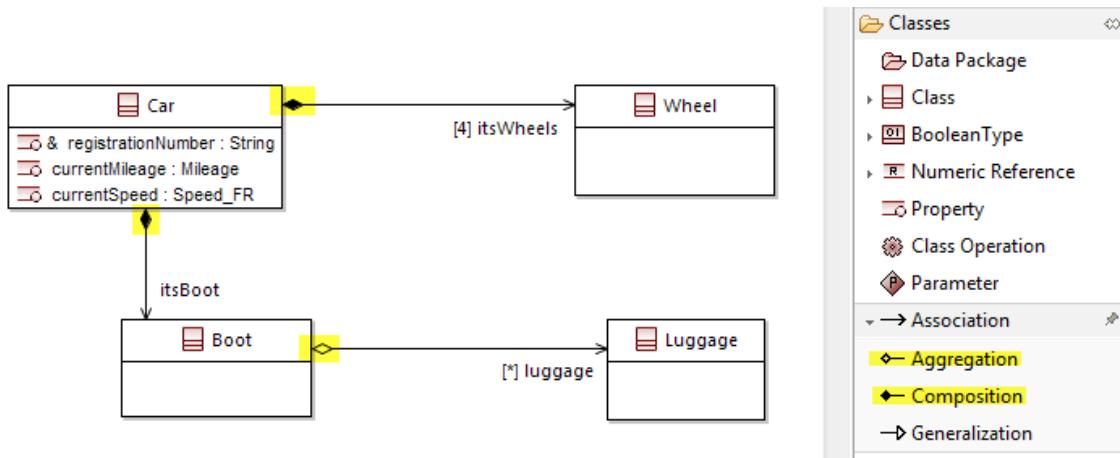
Take care: as this concept does not have a universally agreed semantics (it is a semantic variation point in UML), it is advised not to use it unless you define a clear semantics for your project.

"Composition" (or composite aggregation) is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite. Compositions may be linked in a directed acyclic graph with transitive deletion characteristics; that is, deleting an element in one part of the graph will also result in the deletion of all elements of the composition subgraph below that element.

Composition is represented by a black diamond on the origin end of the association.

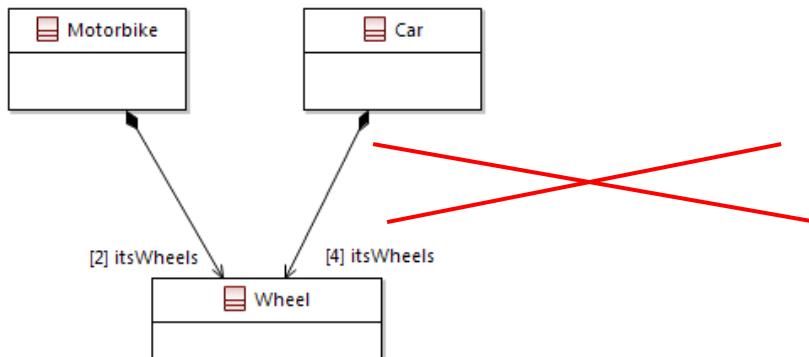


An association with Aggregation Kind = AGGREGATION (shared association in UML) differs in notation from simple associations in adding a hollow diamond as a terminal adornment at the aggregate end of the association line. An association with Aggregation Kind = COMPOSITION has also a diamond at the aggregate end, but differs in having the diamond filled in.

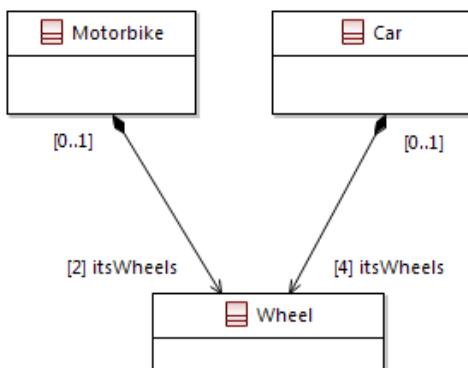


In our example, we have expressed the fact that a car always contains four wheels and a boot. These elements are present when you buy the car; they belong to the product definition (EPBS). On the contrary, there may be luggage in the boot from time to time, but luggage does not belong to the product definition.

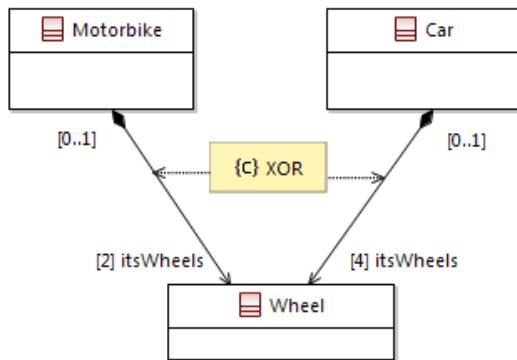
Take care not to create multiple compositions where the same Class is involved as a part. It is very often the symptom of a wrong analysis. At least, pay attention to the cardinality on the composite side: the same element cannot belong to several composites! For instance, the following diagram is correct when you read it from top to bottom. But when you read it from bottom to top, it expresses the false assertion that a Wheel belongs both to a Car and a Motorbike!



A better solution is to specify [0..1] cardinalities on the composite sides.

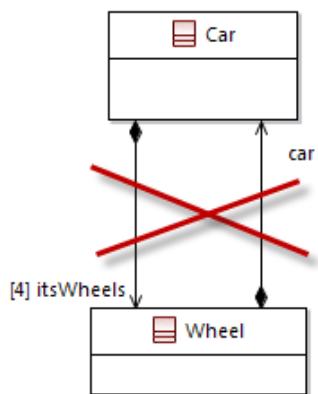


An even better model is the following one (with the XOR constraint):



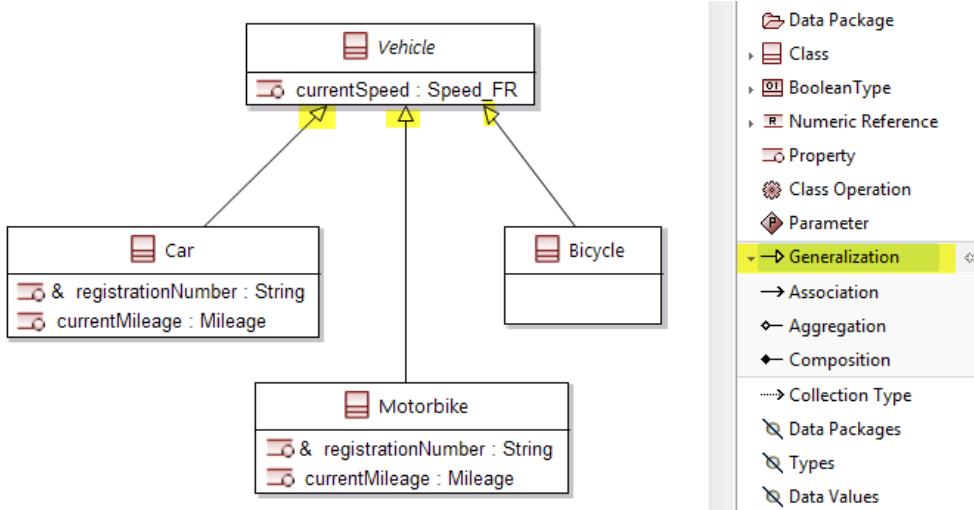
Refrain absolutely from circular compositions (or aggregations). Warning : Melody model checking does not detect them in V2.5.

Note also that Aggregation type, navigability, and end ownership are orthogonal concepts, each with their own explicit notation.



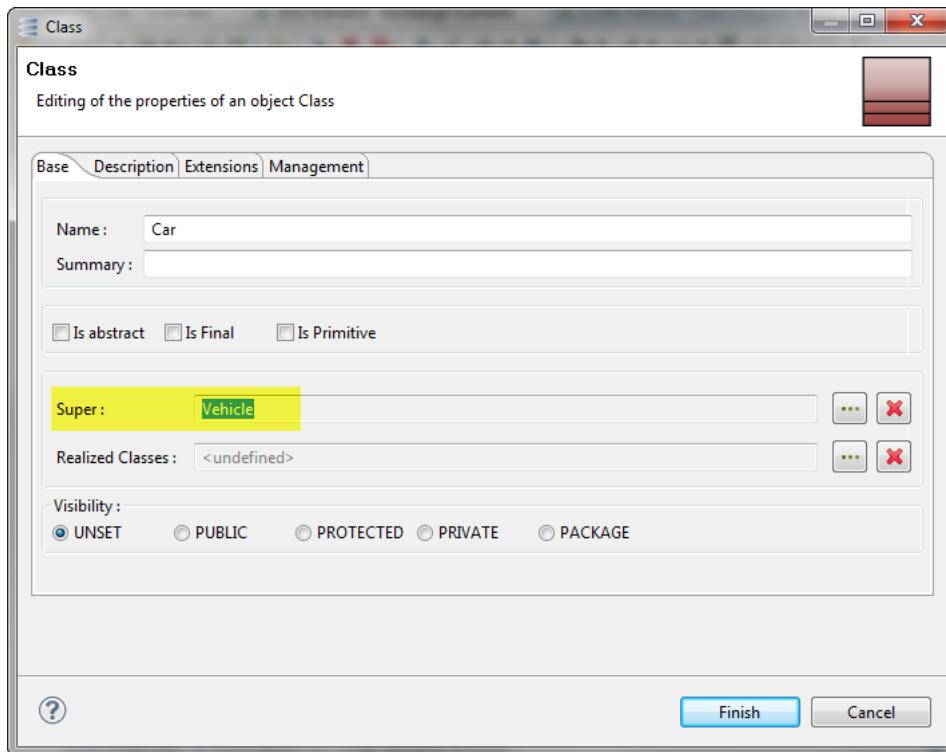
### 3.7 Generalization

Where a generalization relates a specific class to a general class, each instance of the specific class is also an instance of the general class. Therefore, properties specified for instances of the general class are implicitly specified for instances of the specific class. Any constraint applying to instances of the general class also applies to instances of the specific class.



A Generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classes. The arrowhead points to the symbol representing

the general class. The Generalization relationship is also shown in the Properties sheet under the topic “Super”, which indicates the super-class.

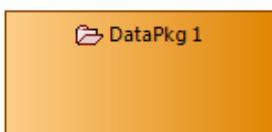


Mutual generalizations between classes A and B are impossible and Melody prevents to create the second generalization in the opposite direction. Moreover, Melody even detects circular generalizations and prevents them.

## 3.8 Package

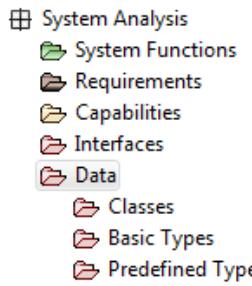
### 3.8.1 Definition

A package is used to group elements, and provides a namespace for the grouped elements. A package may contain other packages. A package owns its members, with the implication that if a package is deleted from a model, so are its owned elements. In Melody, packages are shown as orange rectangles.

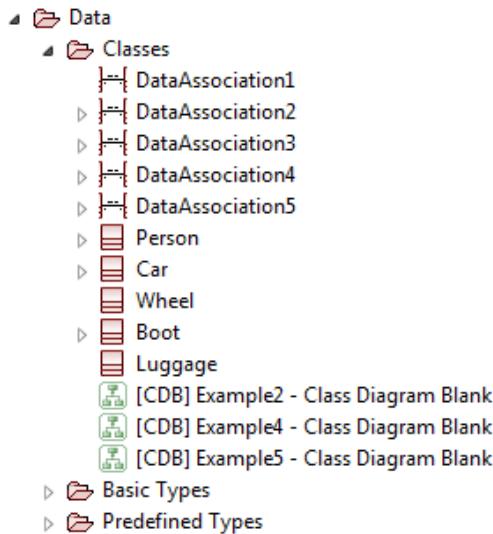


In our example, we have organized all the System Analysis Types into three packages:

- *Predefined Types* was created by Melody;
- *Basic Types* contains all Units, Numeric Types, Physical Quantities, etc., as well as Primitive Classes;
- *Classes* contains the domain concepts.



It is highly recommended not to put all the types « flat » in the *Data* existing package. Inside *Classes*, and *Basic Types*, we could have created subpackages to better structure the Data Model. Usually classes and basic types are mixed in packages for the structuration in packages shall rely on business criteria and not on type kind (class vs datatype). A useful recommendation is to limit the number of datatypes and classes in a package to 20.



A package can contain Types, Exchange Items, but also Associations and diagrams.

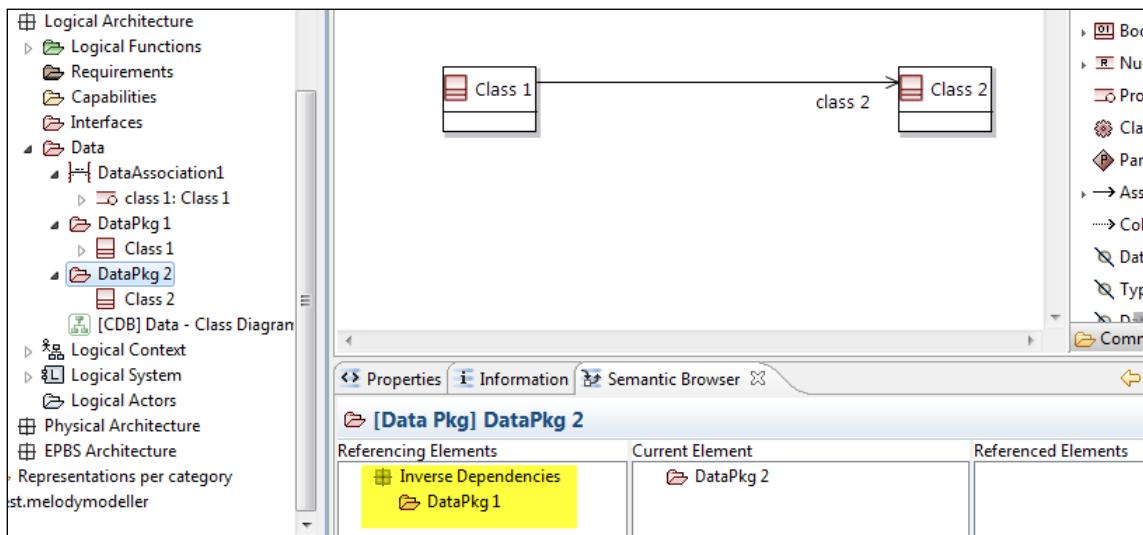
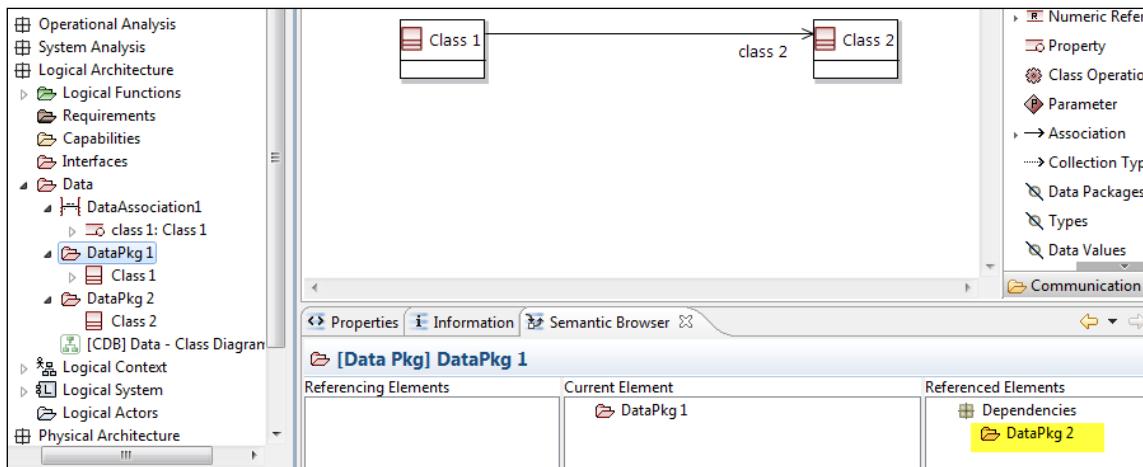
### 3.8.2 Dependencies

Data Packages should display high cohesion and low coupling. It is very important to avoid:

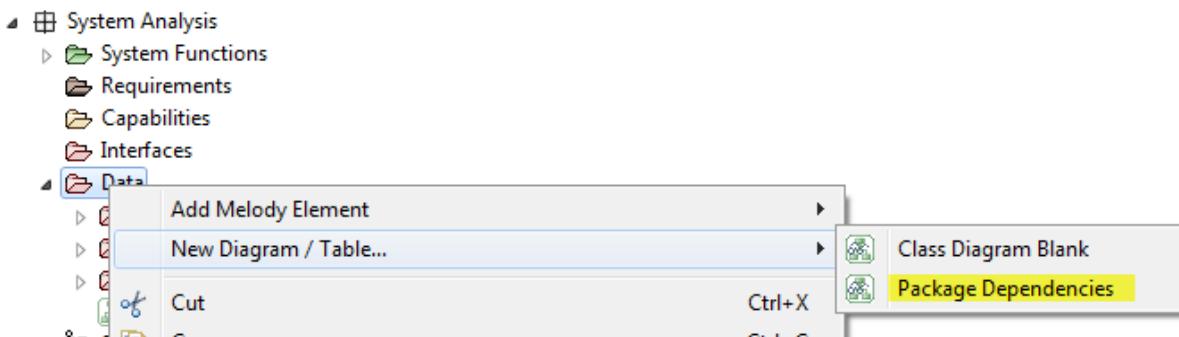
- self-dependencies
- circular dependencies

Package dependencies can be deduced from crossing navigable associations or generalizations between types (this is why it is already important to restrict the navigability of associations to only one direction), and also from the fact that properties are typed by elements belonging to another package.

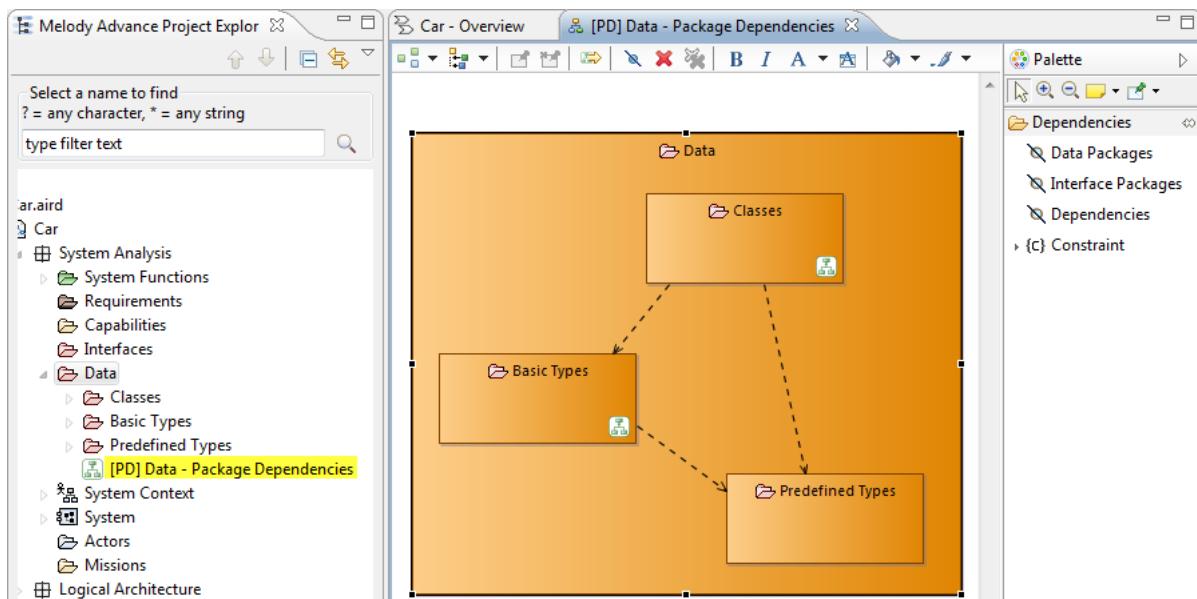
For instance, in the next figure, we see that Melody detects the dependency between packages, because an association exists between classes owned by these packages.



A very interesting feature of Melody is the ability to calculate and draw dependencies between packages based on associations and generalizations between owned Types.



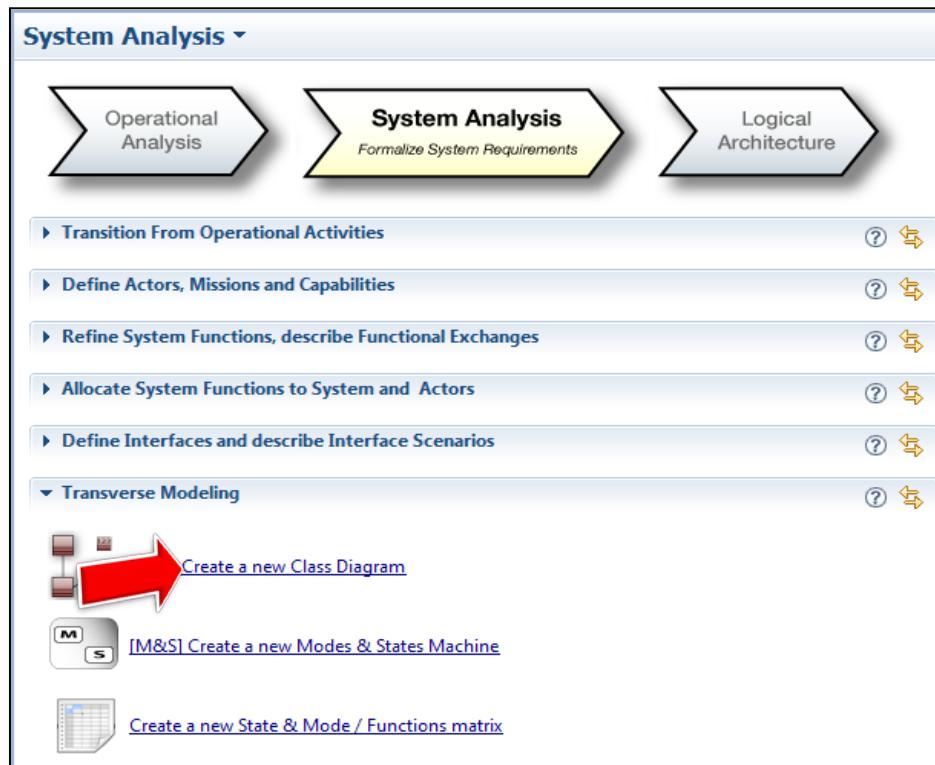
On our example, Melody was able to calculate that *Classes* depends on *Basic Types* and *Predefined Types* (properties of classes are typed by Basic Types and Predefined Types), and that *Basic Types* depends on *Predefined Types*. In the Package Dependencies diagram, if we insert all packages, the following diagram is automatically displayed:



We can check easily that there are no mutual dependencies, nor circular ones.

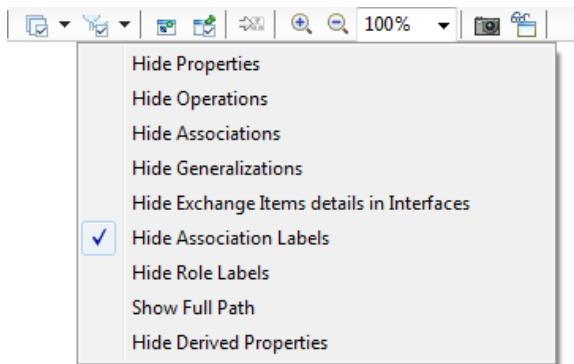
### 3.9 CDB (Class Diagram Blank)

All definitions of Classes, Types and Exchange items are done in Melody through Class Diagrams. These CDB (Class Diagram Blank) are available at each Arcadia abstraction level.



#### 3.9.1 Filters

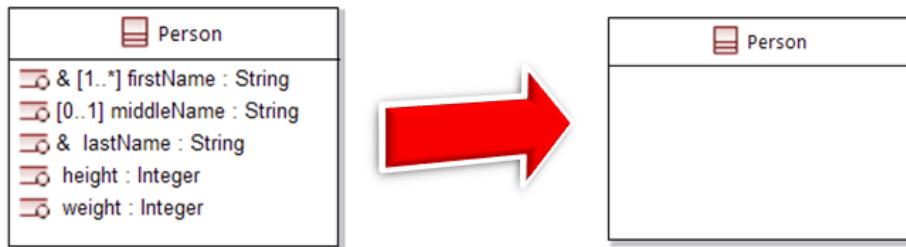
There are several filters available in the CDB.



Only the “Hide Association Labels” filter is activated by default. We have given examples of this filter and the following one: “Hide Role Labels” in the [Association Name §](#).

### 3.9.1.1 Hide Properties

This filter hides the properties inside classes. But it does not hide the literals inside the basic types.

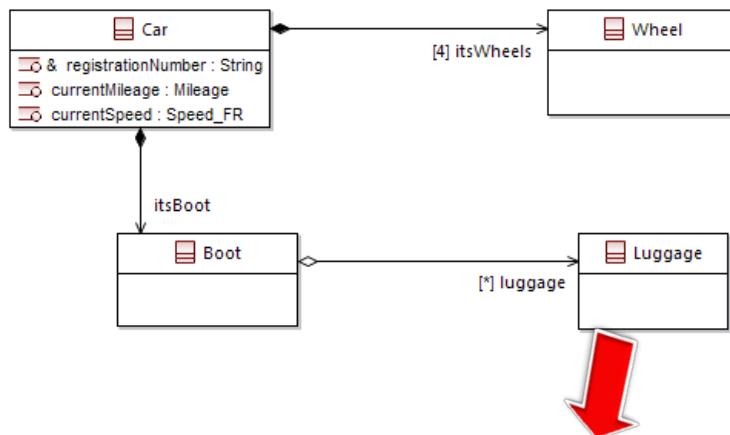


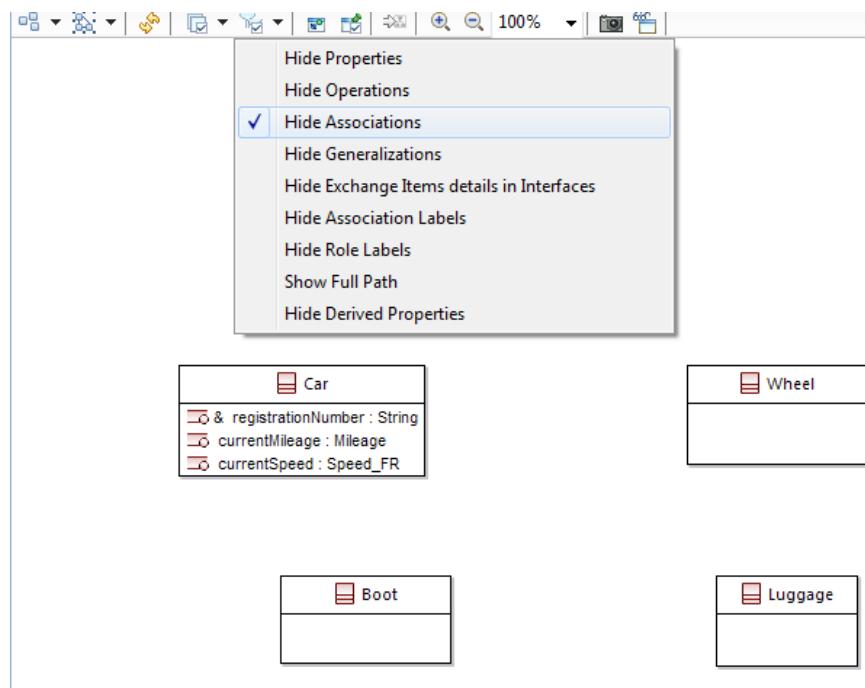
### 3.9.1.2 Hide Operations

This filter hides the operations inside classes. Remember we strongly advise not to use operations inside classes.

### 3.9.1.3 Hide Associations

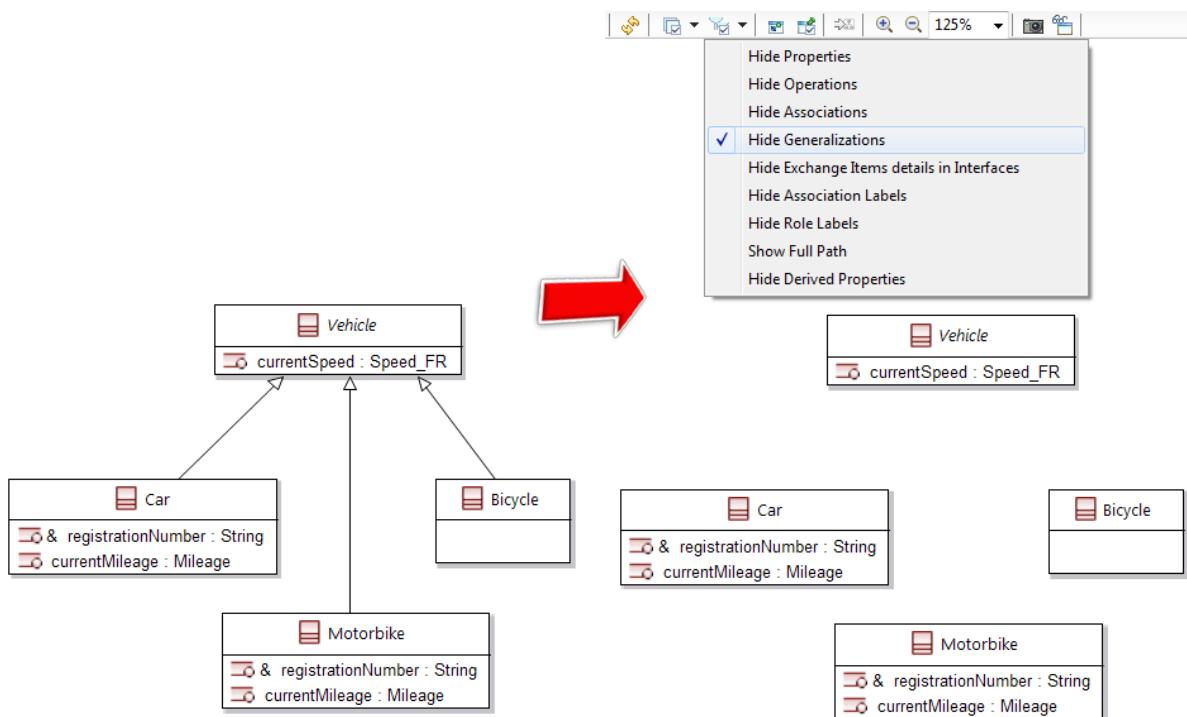
This filter hides the associations between classes, including aggregations and compositions.





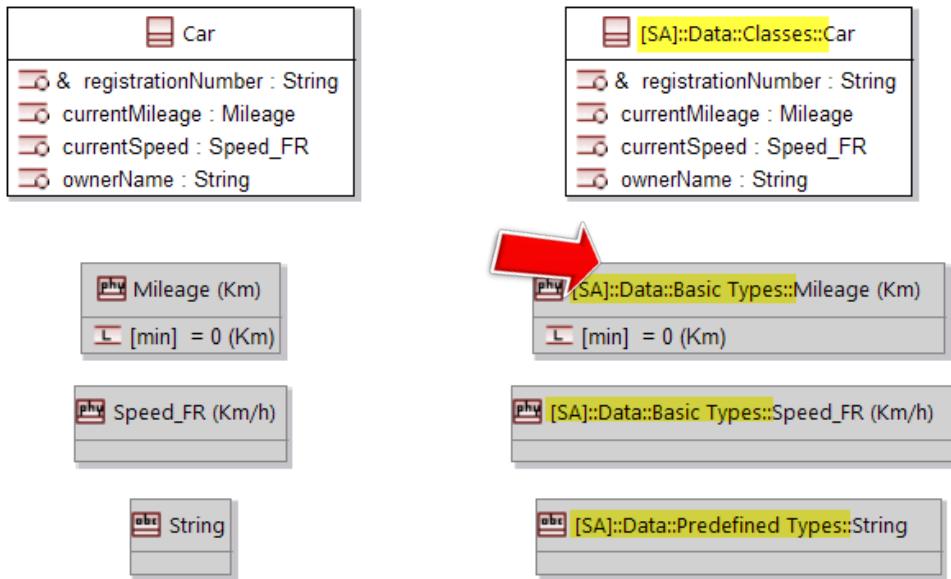
### 3.9.1.4 Hide Generalizations

This filter hides the generalizations between classes.



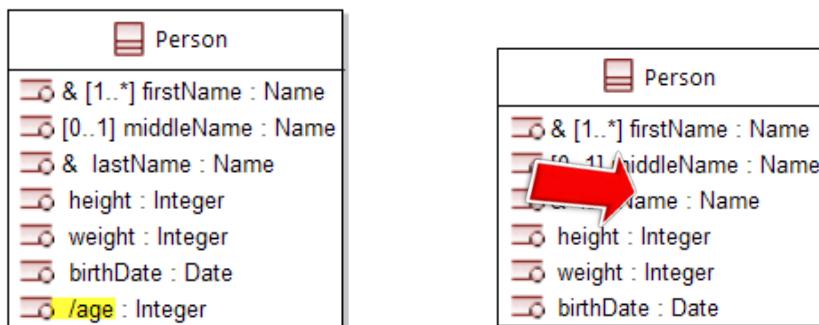
### 3.9.1.5 Show Full Path

This filter shows the full path of owning packages.



### 3.9.1.6 Hide Derived Properties

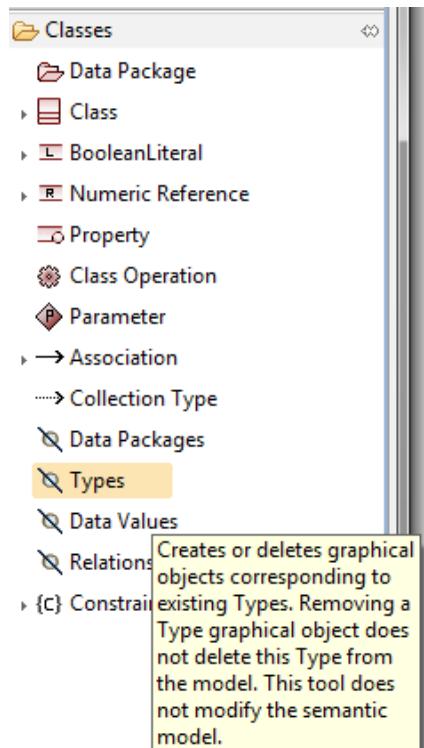
This filter hides the [derived](#) properties inside classes.



## 3.9.2 Toolbox

### 3.9.2.1 Insert / Remove Types

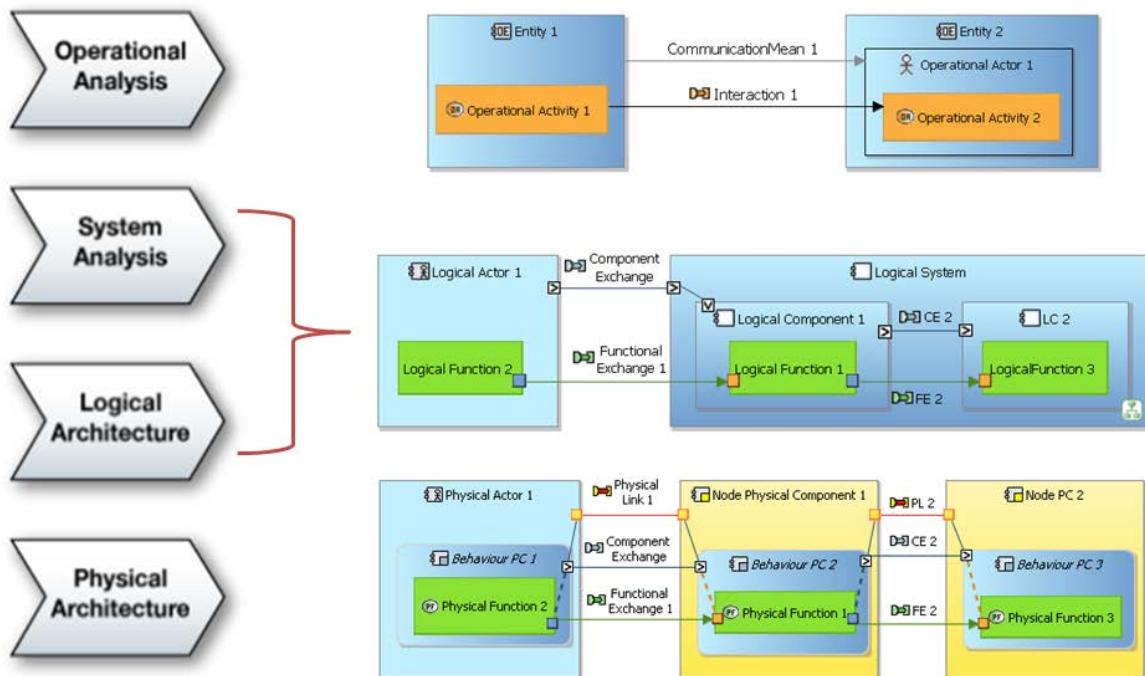
All Classes, Primitive Classes and Basic Types are managed through the same tool: Insert/Remove Types.



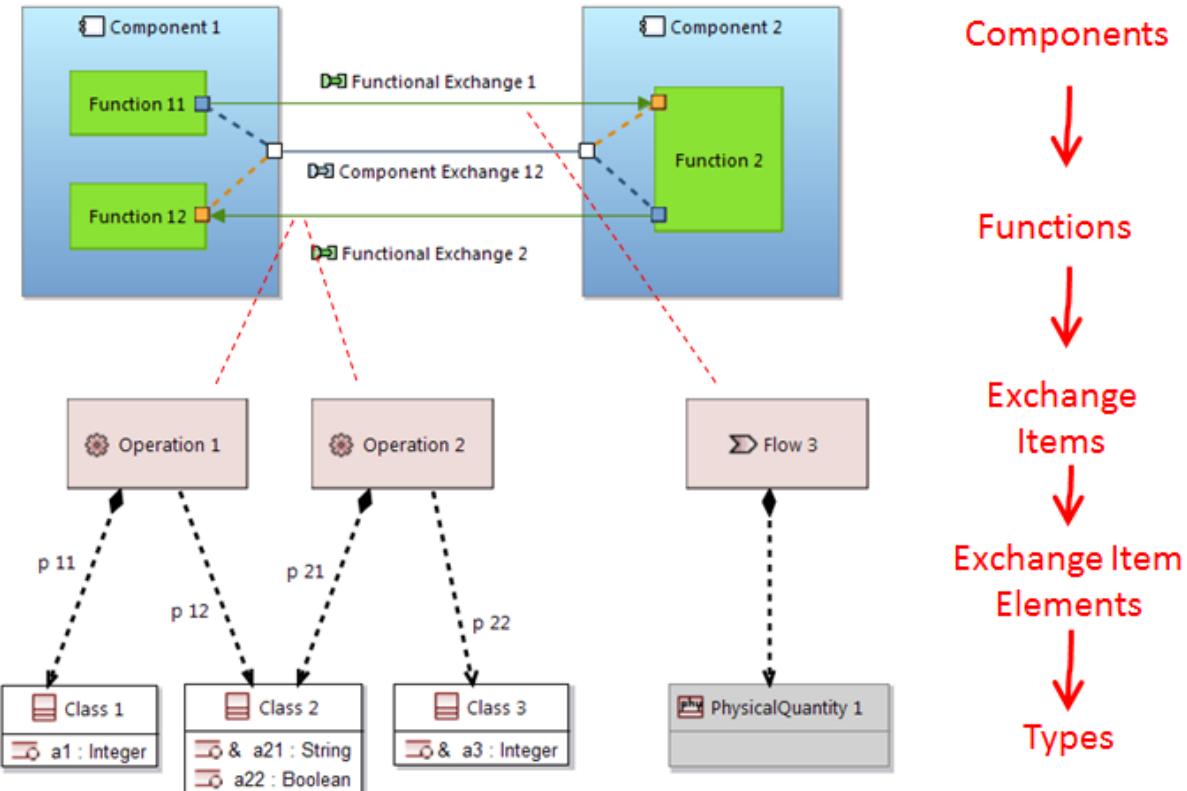
## 4 BASIC CONSTRUCTS : COMMUNICATION MODEL

### 4.1 Overview of Main Communication Concepts

The following figure gives a summary of Exchange concepts throughout the Arcadia abstraction levels:



- Data to be allocated to a port or exchange are in fact allocated to it through one or more Exchange items.



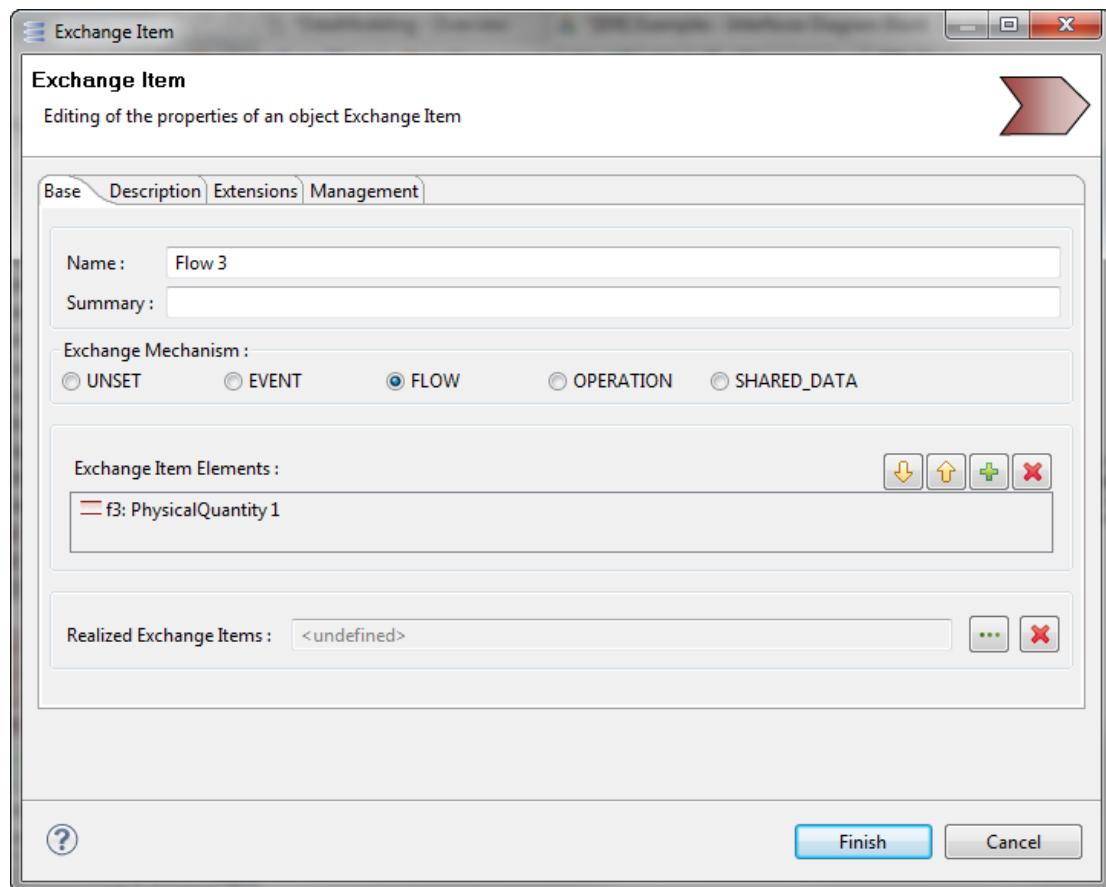
## 4.2 Exchange Item

An Exchange Item defines a communication media and a set of data

- semantically coherent with regards to their use in a given context
- used as a whole for functional exchanges:
  - same communication principles
  - simultaneity of transportation
  - same non-functional properties (e.g. security level, integrity requirement, expected performance...)
  - indivisibility (an exchange item is atomic)
- referring to kinds of data described in the data model.

Exchange Items help in:

- structuring ports/exchanges in order to organize exchanged data
- avoiding to define several times the same set of data when used in several ports/exchanges
- imposing a unification of interactions or exchanges of the same kind everywhere in the system (e.g. standard for messaging, protocols, ...)
- constituting a checkable contract in exchanges with outside the system
- carrying a requested communication principle if necessary (e.g. event, dataflow, message, service, shared data, continuous flow...).



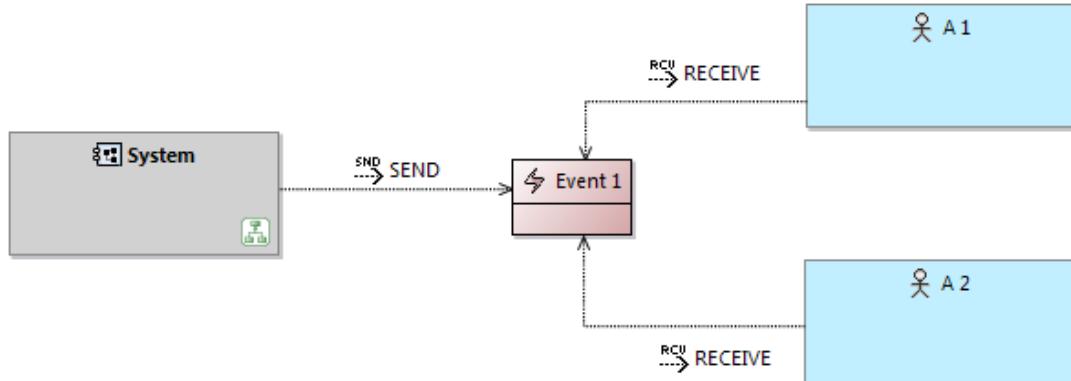
Field	Description	Default
Name	The name of the current Element.	<generic name>
Summary	A short headline about the role of the current Element.	<blank>
Exchange Mechanism	Data are exchanged between Functions or Components according to standard communication patterns. An Exchange Item therefore carries the communication mechanism specifying how its content is actually exchanged. Possible communication mechanisms for Exchange Items are: OPERATION, EVENT, FLOW, SHARED DATA. In the early stages of the engineering process, the communication mechanism applying to a given Exchange Item might not be known yet. In that case, UNSET can be used.	UNSET
Exchange Item Elements	One or several references to elements from the Data Model (Data types, Classes, Unions, Collections, etc.). Adding references towards Data elements from this widget creates Exchange Item Elements and displays a dedicated editor allowing setting the properties of the newly created Exchange Item Element (name, type, cardinality, direction).	<blank>
Realized Exchange Items	One or several Exchange Items in the previous engineering phase refined by the current Exchange Item.	False

### 4.2.1 Event

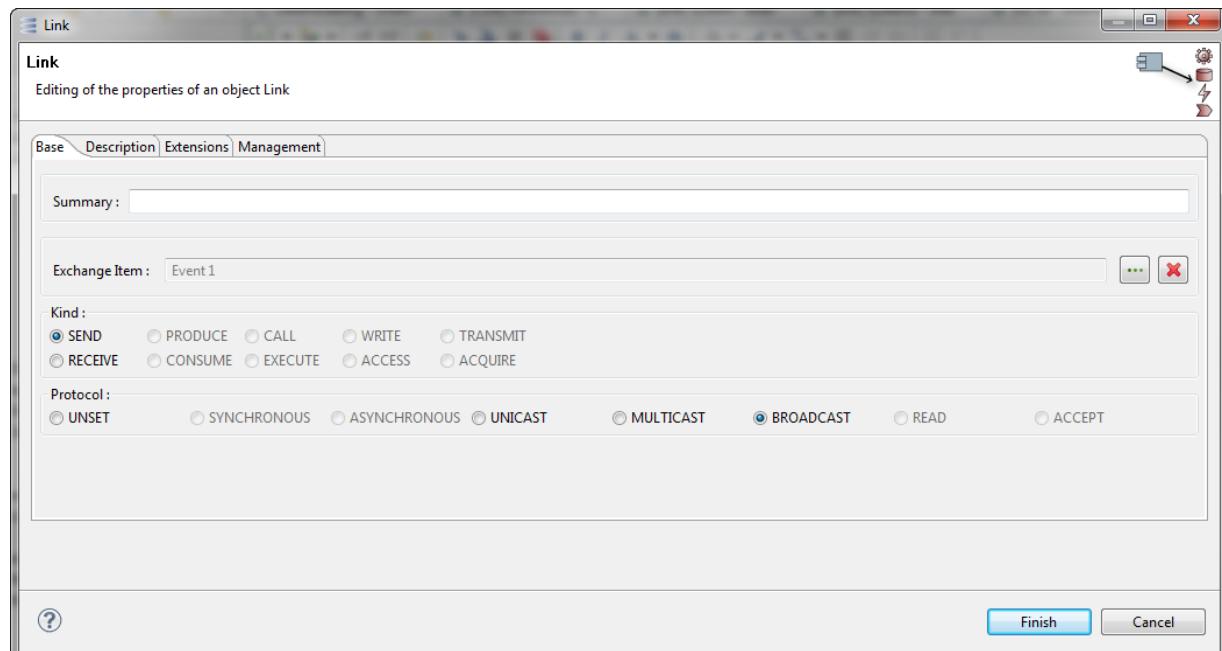
Exchange Item with a communication mechanism set to EVENT.

Structured Data sent from one Component / Function and received by a unique receiver (unicast), a defined set of receivers (multicast) or an undefined set of receivers (broadcast).

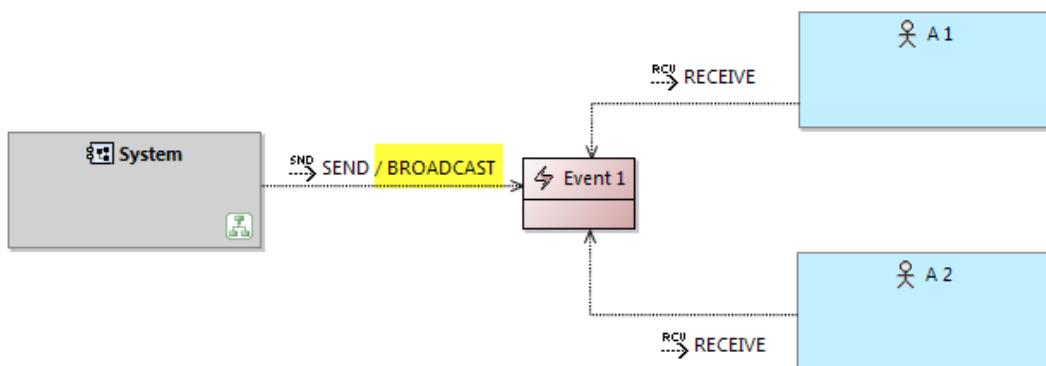
It is possible to represent explicitly the Event in an IDB, along with Communication Links.



Using the Communication Link editor, it is possible to specify the SEND protocol. Here, BROADCAST is selected.



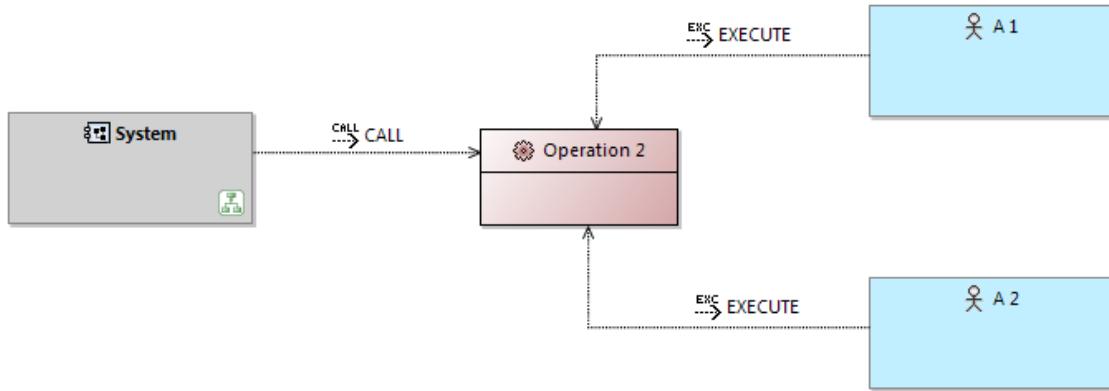
The Communication Link label on the Interface diagram is then updated.



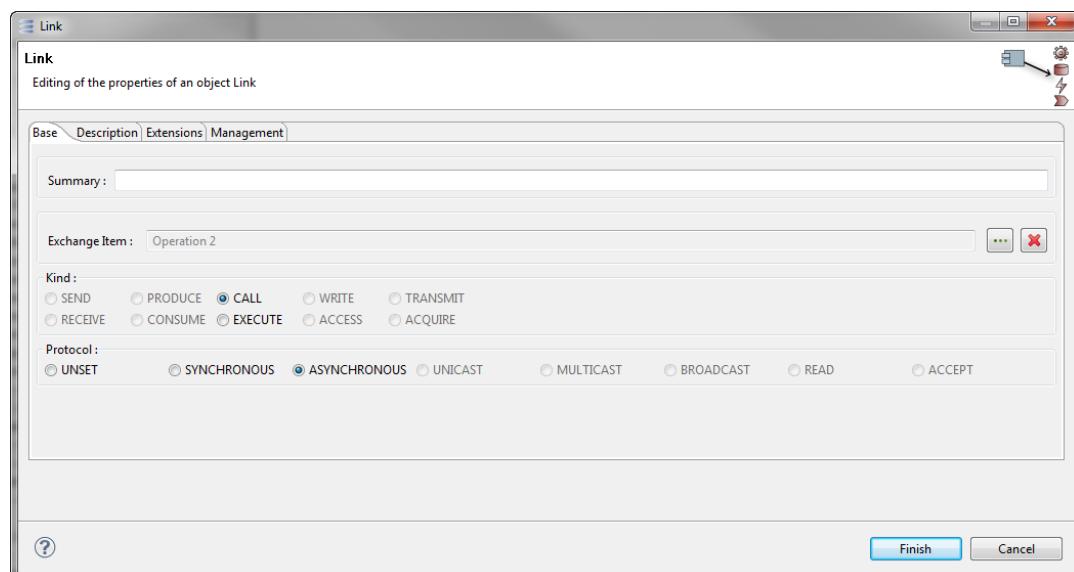
### 4.2.2 Operation

Exchange Item with a communication mechanism set to OPERATION.  
 Piece of behavior with input and output parameters, executed by one Component / Function and requested by others. Operations can be synchronous or asynchronous.

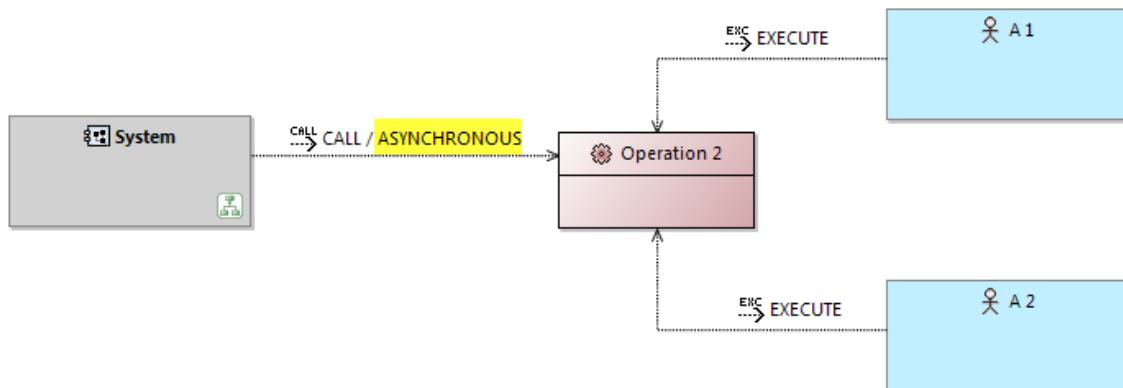
It is possible to represent explicitly the Operation in an IDB, along with Communication Links.



Using the Communication Link editor, it is possible to specify the CALL protocol. Here, ASYNCHRONOUS is selected.



The Communication Link label on the Interface diagram is then updated.

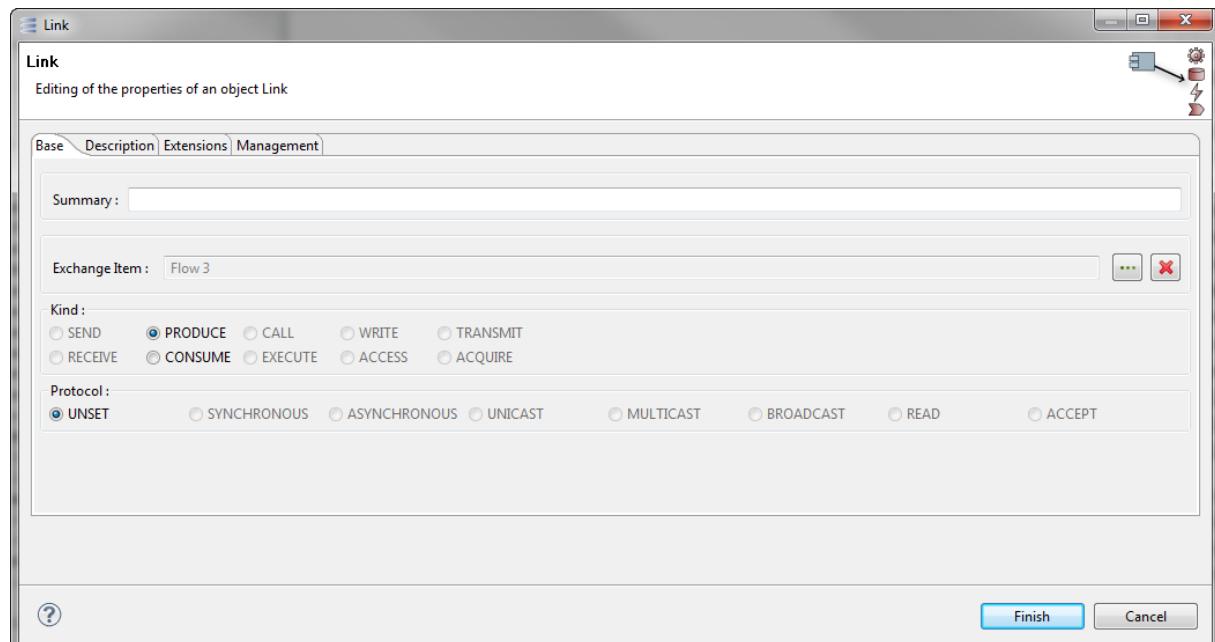
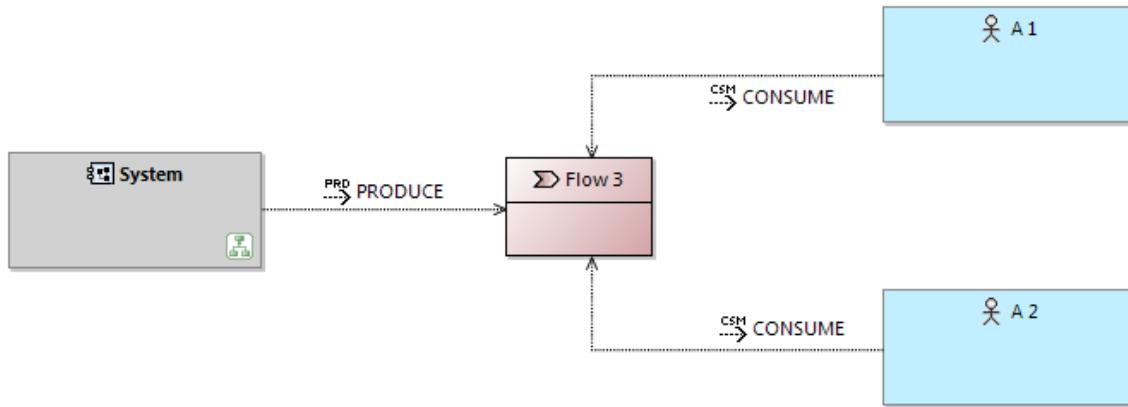


### 4.2.3 Flow

Exchange Item with a communication mechanism set to FLOW.

Continuous or discrete Flow of information / material (fluid, electric power, calories, etc.) produced by a Component / Function and consumed by one or several others.

It is possible to represent explicitly the Flow in an IDB, along with Communication Links.



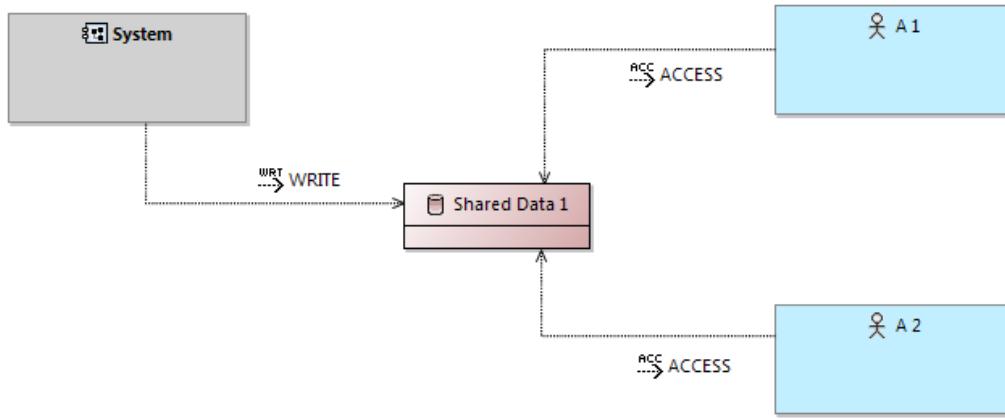
Using the Communication Link editor, it is not possible to specify the PRODUCE protocol, contrarily to the other kinds of Exchange Items.

### 4.2.4 Shared Data

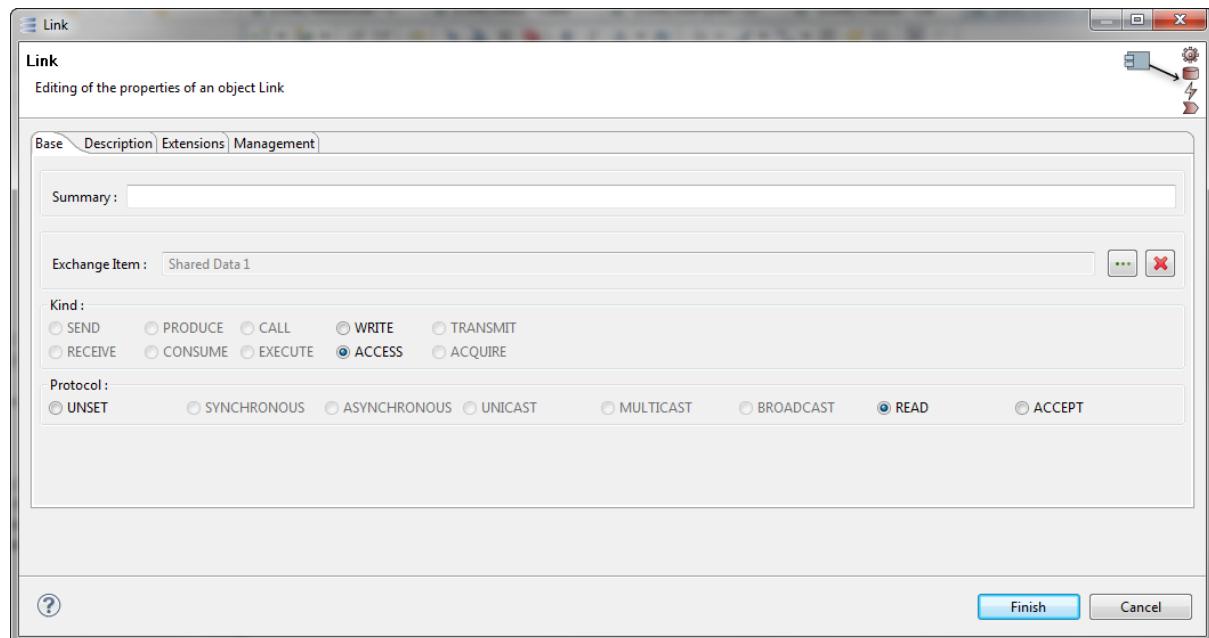
The data referenced by the Exchange Item is:

- Either shared and concurrently accessed by several Components / Functions (writers or readers);
- Either exchanged through a Publish/Subscribe mechanism.

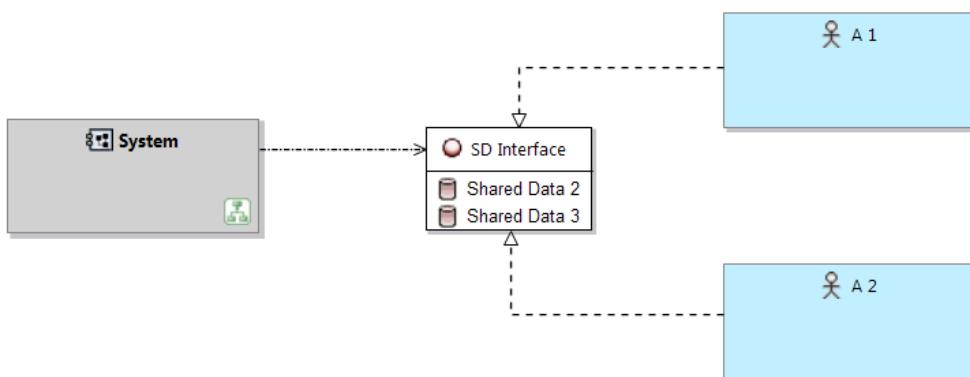
It is possible to represent explicitly the Shared Data in an IDB, along with Communication Links.



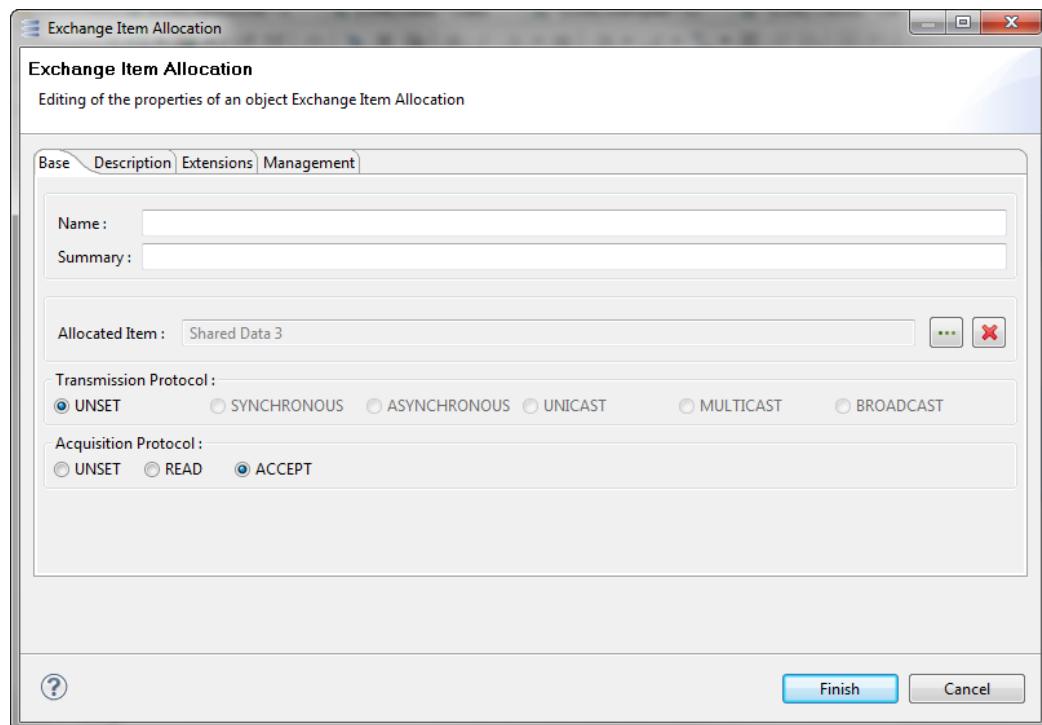
Using the Communication Link editor, it is possible to specify the ACCESS protocol. Here, READ is selected.



The Shared Data can also be grouped in an Interface.

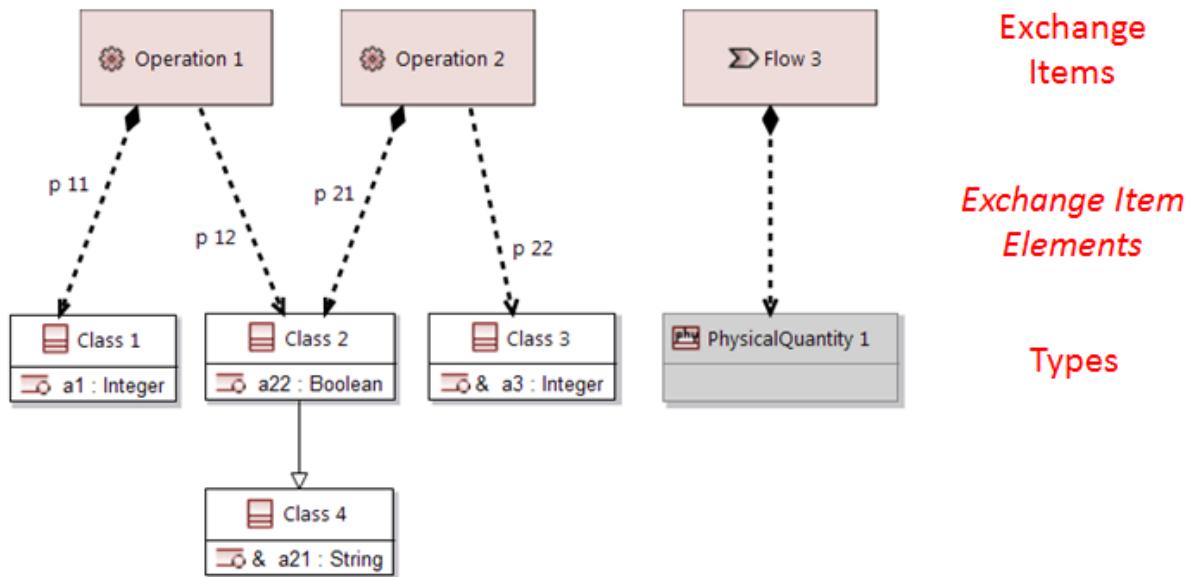


It is also possible to change the ACCESS protocol using the editor of Exchange Item Allocations. Here, ACCEPT is selected.



### 4.3 Exchange Item Element

Exchange Items are structured through Exchange Items Elements in the same way as classes are structured in Properties. These elements are in turn defined by classes, complex types and simple types.

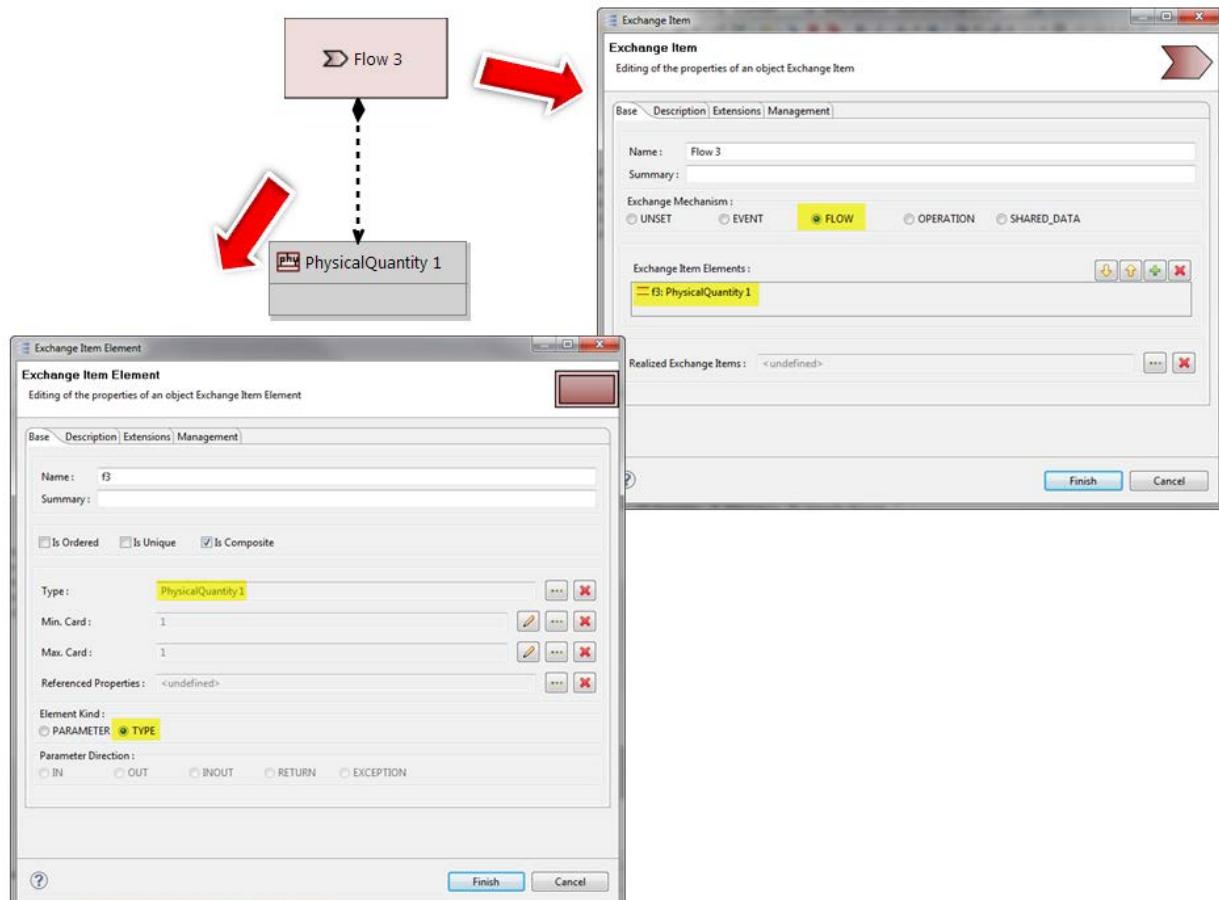


An Exchange Item Element is either of Kind PARAMETER or TYPE.

It has the following features:

- Is composite. **This feature specifies that the Exchange Item containing the element is** a container for the object or value contained in the element (Default is true). If false, the target class shall own or inherit a key (see for example, p12 and p22 in the previous figure).

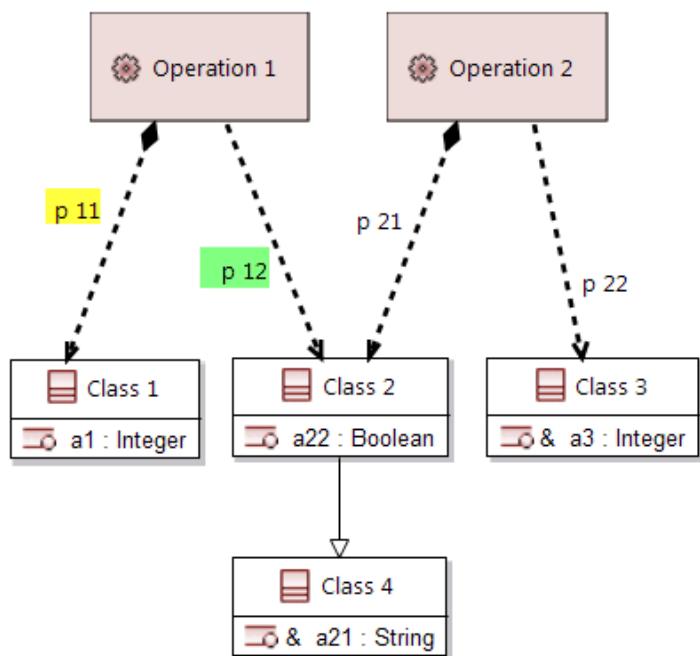
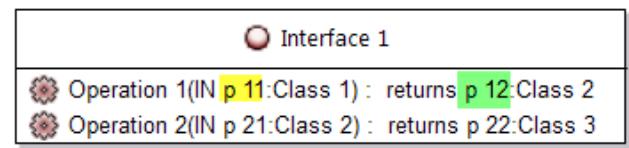
- Is ordered.** For a multivalued multiplicity (cardinality max > 1), this feature specifies whether the values in an instantiation of this element are sequentially ordered (Default is false).
- Is unique.** For a multivalued multiplicity (cardinality max > 1), this attribute specifies whether the values in an instantiation of this element are unique (Default is true).

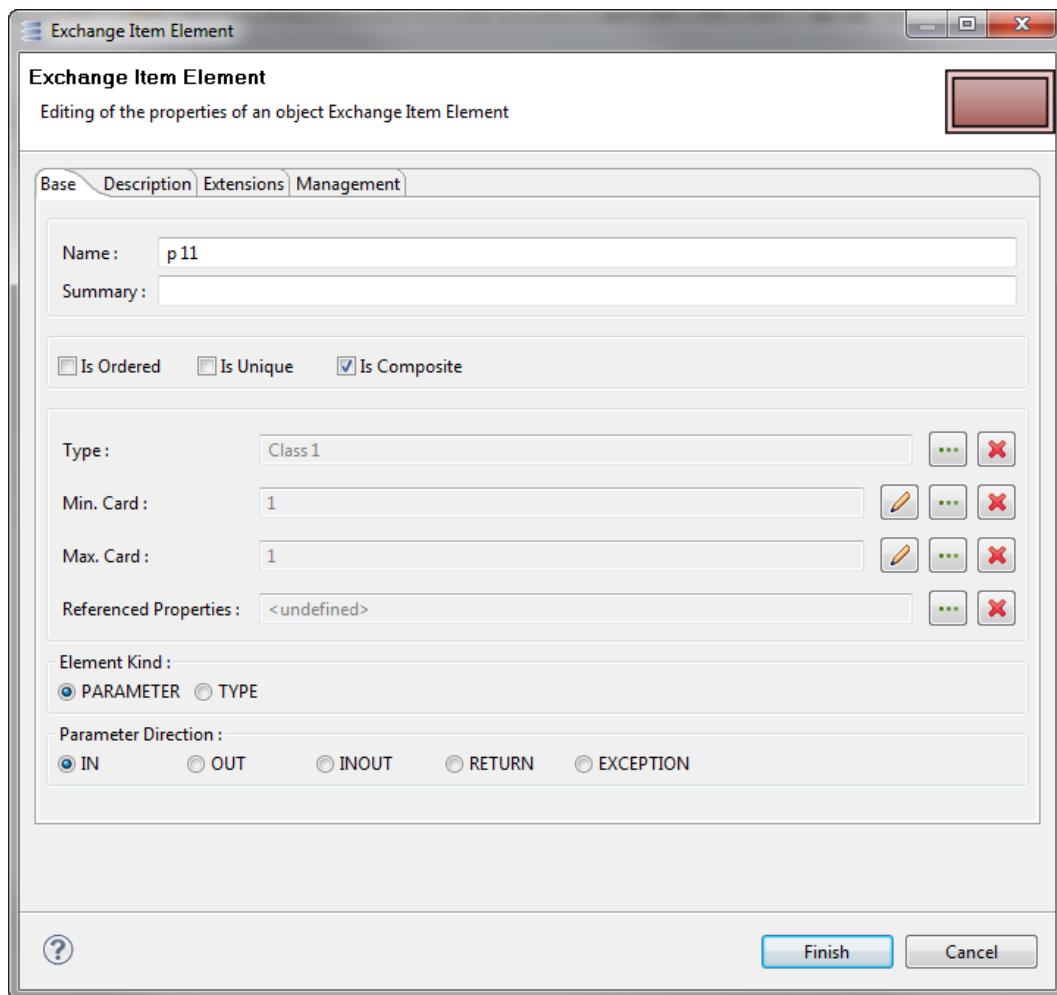


When an Exchange Item has a communication mechanism of type OPERATION, its Exchange Item Elements are mandatorily of type PARAMETER, and are used to define the parameters of this operation. It is also mandatory then to specify each parameter direction.

Element Kind :	<input checked="" type="radio"/> PARAMETER <input type="radio"/> TYPE
Parameter Direction :	<input checked="" type="radio"/> IN <input type="radio"/> OUT <input type="radio"/> INOUT <input type="radio"/> RETURN <input type="radio"/> EXCEPTION

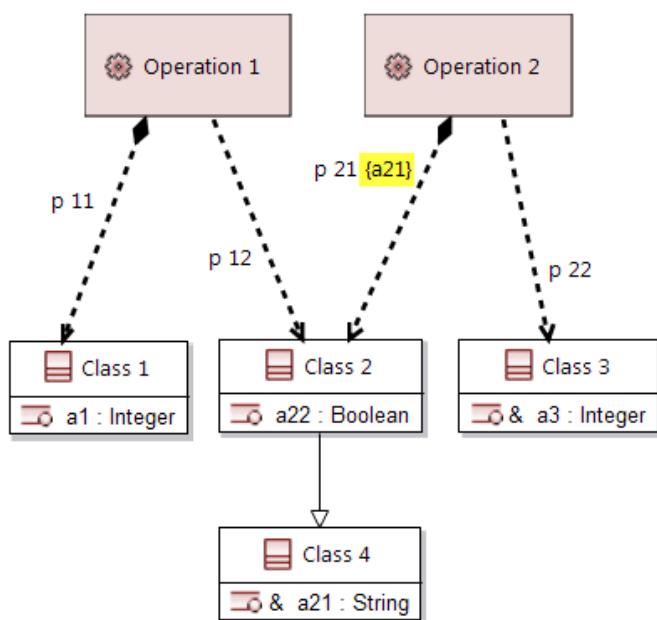
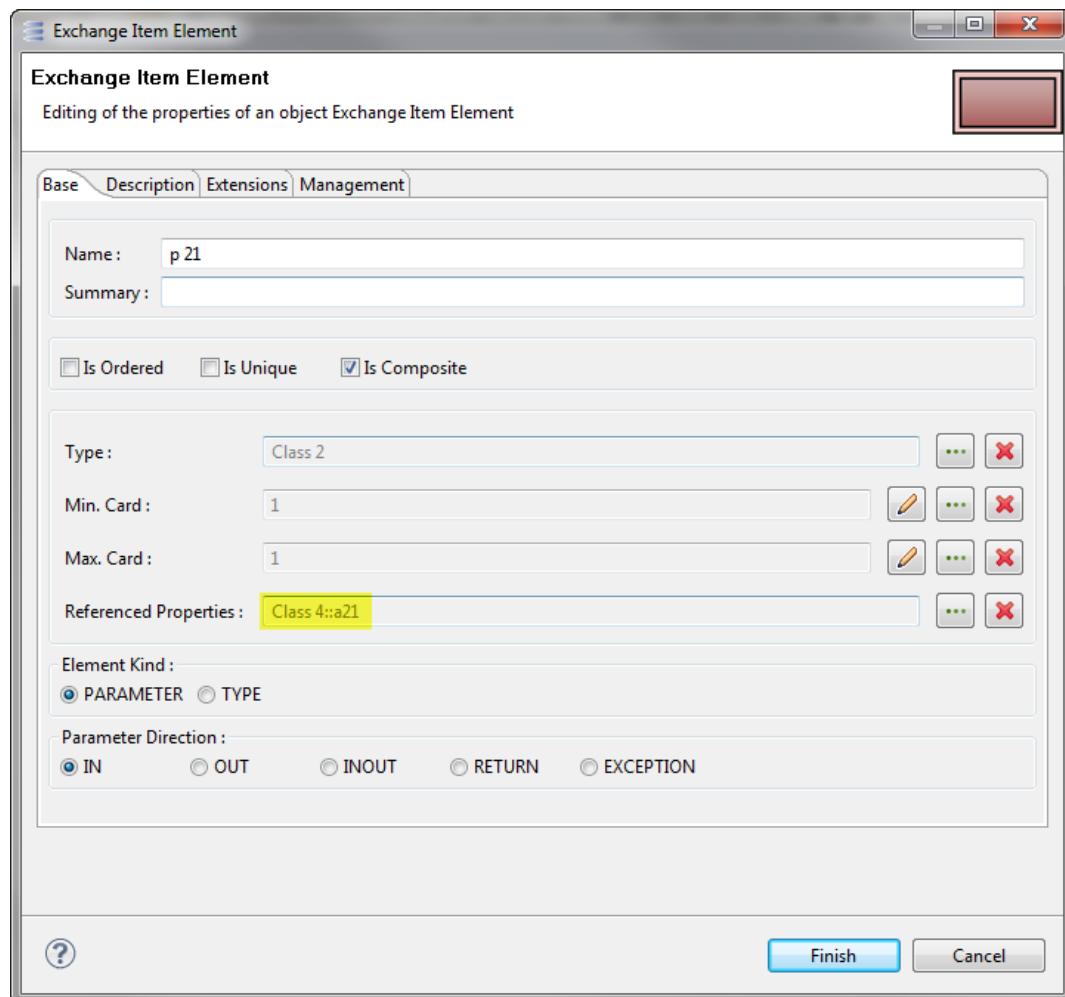
Naming Exchange Item Elements is mandatory when kind = PARAMETER.





Note that for p11, isComposite must be true as Class1 does not have a key property, but p12 can be a reference (isComposite = False), as Class2 inherits its key from Class4.

Observe also that it is possible to restrain the Exchange Item Element to a subset of the Class (or data type) properties. In our example, parameter p 21 of Operation 2 could reference only property Class4::a21 through the "Referenced Properties" field.



When an Exchange Item has a communication mechanism of type EVENT, FLOW, or SHARED \_DATA, its Exchange Item Elements are either of type PARAMETER or TYPE,

If the Exchange Item Element is of Kind TYPE, the Exchange Item only owns this Exchange Item Element, and the structure of the Exchange Item is completely defined by the type of

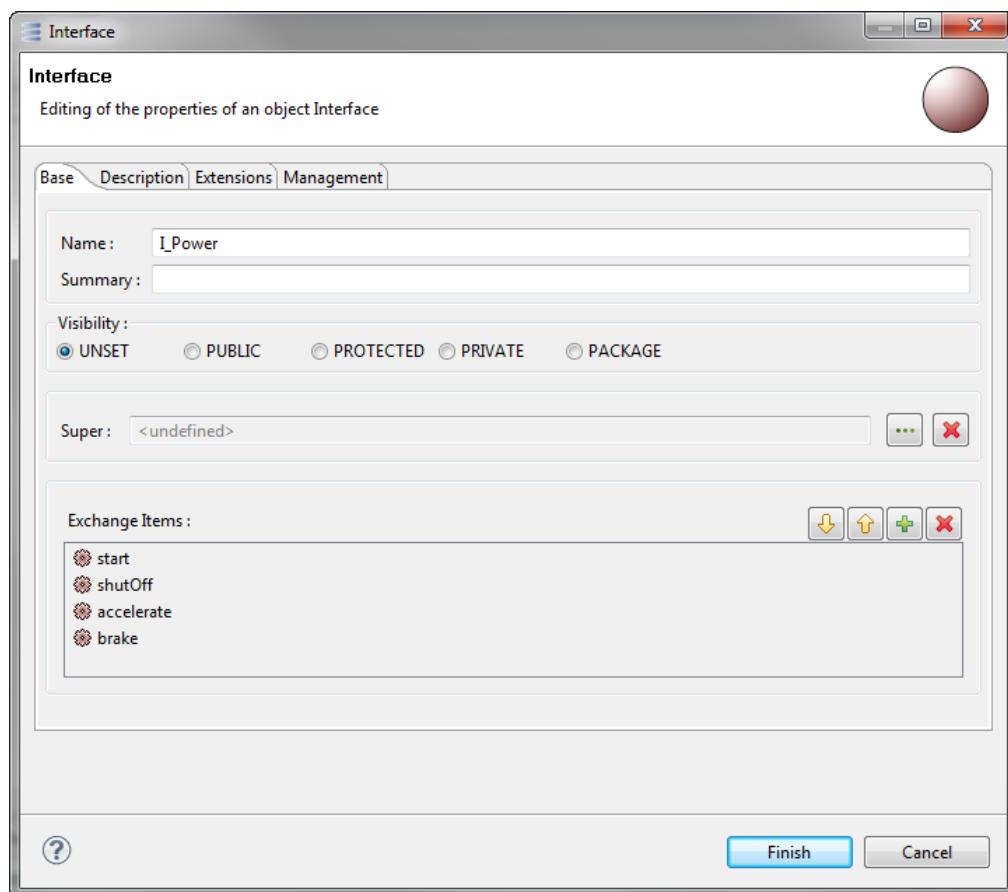
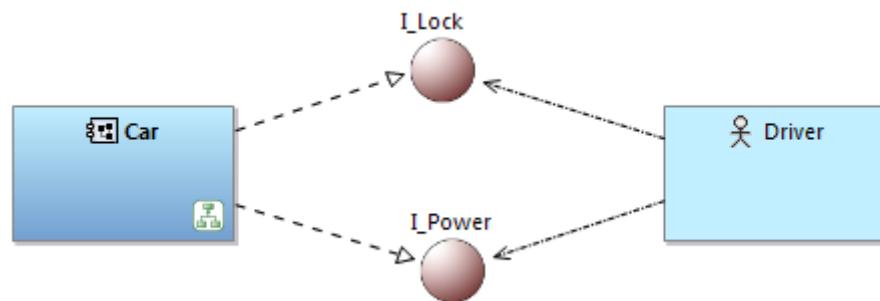
the Exchange Item Element, and its features: cardinality, 'is composite', 'is ordered' and 'is unique'.

Else, all its Exchange Item Elements are of type PARAMETER, and each Exchange Item Element enables to define a structural property of the owning Exchange Item, in the same way as class [properties](#) define structural properties of a class.

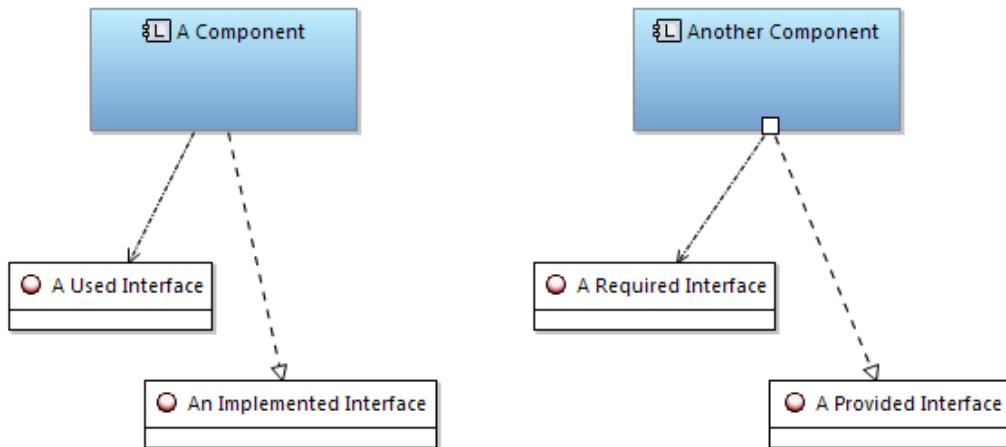
In a future version of Melody, PARAMETER will be renamed (the terms "FEATURE" or "MEMBER" are candidates), which is more general (PARAMETER is a very good name for Exchange Items of type OPERATION, but much less for the other types).

## 4.4 Interface

Interfaces are contracts specifying how components can interact with each other. Interfaces are defined by grouping Exchange Items.



Interfaces are implemented/provided or used/required by Components/ Components Ports. Each Component/ Component Port can implement - provide / use - require several Interfaces.

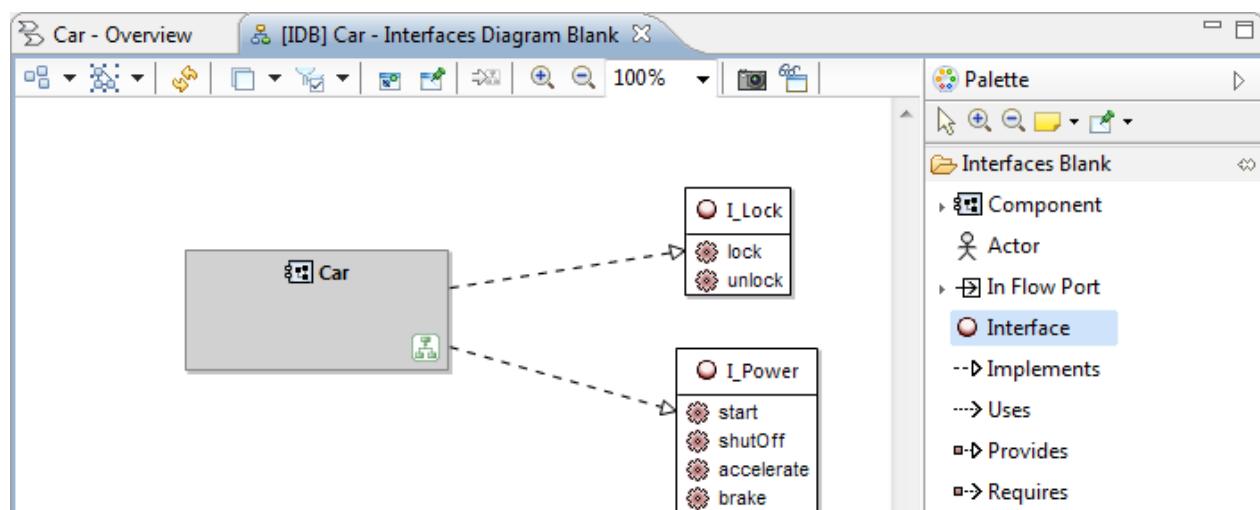


Interface building rules are:

- An interface can reference several exchange items (this may help in hiding complexity if necessary, details being accessible at functional level);
- An exchange item can be referenced by several interfaces;
- Component interfaces can be defined without reference to an exhaustive functional definition: by this way, a component can be defined (or reused) even if only partial or no functional contents are available.

#### 4.4.1 Interface Definition Blank (IDB) diagram

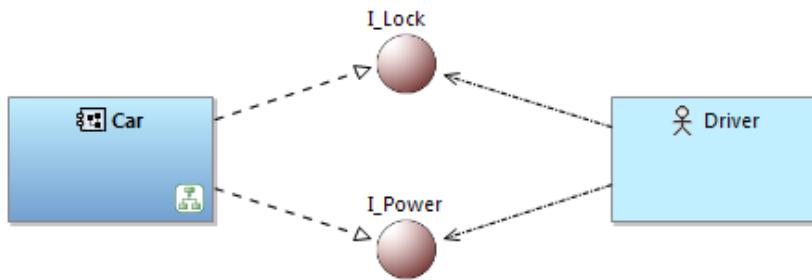
An Interface Definition Blank (IDB) enables to create (/insert) Components, Ports, Interfaces, etc.



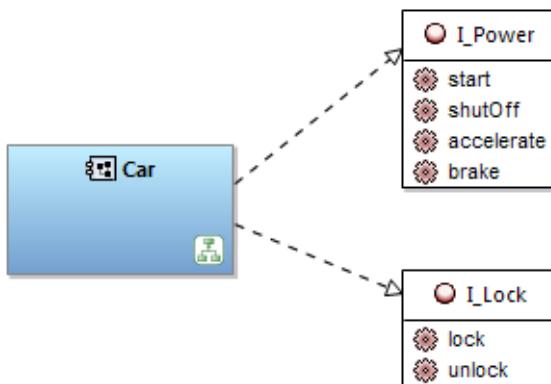
#### 4.4.2 CEI and CDI diagrams

These two related diagrams are contextual diagrams, created from one specific Component (or from the System at System Analysis level).

The Contextual Component External Interfaces (CEI) diagram shows all the Interfaces of a Component ( / System) as well as all the relationship of these Interfaces with other Components / Actors.



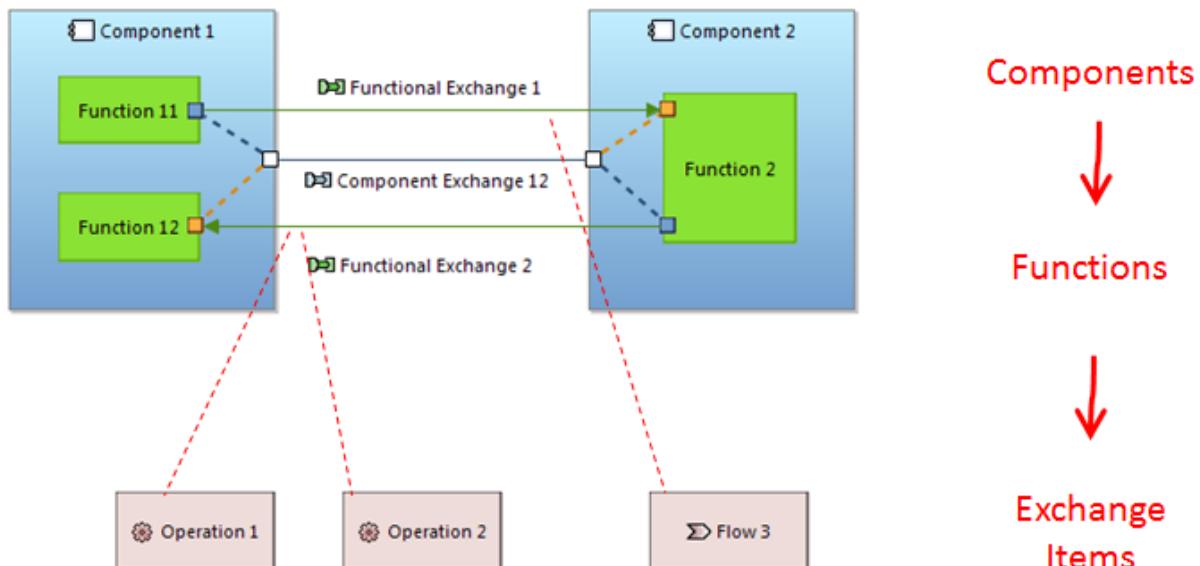
The Contextual Component Detailed Interfaces (CDI) diagram shows all the Interfaces of a Component as well as their contained Exchange Items.



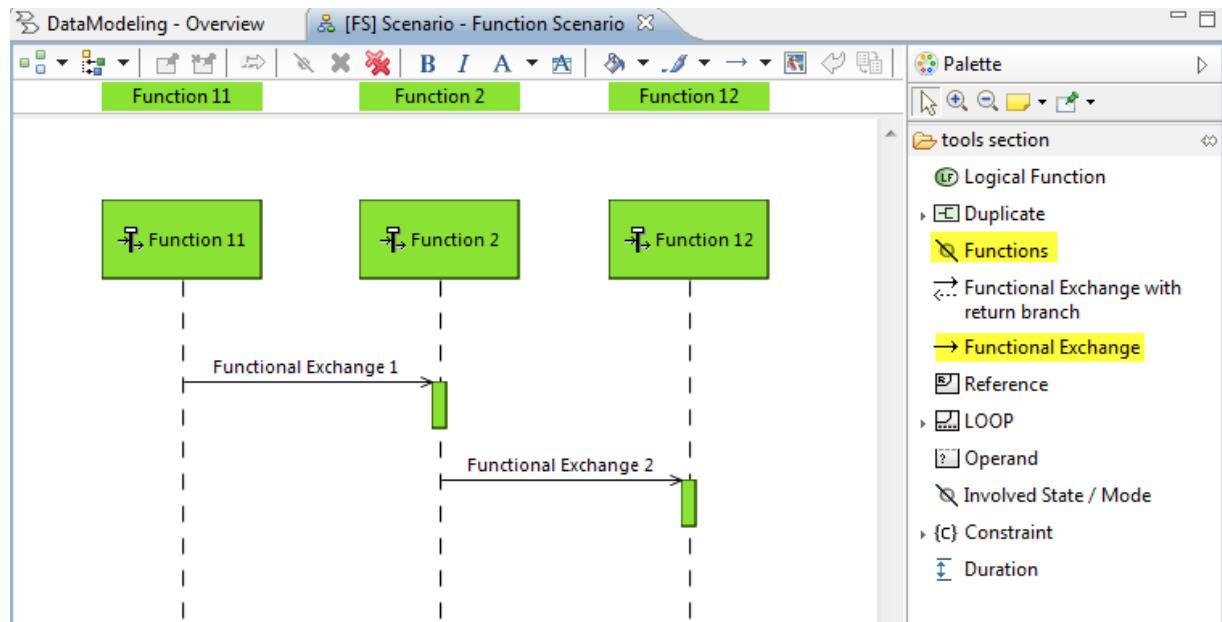
Take care! Even if they are Contextual, these diagrams are correctly updated when you add a relationship to an Interface in the model, but not completely if you remove a relationship with one Interface. You will have to clean the diagram by removing the disconnected Interface.

#### **4.4.3 Interface Scenario (Vs. Functional Scenario and Exchange Scenario)**

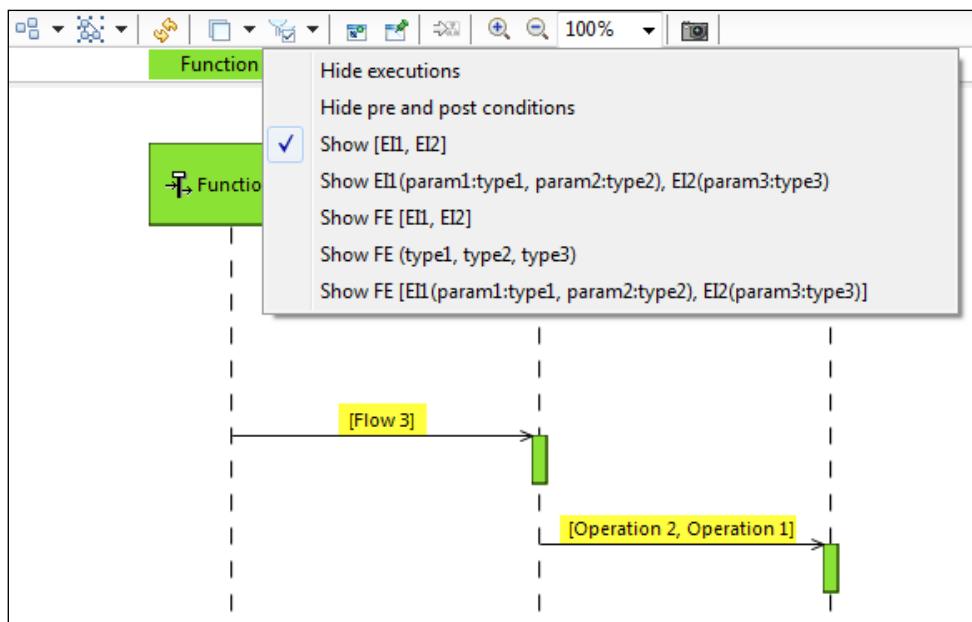
To illustrate the differences between the three different kinds of Scenarios, let us consider again the simple architecture represented by the following diagram:



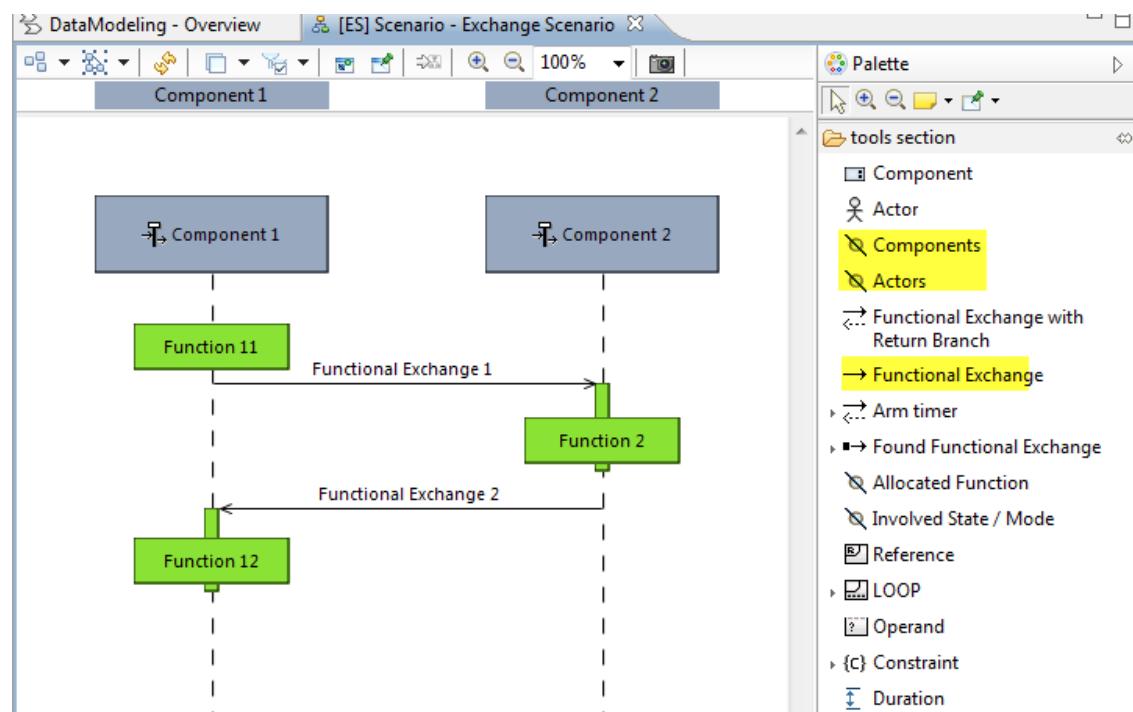
A Functional Scenario enables to draw sequence messages representing Functional Exchanges between lifelines representing Functions.



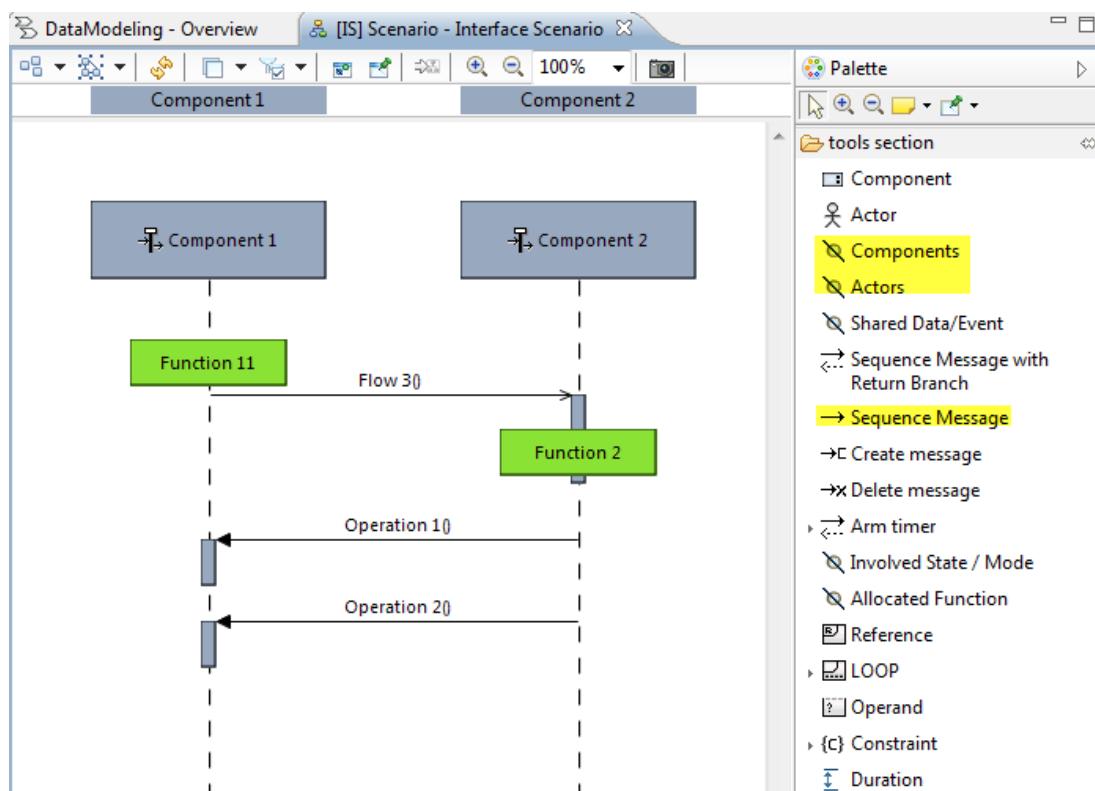
It is nevertheless possible to show directly the conveyed Exchange Items instead of the Functional Exchanges on the diagram, by applying a specific filter.



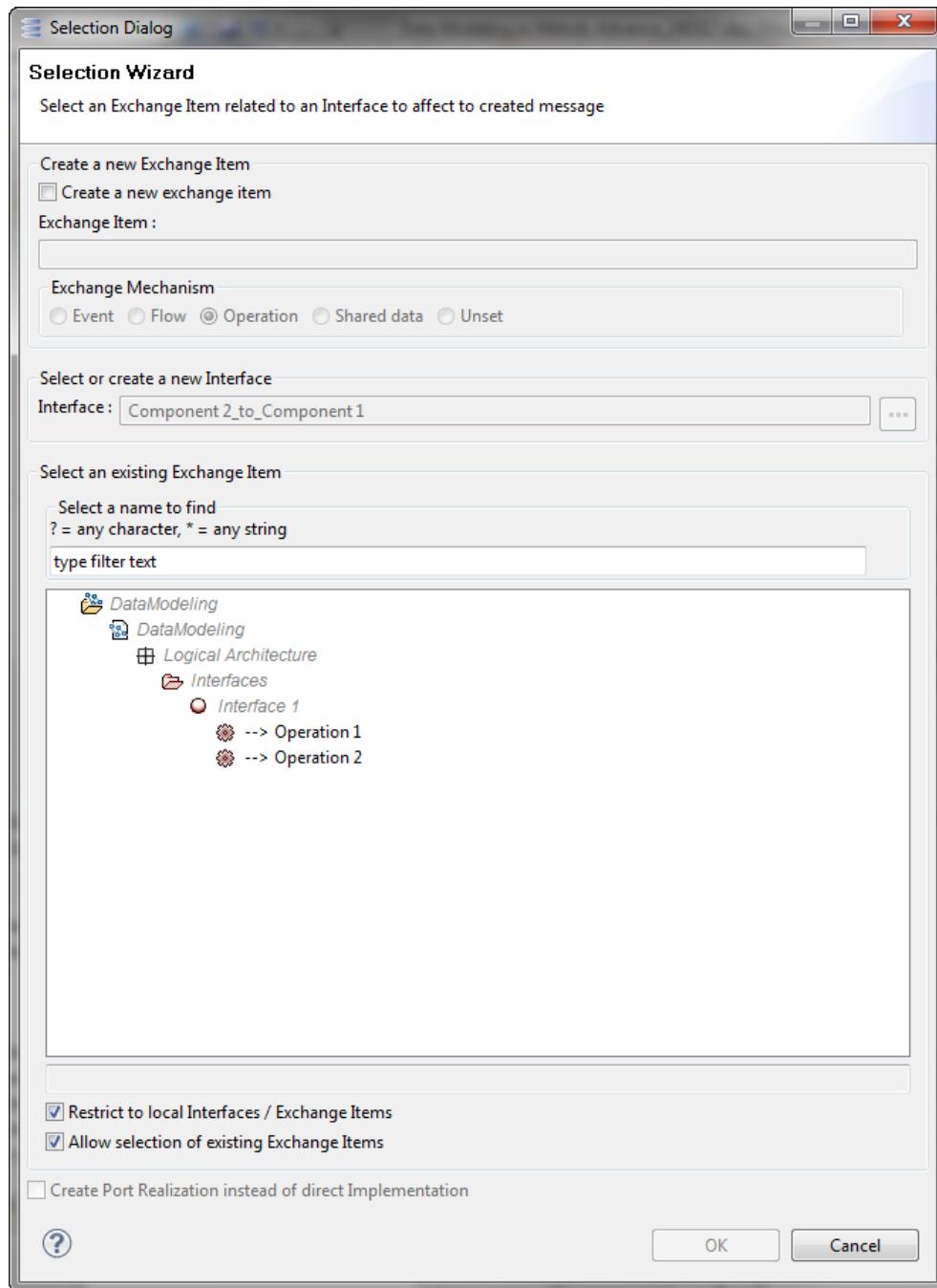
An Exchange Interface Scenario enables to draw sequence messages representing Functional or Component Exchanges between lifelines representing Components. It is also possible to see the conveyed Exchange Items by applying the same filter.



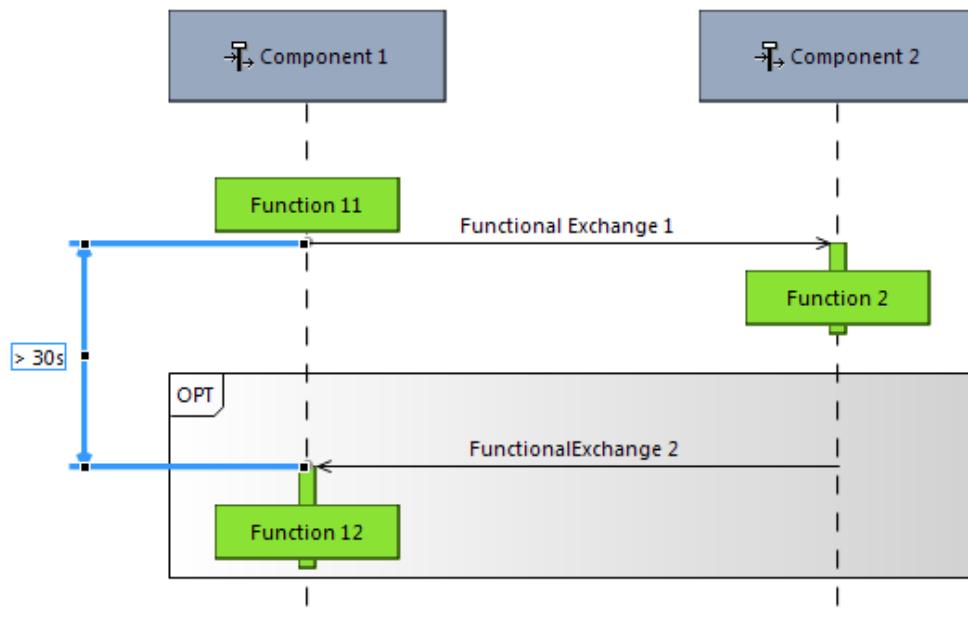
An Interface Scenario enables to draw sequence messages representing directly Exchange Items between lifelines representing also Components.



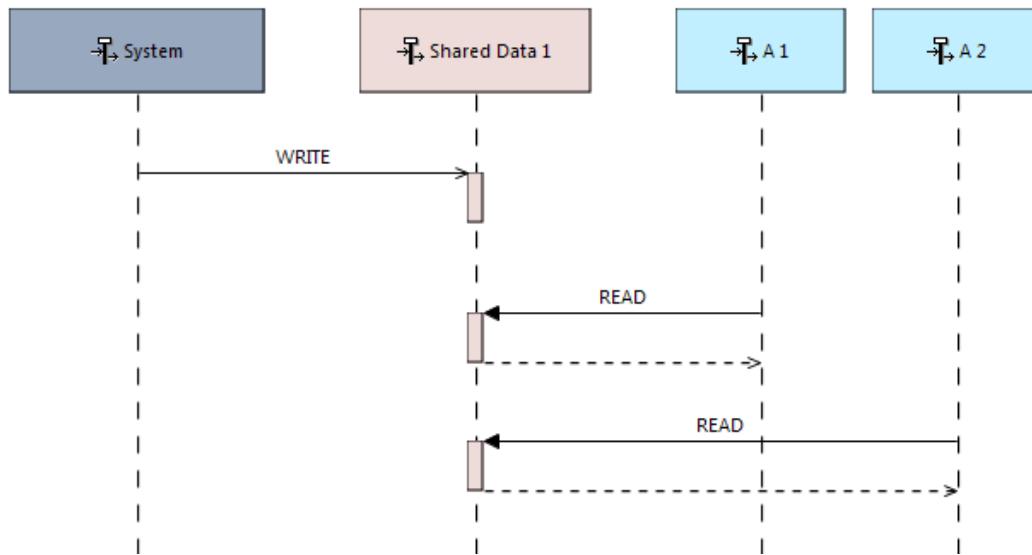
Note that it is possible to create new Exchange Items on the fly, and even new Interfaces, instead of selecting existing ones.



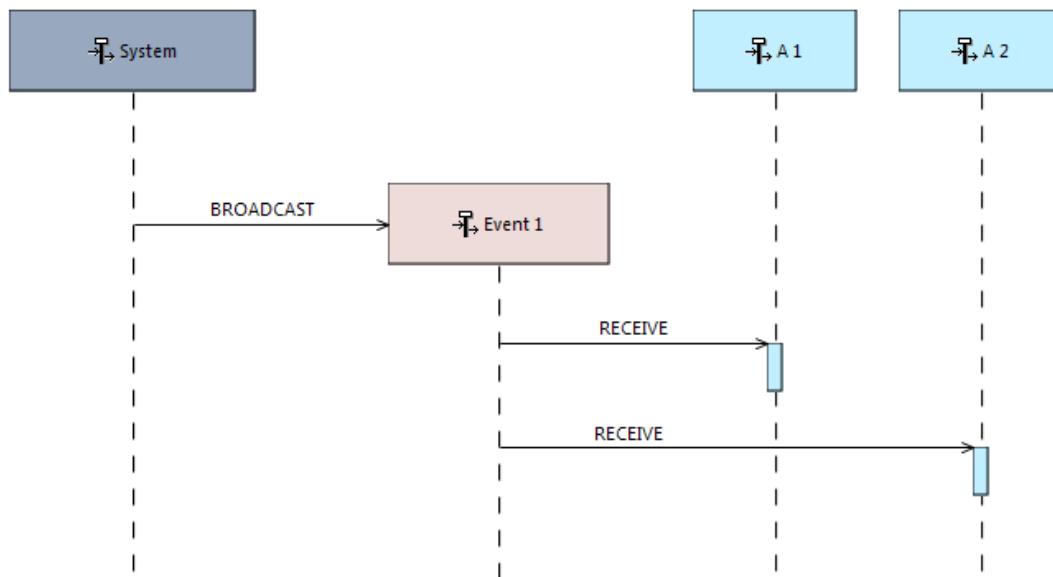
A big difference between Scenario and Data Flow Blank diagrams is that you can specify duration constraints on scenarios, and also loops, alternatives or optional conditions with the “combined fragment” concept.



Note that it is also possible to display [Shared Data](#) as Lifelines in Interface Scenarios, to explicitly show the READ / WRITE access to them.



The same is true for the [Events](#).



## 5 ADVANCED CONSTRUCTS

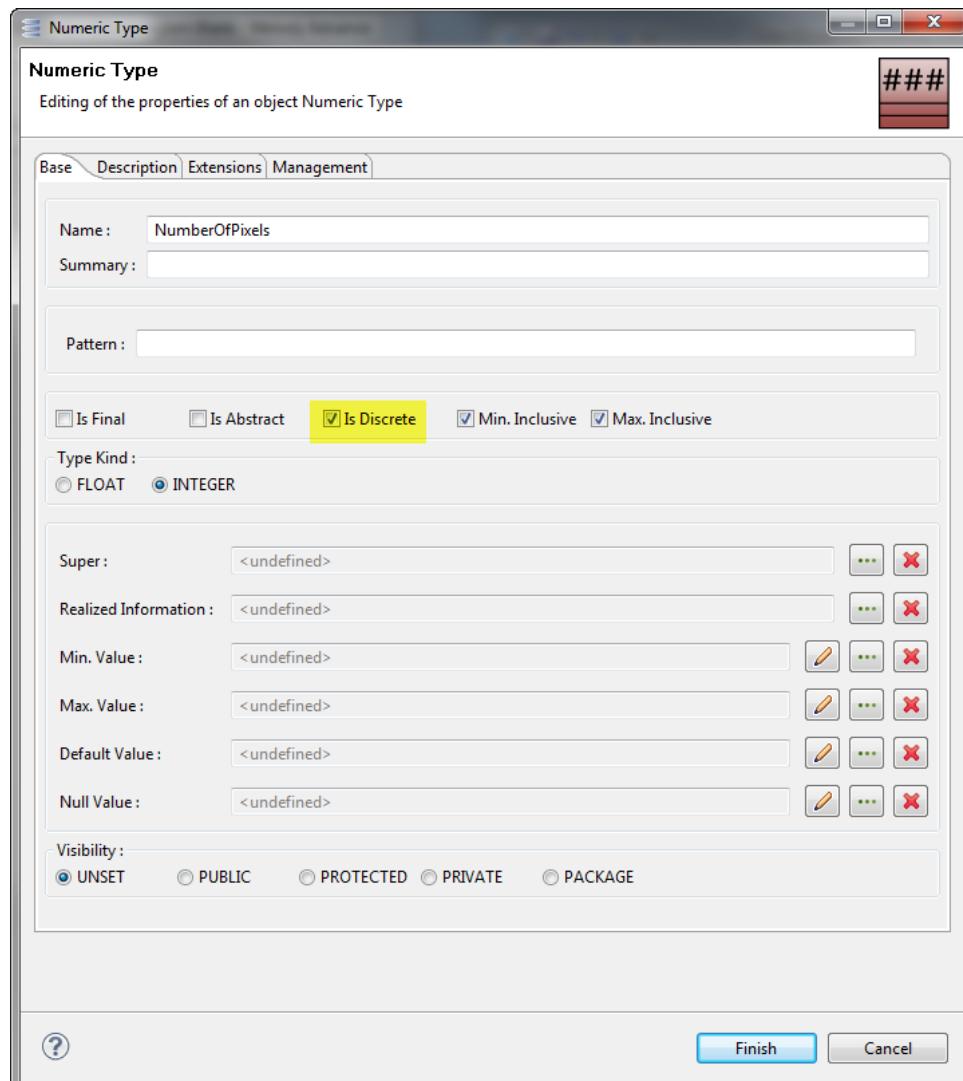
These constructs enable the modeler to go further, but are less important for beginners. They require more time and practice to be mastered.

### 5.1 Class and Type advanced features

Note that most of these advanced features are also available for Properties, Associations, Exchange Item Elements, etc.

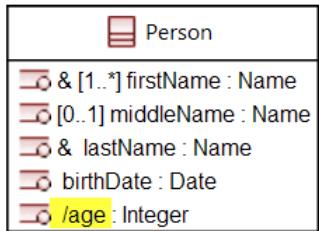
#### 5.1.1 isDiscrete

When the type is declared discrete, it means that it defines non continuous or separable values. Examples of common discrete types are Boolean and Enumeration. This quality is required for a type used as a discriminant in a [union](#) class.

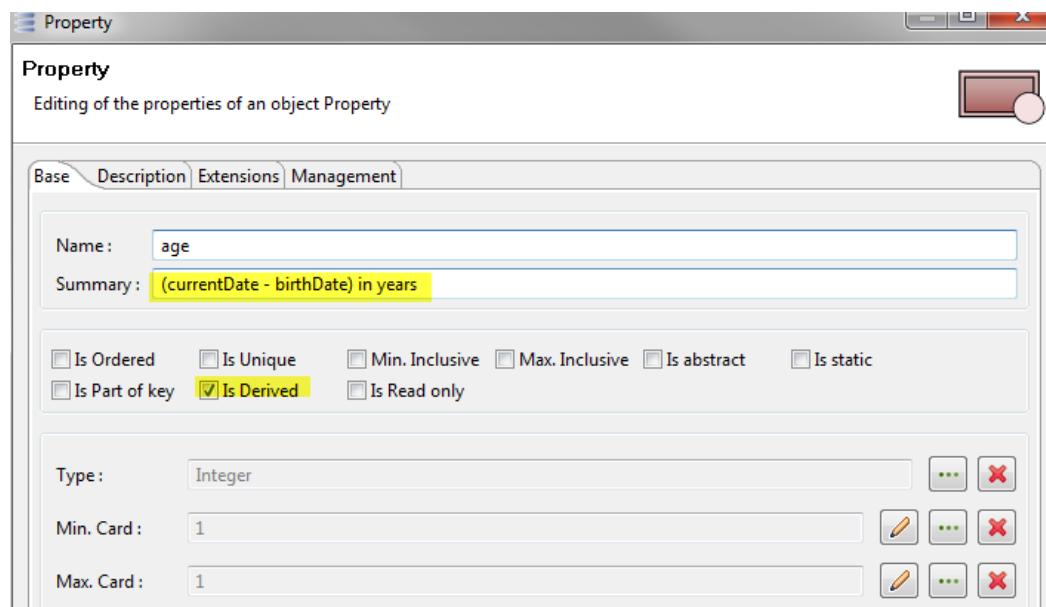


### 5.1.2 isDerived

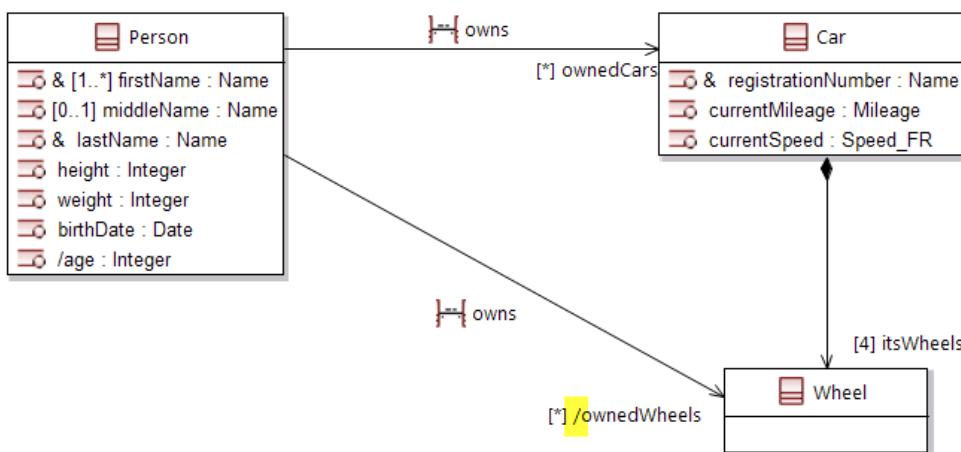
Specifies whether the property or association is derived from other model elements such as other properties, associations or constraints. The default value is false. For instance, a common derived property is the age of a Person, derived from the birthDate. The notation of a derived property is “/ derivedProperty”.

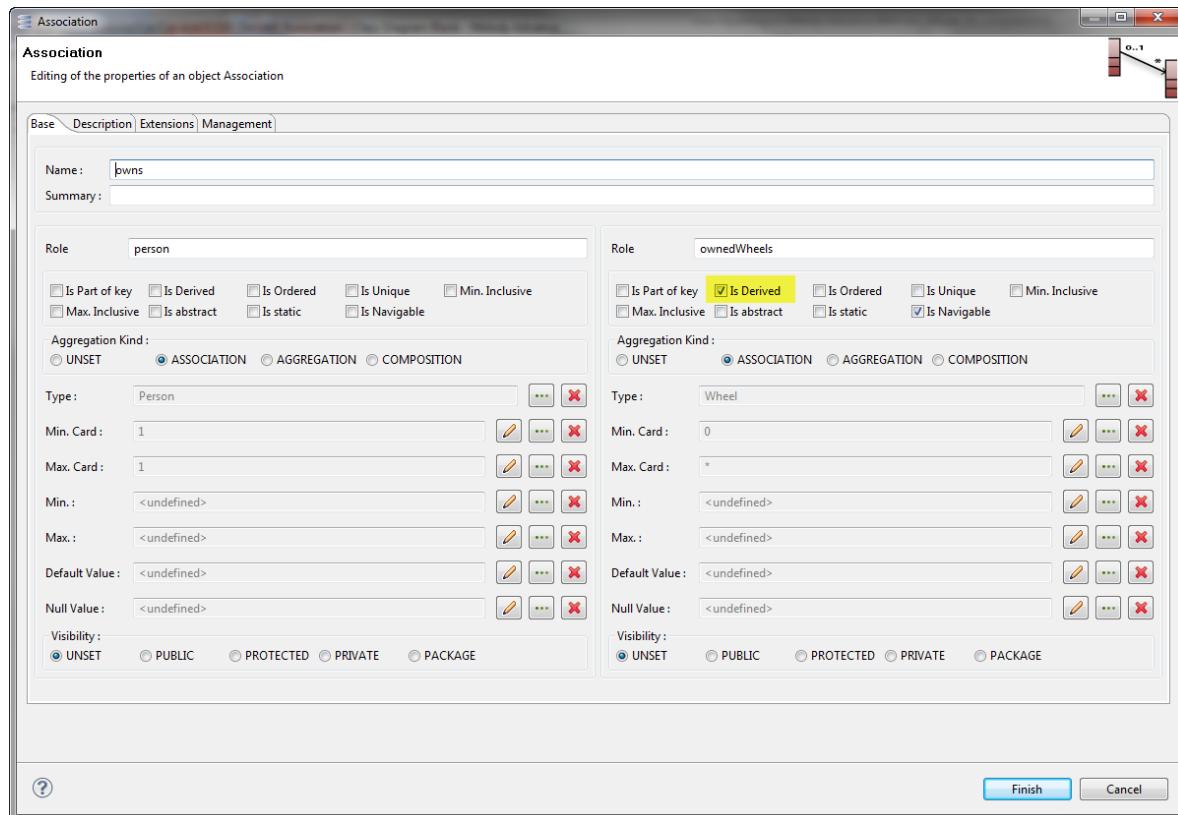


A good practice is to explain how the property is derived within the description field of the property or its summary.



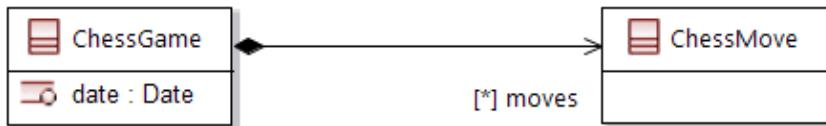
Association roles can also be derived as in the following example. A person may own several cars. As each car contains 4 wheels, the person also owns all the wheels!

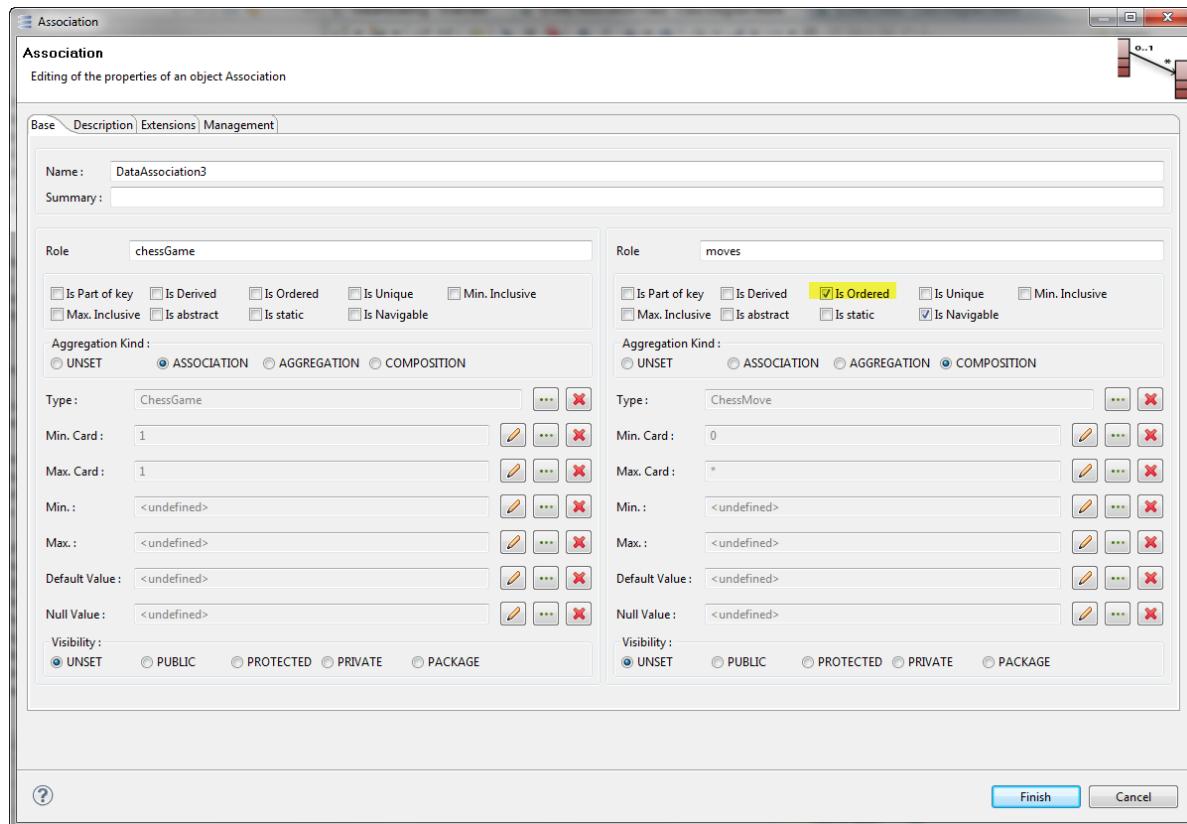




### 5.1.3 isOrdered

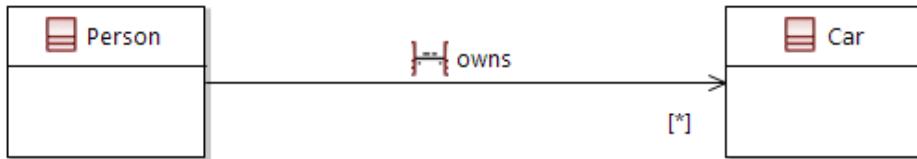
For multivalued elements (Max. Card > 1), this feature specifies whether the values in an instantiation of this element are sequentially ordered (Default is false).  
For instance, when you play chess, a game consists in a sequence of moves:





If the element is specified as ordered (i.e., `isOrdered` is true), then the set of values in an instantiation of this element is ordered. This ordering implies that there is a mapping from positive or null integers to the elements of the set of values. If an element is not multivalued, then the value for `isOrdered` has no semantic effect.

If the element is specified as unordered (i.e., `isOrdered` is false), then no assumptions can be made about the order of the values in an instantiation of this element.

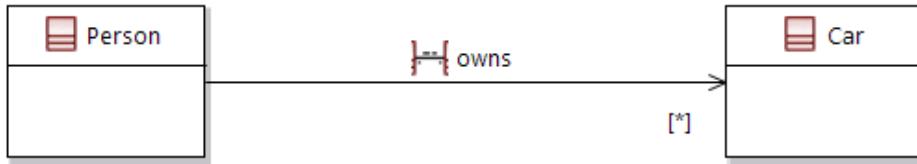


When one or more ends of the association are ordered, instances of association (links) carry ordering information in addition to their end values.

#### 5.1.4 `isUnique`

For multivalued elements ( $\text{Max. Card} > 1$ ), this feature specifies whether the values in an instantiation of this element are unique (Default is *false* in Melody Advance, although it should be *true*).

When `isUnique` is true, the collection of values may not contain duplicates. This is the most common case, as in the next figure (A person may own several cars, which are all different!).



When one or more ends of the association have `isUnique=false`, it is possible to have several links associating the same set of instances. In such a case, links carry an additional identifier apart from their end values. For instance if one wants to record a log of user connections, and count the number of times a specific user logged in during a specific duration, `isUnique` should be false. In this case, an additional attribute (such as connection time) may be needed.

Another simple example is shown on the next figure. In the game of chess, a game is a sequence of moves, involving one chess piece going from one square to another one. Several different moves may involve exactly the same piece and the same squares.



In combination with the feature [isOrdered](#), the following table (from UML 2.4 specification) indicates pertinent collection types:

**Table 7.1 - Collection types for properties**

isOrdered	isUnique	Collection type
<i>false</i>	<i>true</i>	<i>Set</i>
<i>true</i>	<i>true</i>	<i>OrderedSet</i>
<i>false</i>	<i>false</i>	<i>Bag</i>
<i>true</i>	<i>false</i>	<i>Sequence</i>

We can add that these two features are particularly important for code generation, but much less for pure system engineering modeling.

### 5.1.5 Is Final

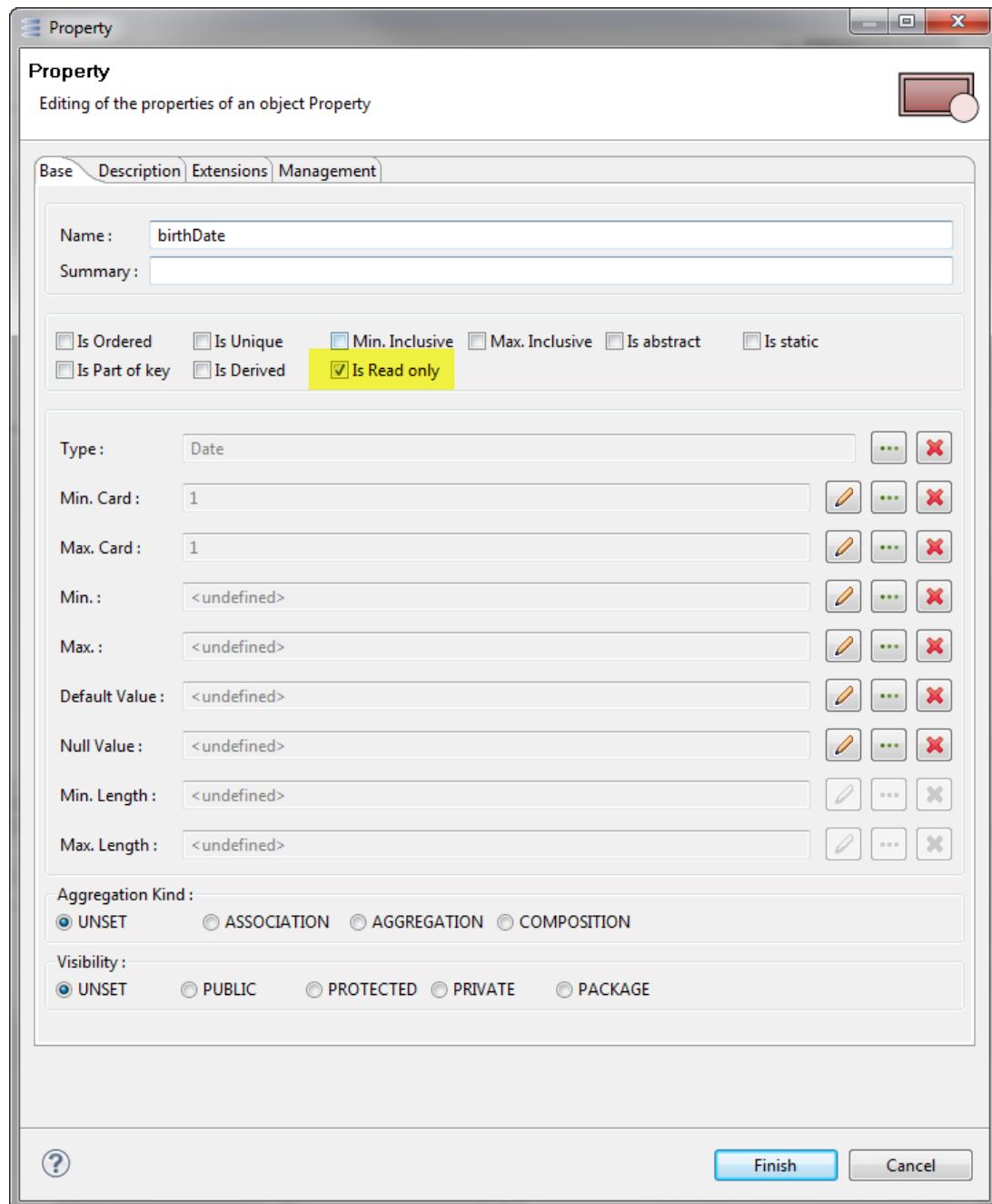
If a type is marked as `isFinal`, then it cannot be any longer specialized by any sub-type.

Take care: if a type is marked as `isFinal`, and you still create a sub-type, there is no error or even warning produced by the model-checking tool.

### 5.1.6 Is Read Only

If a property is marked as `readOnly`, then it cannot be updated once it has been assigned an initial value (Default is `false`).

For instance, the birthdate of a Person is normally readOnly, unlike the age or weight.



### 5.1.7 Pattern

Numeric, String and Enumeration types can be defined with a pattern which specifies the lexical space for their literal values. The syntax for expressing pattern is that of regular expressions in Java.

By default (when the pattern field is left empty in the type editor) a numeric type with kind:

- INTEGER has a built-in pattern which is  $(+|-)?([1-9][0-9]^*|0)$
- FLOAT has a built-in pattern which is  $(+|-)?([0-9]^*[.][0-9]+|[.][0-9]^*)(e|E)(+|-)[0-9]^+$

When running the validation tool, if any literal does not conform to the pattern defined for its type an error is raised.

### 5.1.8 Visibility

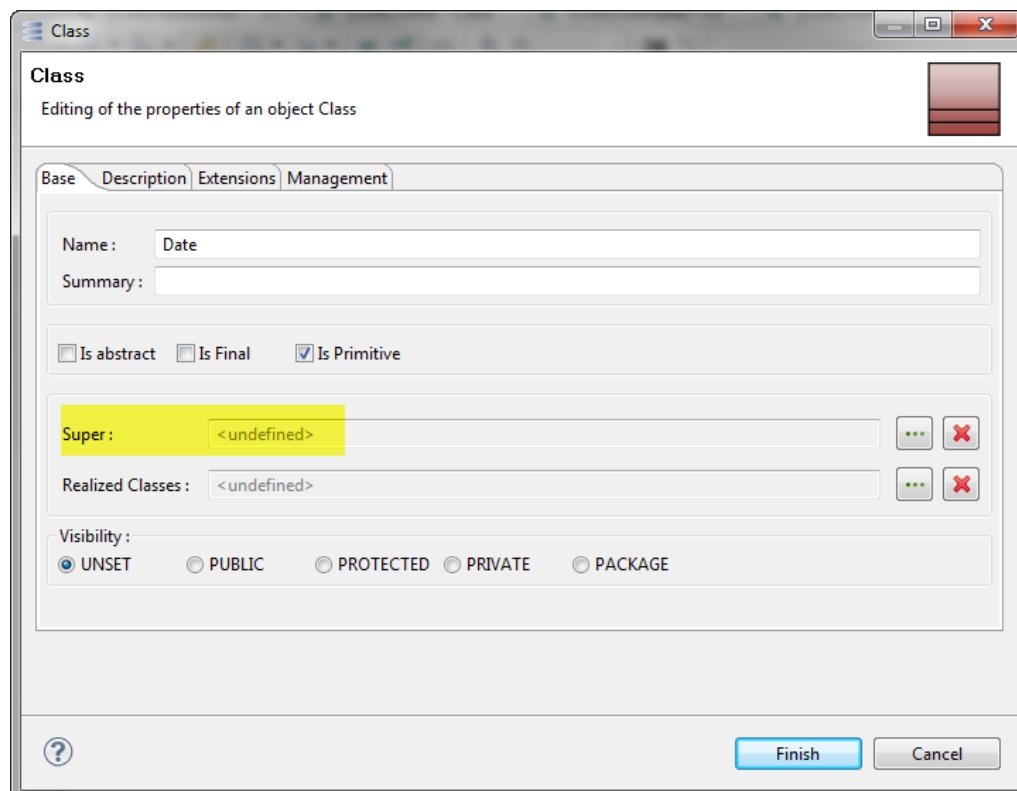
This feature comes from UML and object-oriented languages.

VisibilityKind is an enumeration type that defines literals to determine the visibility of elements in a model:

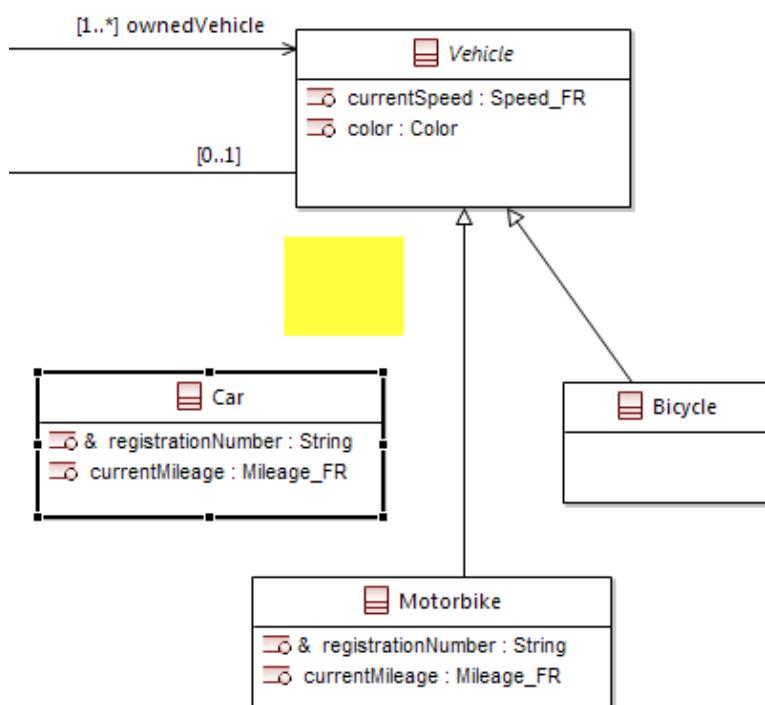
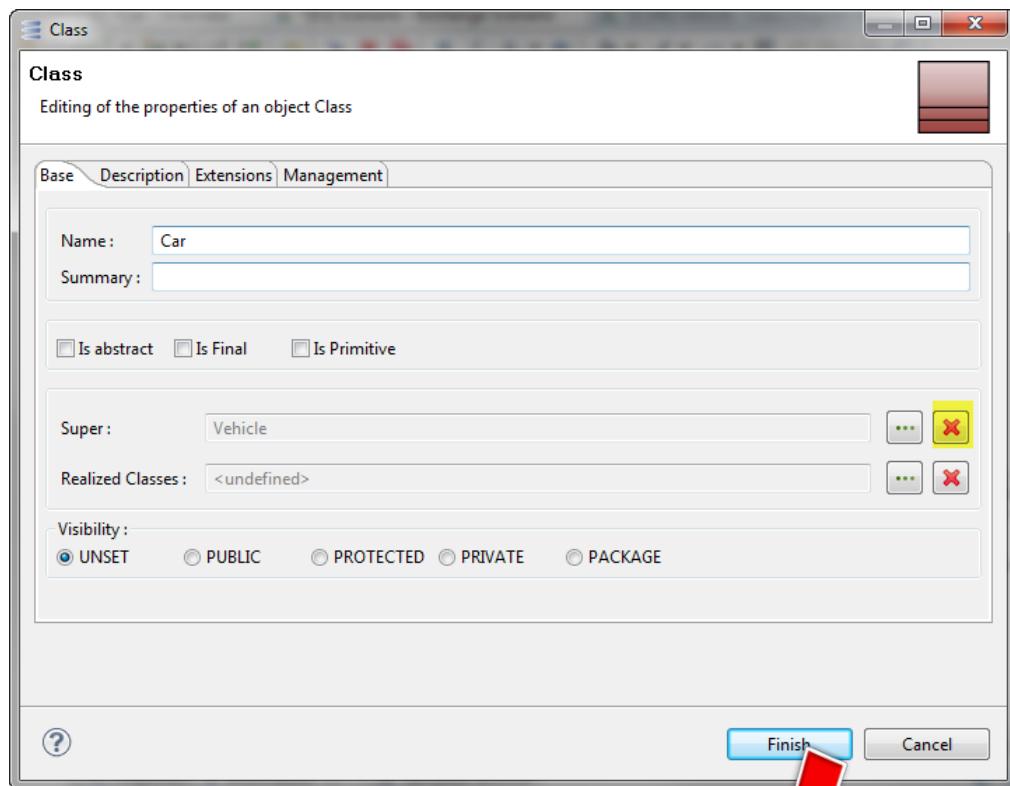
- A public element is visible to all elements that can access the contents of the namespace that owns it.
- A private element is only visible inside the namespace that owns it.
- A protected element is visible to elements that have a generalization relationship to the namespace that owns it.
- A package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace.

### 5.1.9 Super

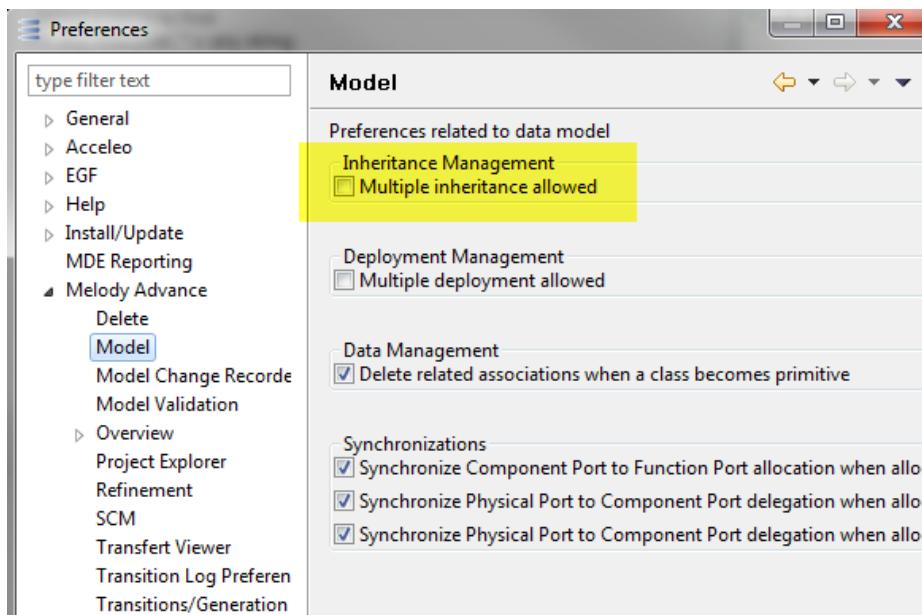
This field indicates the super-types (or generalized types) of the current type. It is empty by default.



As soon as a Generalization relationship is established, the “Super” field is automatically filled, as we saw in § [Generalization](#). Take care that this field can be edited. For instance, if you remove the *Vehicle* super-class from the *Car* Properties sheet, it deletes the generalization relationship and modifies the underlying model!



It is possible to allow multiple inheritance to define several super types.



Be careful: this action is not recommended. That is why it is not the default option, for it has different meanings depending on the targeted technology (e.g. with respect to late binding and overloading), it is not systematically supported, and it makes automatic processing of models a lot more complex.

## 5.2 Expression

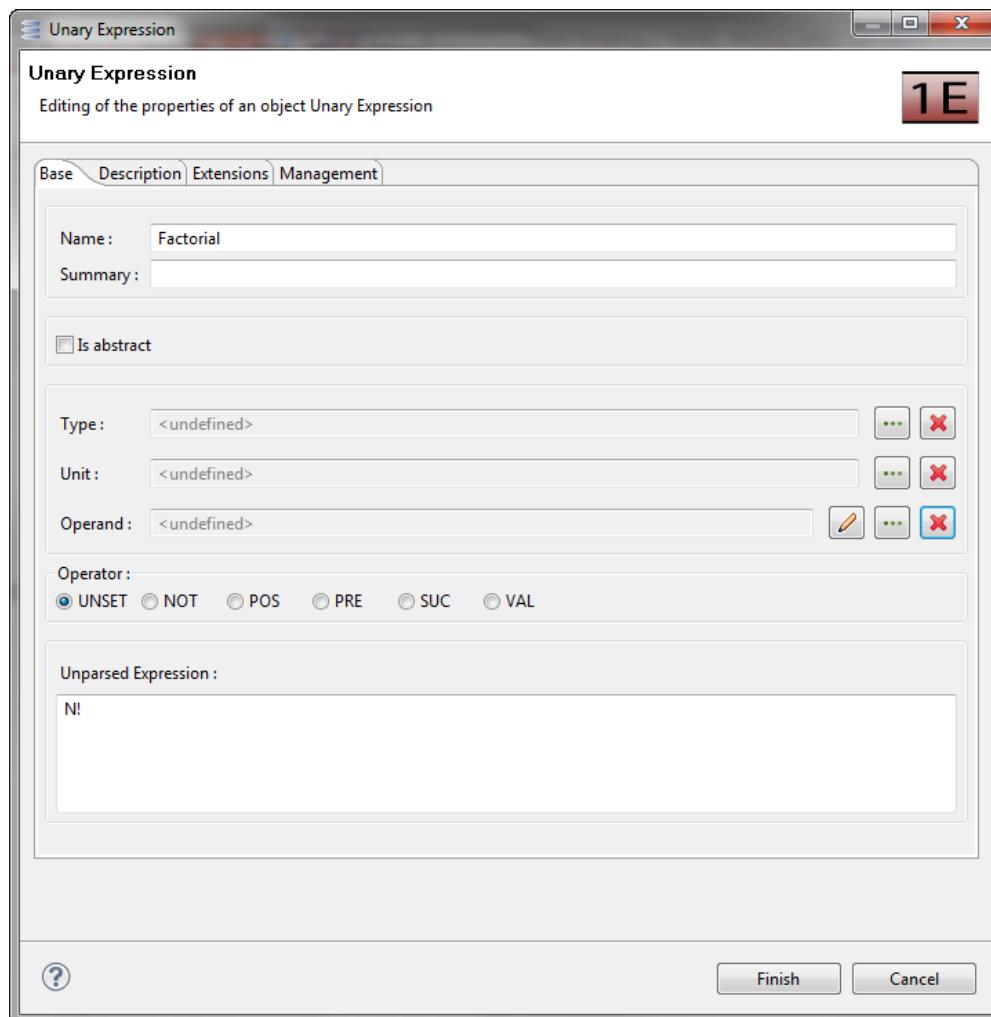
An Expression is a formula that yields values when evaluated in a context.

### 5.2.1 Unary Expression

A Unary Expression is one specific kind of Expression consisting of one Operator (+, -, not, or, etc.) and exactly one Operand. The Operand can itself be an Expression.

We can for instance create a Factorial Unary Expression to calculate the factorial of any interesting value.





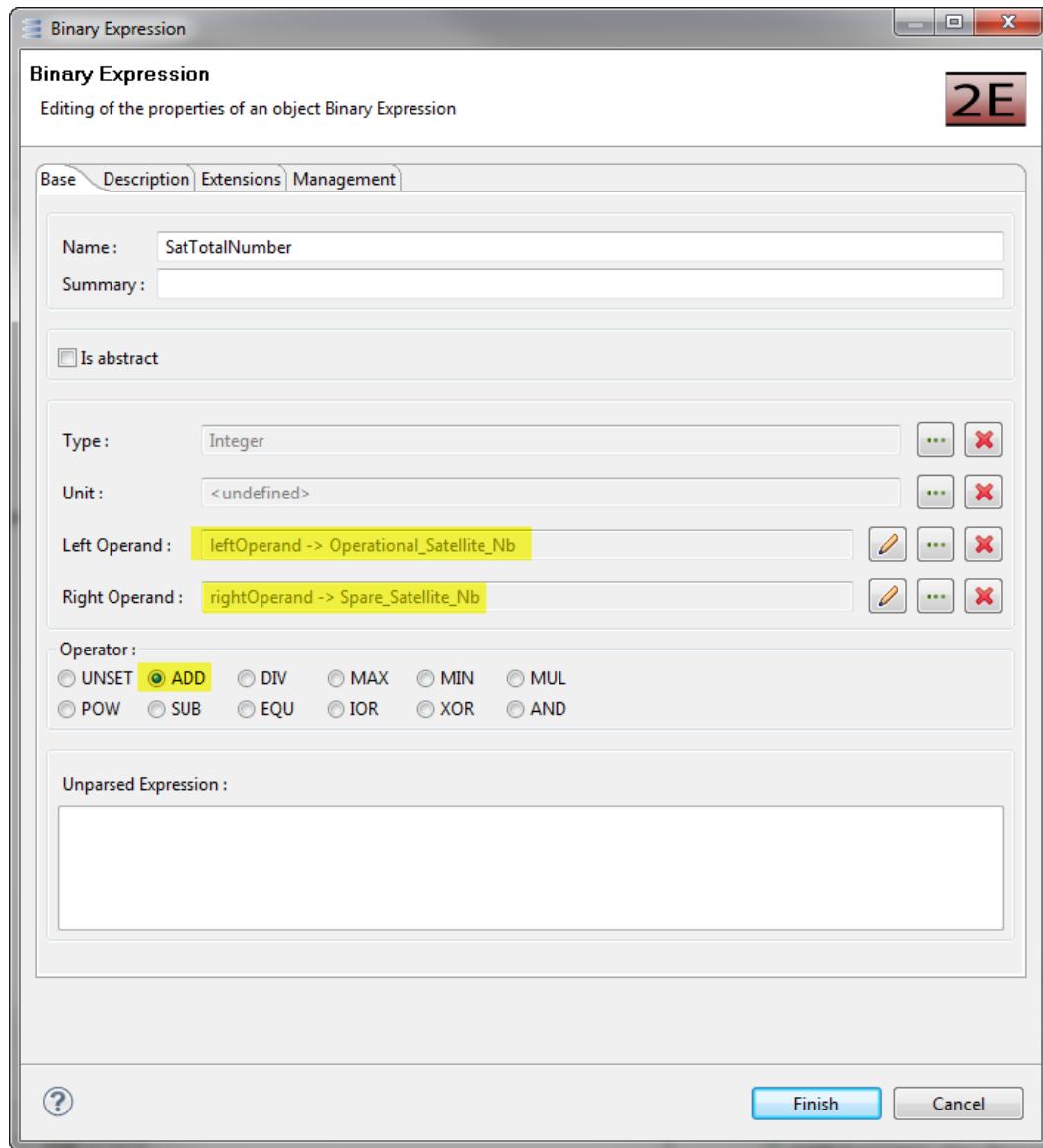
### 5.2.2 Binary expression

A Binary Expression is one specific kind of Expression consisting of one Operator (+, -, not, or, etc.) and exactly two Operands. Operands can themselves be Expressions.

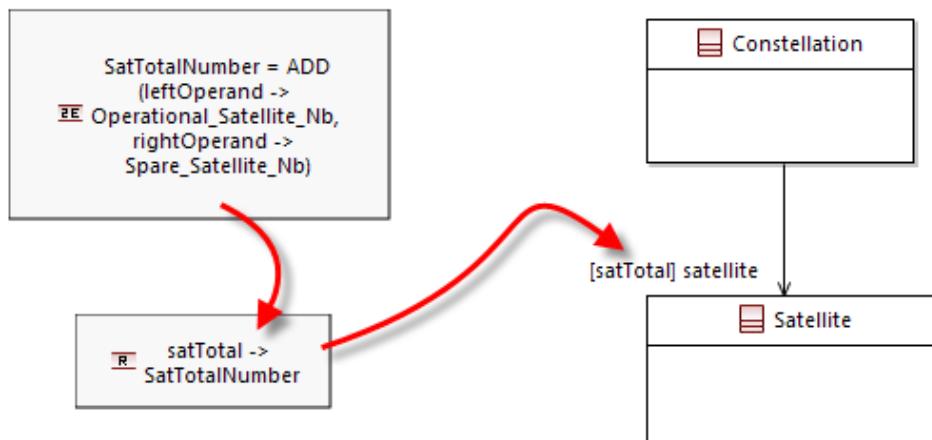
```

SatTotalNumber = ADD
  (leftOperand ->
  EE Operational_Satellite_Nb,
   rightOperand ->
   Spare_Satellite_Nb)
  
```

Let us give a simple example with an expression calculating the total number of satellites in the constellation :

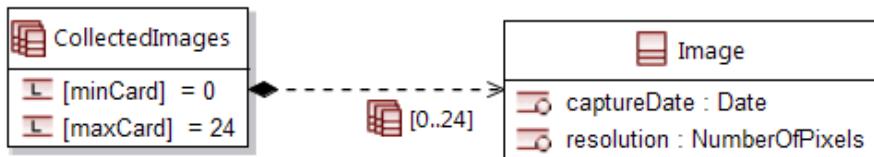


It is then possible to create a [Reference](#) from the Expression, and to use this reference for association cardinalities:

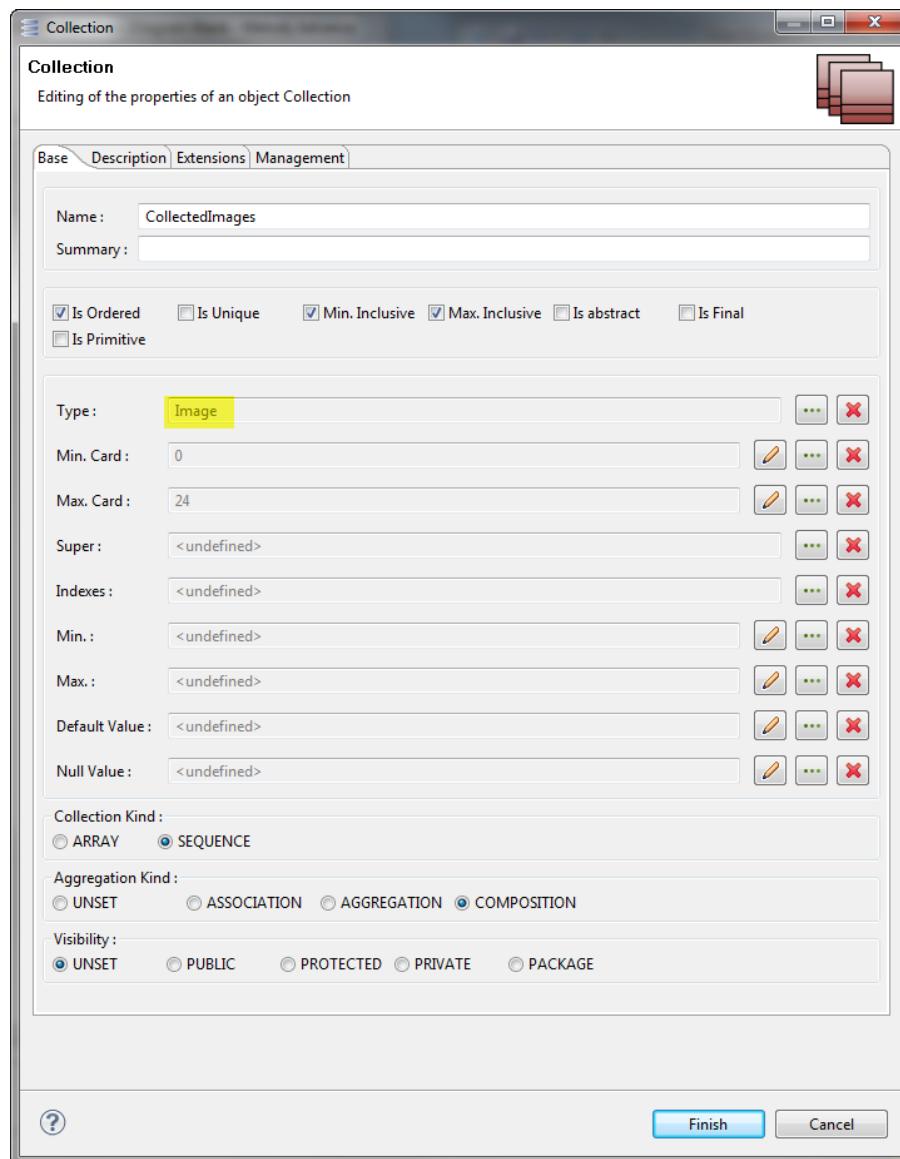


### 5.3 Collection

A Collection is a data modeling construct allowing the definition of sets of elements.

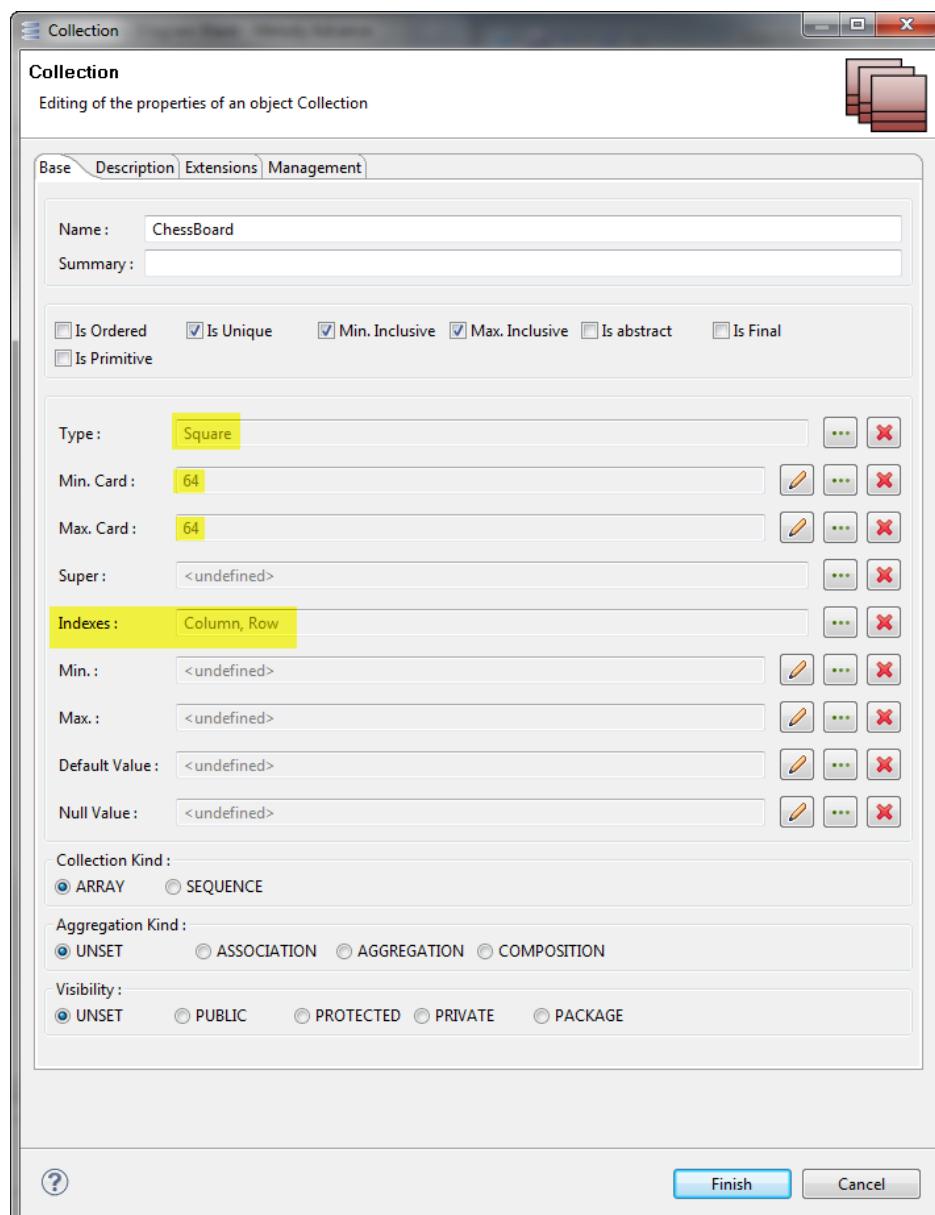
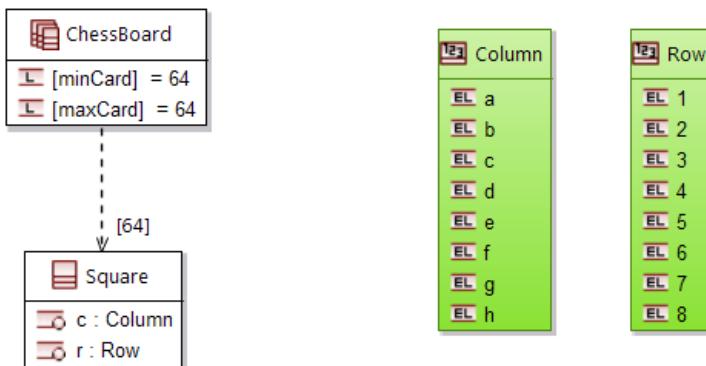


To properly define a Collection, you have to choose the type of its elements. It can be a class, a complex type or a simple type. Min and max cardinalities should be indicated and they appear in diagrams.



Collections can be indexed. The type of an index should be discrete. Very often, collections are not indexed. The implicit index is then an Integer. If the collection has one or several indexes, the cardinalities should be consistent with the product of the index cardinalities.

For example, a chess board consists in exactly 64 squares, indexed by the couple (column, row).



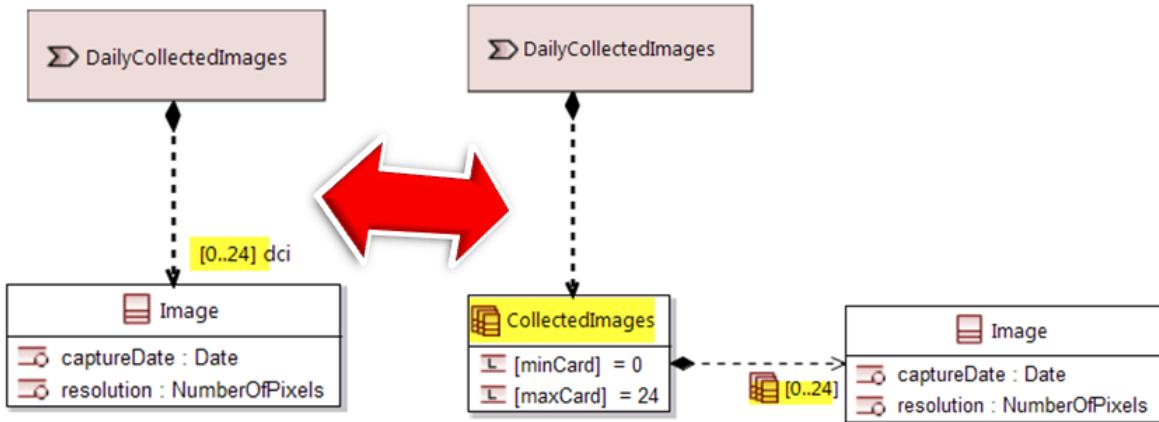
Collections can be Sequences or Arrays (note that Melody does not offer yet set or bags, for instance). This distinction is closely related to the Interface Definition Language (IDL) constructs from OMG:

- Sequences in IDL may be either unbounded (no maximum size) or bounded (a specific maximum size).

- IDL arrays are always of a fixed size.
- An IDL sequence is similar to a one-dimensional array of elements, but it can be unbounded.

Note that Collections are more design concepts than analysis concepts. To model domain concepts collections, it is often sufficient to use the cardinality adornment on an association, possibly with the “ordered” keyword.

Cardinalities of Exchange Item Elements can also be used to prevent creating unnecessary Collections in the Data Model.

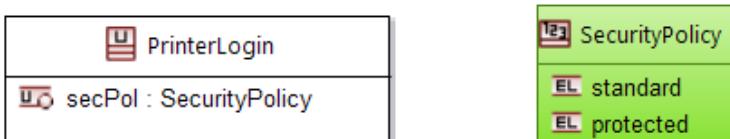


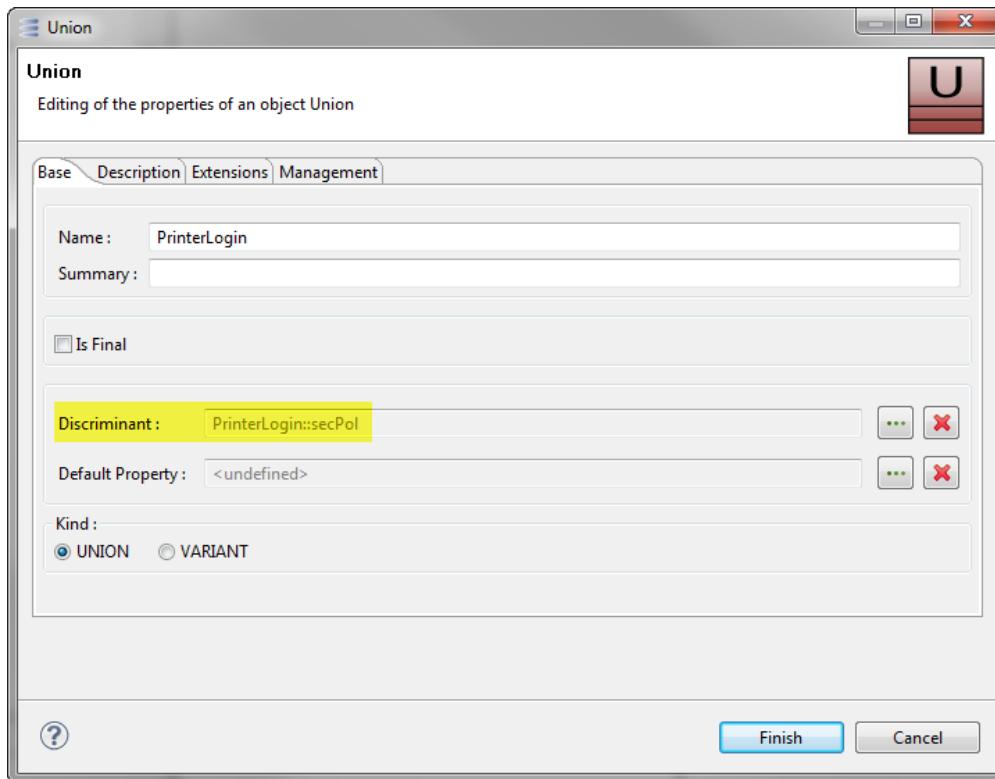
In the example above, the Exchange Item Element `dci` with a `[0..24]` cardinality prevents from creating an unnecessary Collection of Images (`CollectedImages`) in the Data Model.

## 5.4 Union / Variant

Unions and Variants are advanced structures that can take different forms according to the values of a Discriminant.

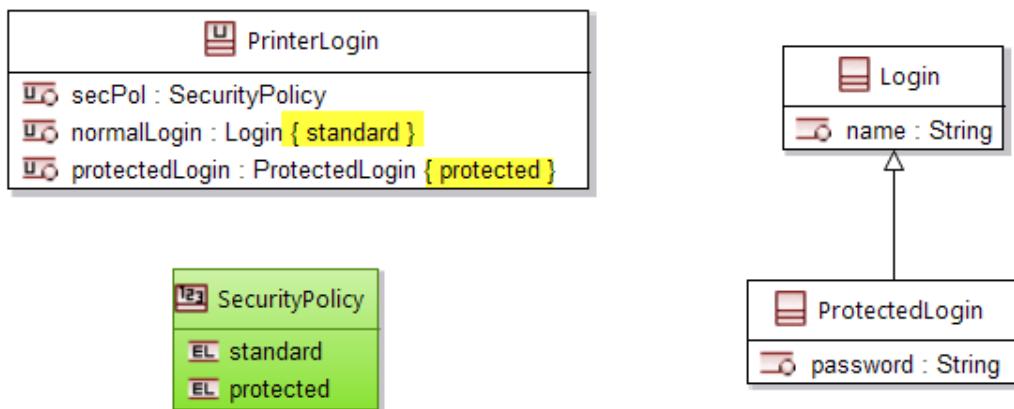
The first thing we have to define on the Union is which `UnionProperty` will be the discriminant. We have to create an [Enumeration](#), and type one `UnionProperty` by this Enumeration, and specify on the Union that we want to use it as a discriminant.





Let us give a simple example: according to a Security Policy, either normal or protected, employees will just need a name or an additional password to ask for a printing job.

We just have to specify two new *UnionProperties*, of types *Login* or *ProtectedLogin*, and we can for each of them define the pertinent Qualifier.



These constructs are mainly used in relation with software programming languages. For instance, in the Interface Definition Language (IDL) from OMG, a union is essentially a group of attributes, only one of which can be active at a time, according to the discriminant. A union saves memory space, because the amount of storage required for a union is the amount necessary to store its largest attribute. And Ada discriminated records are very similar to Melody variants.

Differences between Union and Variant are the following:

- In the Union, one value of the discriminant must be associated to at most one property and a property must always be associated to a value of the discriminant;
- In the Variant, one value of the discriminant can be associated to any number (0 or more) of properties. By convention a variant property that has no discriminant value

associated is deemed to be associated to all values of the discriminant: it is the fixed part.

Differences between OMG/IDL and Melody/Union are the following:

- Major: Melody/Union has no default clause;
- Minor: in Melody, the discriminant of a Union is a specific property of the Union.

Differences between ADA/discriminated record and Melody/Variant are the following:

- Major: Melody/ Variant has no default clause;
- Major: in Melody, a Variant only supports one discriminant.

## 5.5 Values and References

Values are instances of classes and data types and allow defining fixed data values. There are named values and anonymous values. Only named values can be reused to specify other model elements, such as [cardinalities](#), [default value](#), [null value](#), [min/max](#), [length](#).

### 5.5.1 Simple Values

Simple values were explained in the Basic Constructs chapter:

- See [Literal Numeric Value](#)
- See [Enumeration Literal](#)
- See [Literal Boolean Value](#)
- See [Literal String Value](#)
- ...

### 5.5.2 Complex Values

Complex Value is the valuation of a structured element (for example, [Class](#) or [Union](#)). It contains one Value Part per Property owned by the structured element typing the Complex Value.

A Value Part is thus a valuation of one specific Property of the structured element. The type of the Value Part is the same as the type of the Property.

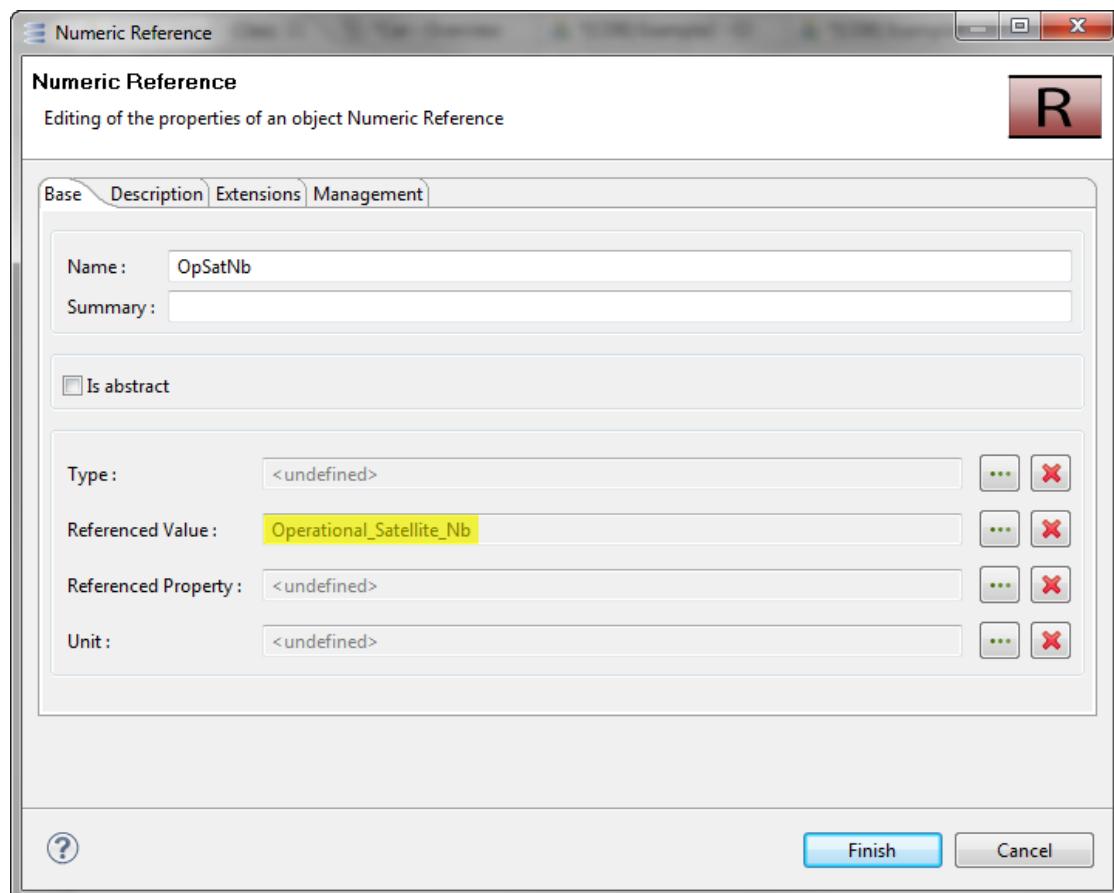
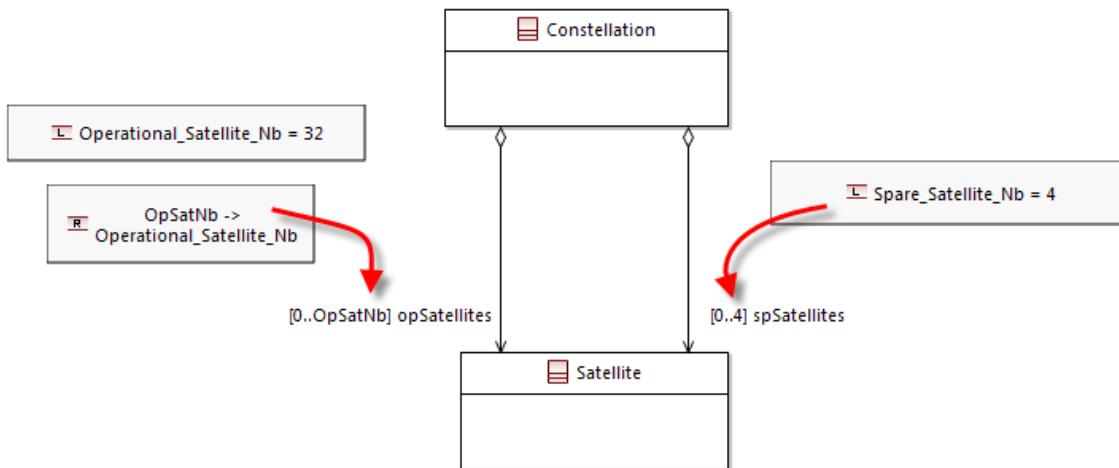
### 5.5.3 Collection Values

Collection Value is the valuation of a [Collection](#). It corresponds to a succession of Values for the type of the Collection.

### 5.5.4 References

Unlike Literals, References (Numeric, String, Expression, Enumeration, Boolean, Complex Value) allow defining Data Values related to other Data Values.

A Reference Data Value can target another Reference Data Value, A Literal Data Value or a Class Property.

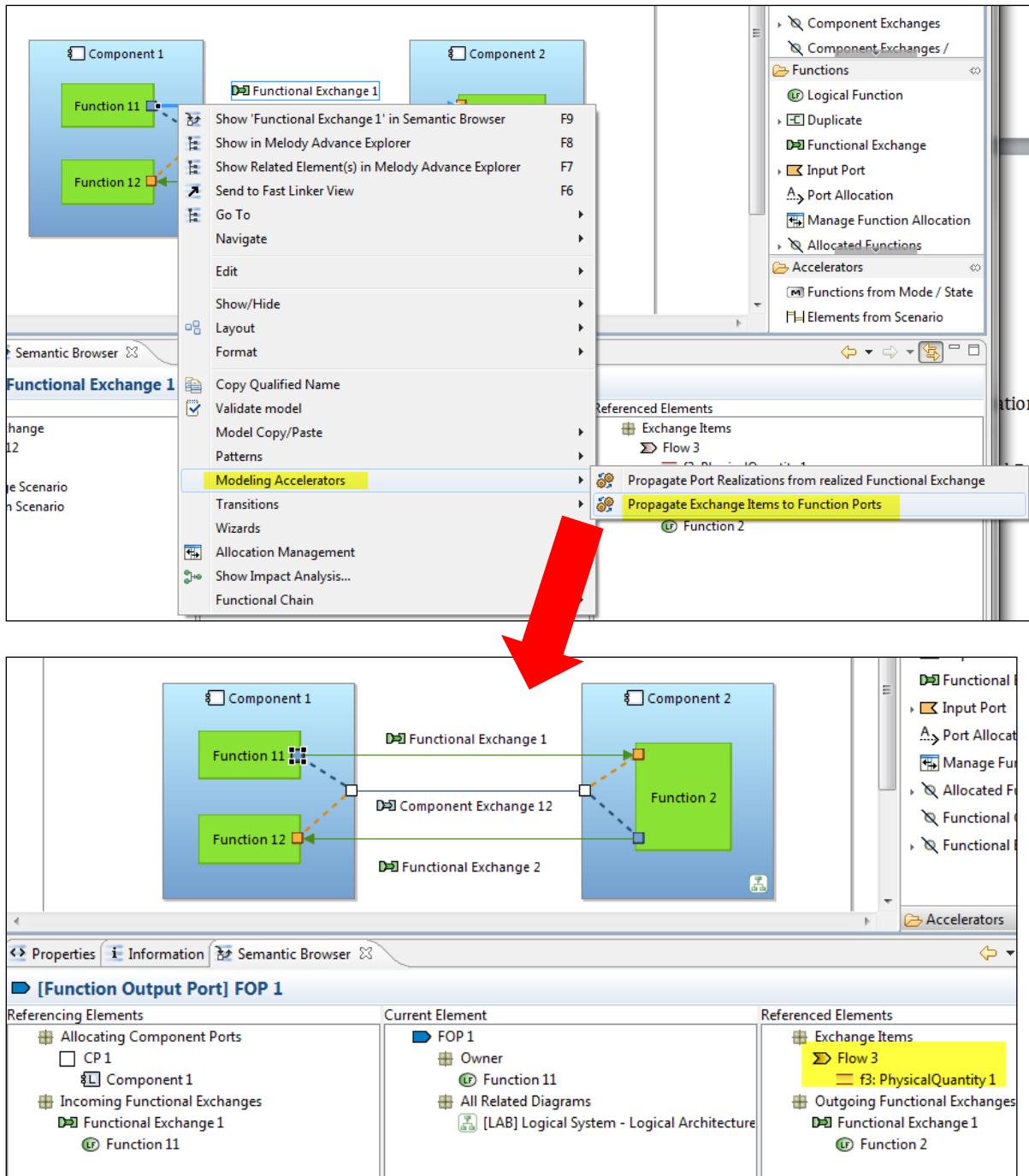


## 5.6 Modeling Accelerators

### 5.6.1 Exchange Items Propagation to Function Ports

Effect: If an Exchange Item is conveyed by a Functional Exchange linking two Function Ports, this accelerator will also associate it to each Function Port.

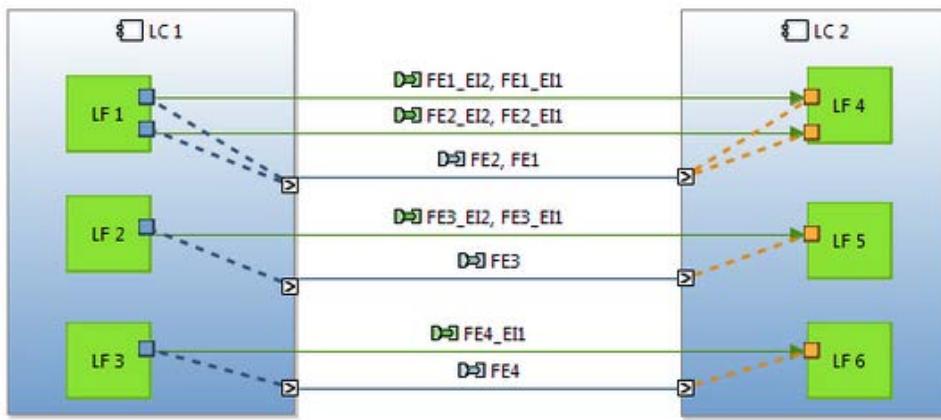
This is interesting mainly when you want to get an autonomous specification of inputs/outputs for each Function, via its owned ports.



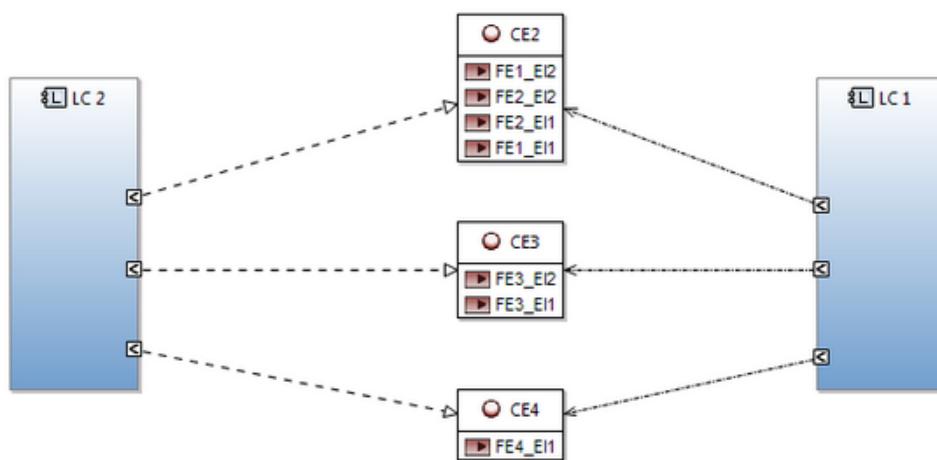
### 5.6.2 Interface Generation from Allocated Functions

Effect: Analyze the Component Exchanges, the Functional Exchanges they carry and the Exchange Items associated to the Functional Exchanges to generate a set of [Interfaces](#) between Components.

Example: Two components, three Component Exchanges, four Functional Exchanges associated to Exchange Items.



Result of the generation of Interfaces: One Interface is created per Component Exchange. All Exchange Items coming from different Functional Exchanges are aggregated in the same Interface.



## 6 METHODOLOGICAL RECOMMENDATIONS

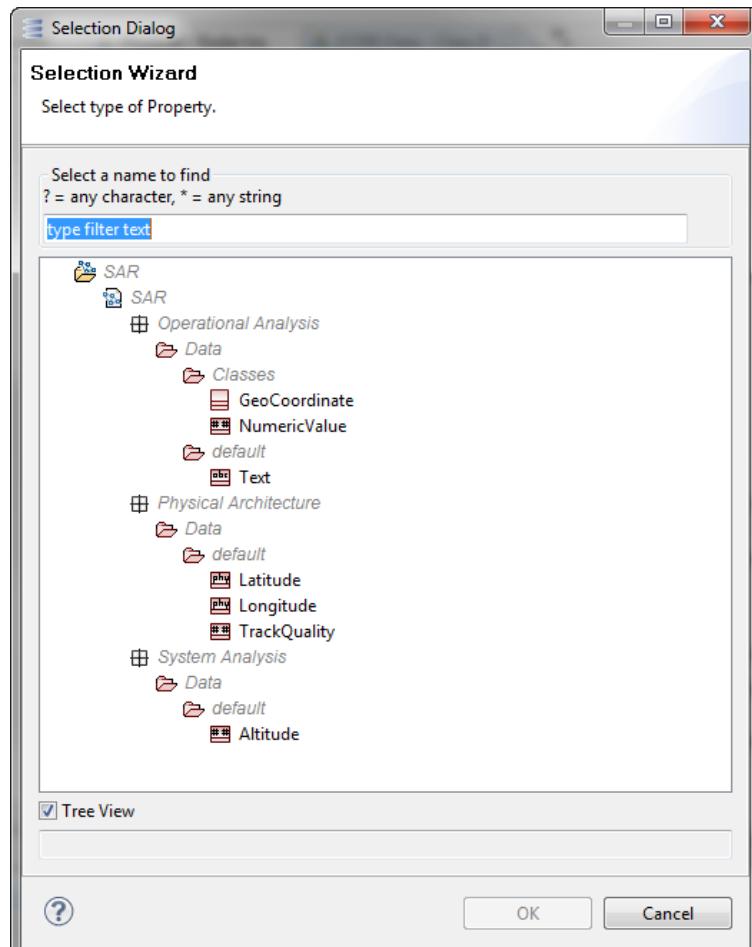
### 6.1 Data Model and Arcadia Abstraction Levels

There are several different ways to organize data modelling in Melody Advance.

In any case, you have to define precisely what your objectives for data modelling are. For instance do you want to describe domain concepts only, or do you want to use the data model for code generation purposes? Also, do you need a complete functional analysis with Component Exchanges and Functional Exchanges, or can you live with just Interfaces on Components without any functions?

A common practice, but not a mandatory one, is to describe all external data at System Analysis level (starting possibly at Operational Analysis), and to describe further internal data at Logical Architecture and/or Physical Architecture levels.

With Melody Advance, you are used to transition Actors, Functions, etc. from one Arcadia level to the next one down, and you cannot refer to a System function at Logical Architecture level, for instance. It is not the same with data: from a given level, you can refer to any Class, Basic Type, etc. defined in an upper level.



The main idea is to facilitate maintenance: if classes and types evolve at a given level, as we only reference them, there is nothing special to do. On the contrary, if we choose to transition manually the data, which is possible in Melody, we will need to update all lower levels as

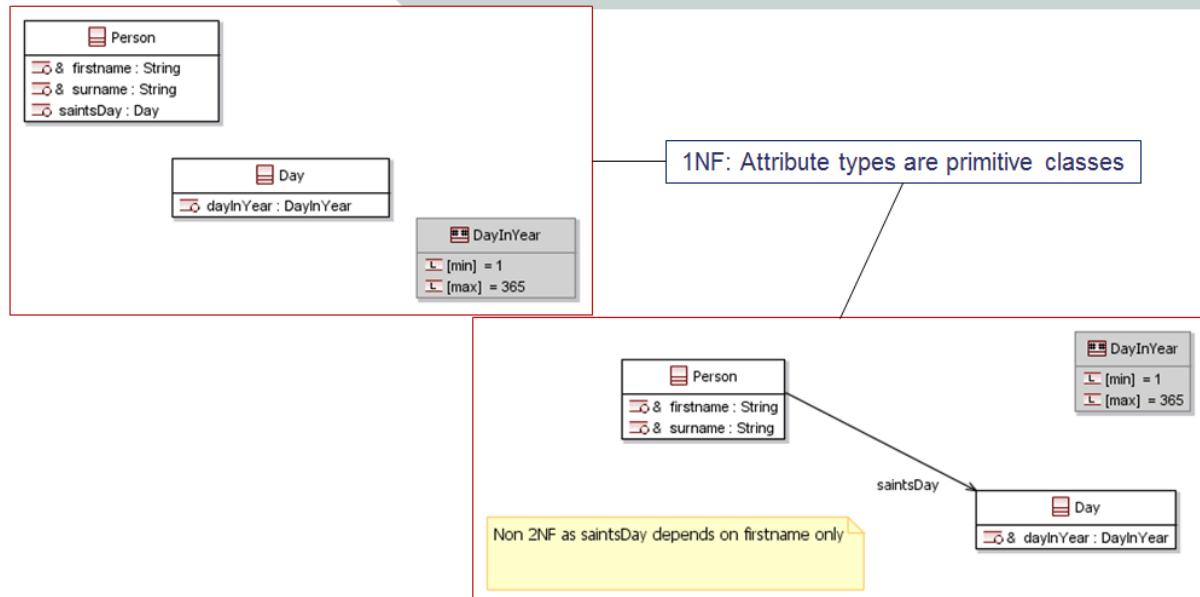
soon as there is a modification at a higher level. So the common recommendation is not to transition data, unless you really need to separate very strongly the different levels of a specific model.

Trouver des exemples programmes : EGNOS, etc. (action Thales)

## 6.2 Data Normalization

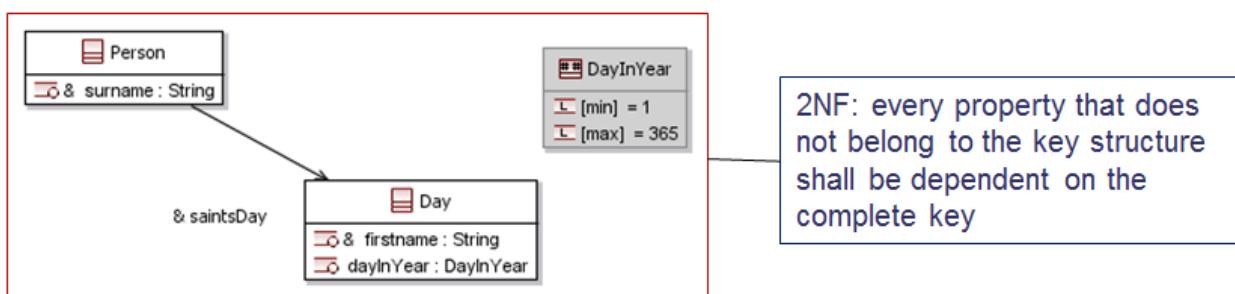
### 6.2.1 1NF

Attribute types are primitive classes. Value of each attribute is atomic, there is no embedded class.



### 6.2.2 2NF

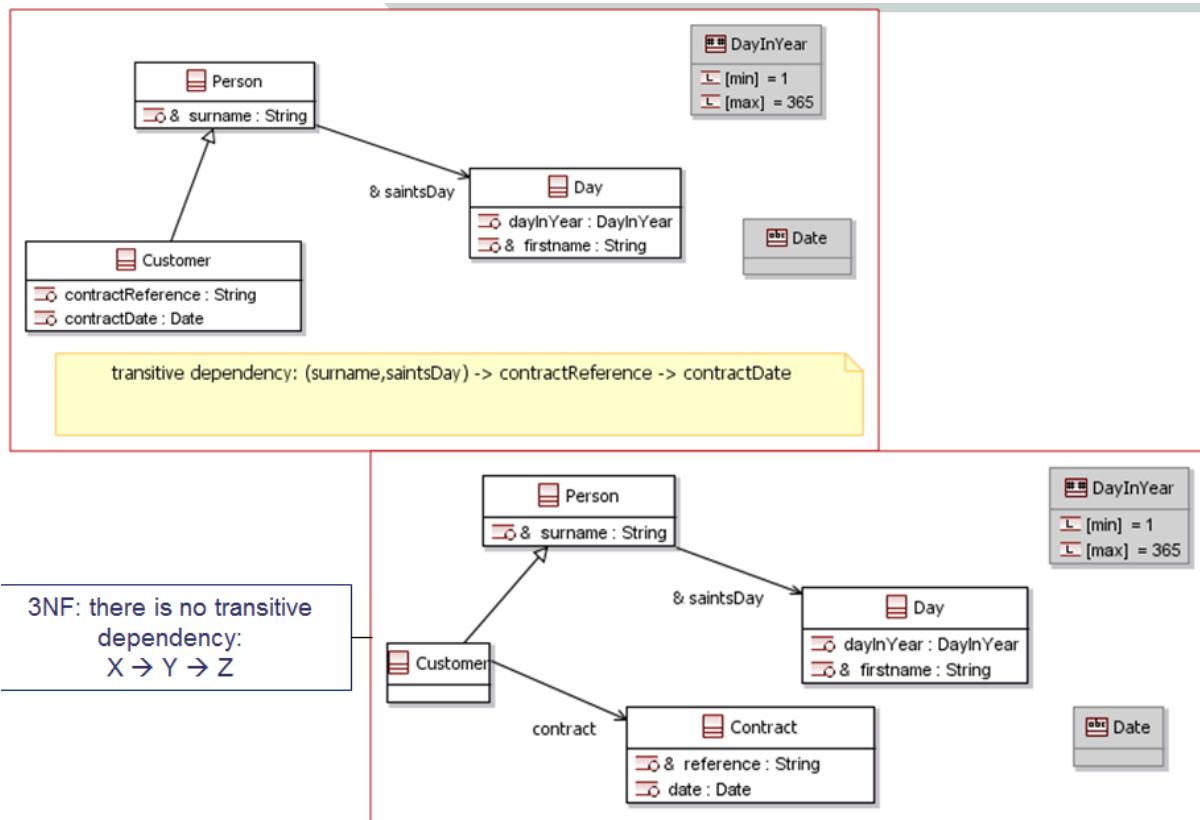
The model shall be 1NF and every property (attribute or association) that does not belong to the key structure shall be dependent on the complete key: the key is the smallest key for the properties of this class.



### 6.2.3 3NF

The model shall be 2NF and there is no transitive dependence: X -> Y -> Z

A property (attribute or association role) 'Y' is functionally dependent on a group of properties 'X' = (X<sub>1</sub>, ..., X<sub>n</sub>) if and only if its value is by nature determined as soon as the value 'V' = (V<sub>1</sub>, ..., V<sub>n</sub>) from group 'X' is known. We note X -> Y.



Contrary to preconceived ideas, in most cases, only one 3NF model exists.  
More precisely, the greater the degree of normalization is required, the more the possible interpretations of the model are reduced.

### 6.3 Validating the interface and data model

A large number of rules checked by Melody apply to the Interface and Data Model. You should rely on them to maintain your model and ensure its consistency and completeness.

For instance, in the *Design – Well-Formedness* category, there is a big *Data* group:

**Constraints**

Constraint categories:	Select constraints to enable:
<ul style="list-style-type: none"> <li>▪ Melody Advance</li> <li>▪ Design           <ul style="list-style-type: none"> <li>▪ Completeness</li> <li><input checked="" type="checkbox"/> Consistency</li> <li>▪ Coverage</li> <li>▪ Well-Formedness               <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> Capabilities</li> <li><input checked="" type="checkbox"/> Components</li> <li><b>Data</b> <span style="background-color: yellow; border: 1px solid black; padding: 2px;"> </span> <ul style="list-style-type: none"> <li>▪ Dataflows</li> <li><input checked="" type="checkbox"/> Interfaces</li> <li><input checked="" type="checkbox"/> Scenarios</li> <li><input checked="" type="checkbox"/> State Machines</li> <li>▪ Use Cases</li> </ul> </li> </ul> </li> <li>▪ Integrity</li> <li>▪ Quality</li> <li>&gt; Transition           <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> pattern</li> <li><input checked="" type="checkbox"/> patternParameter</li> <li><input checked="" type="checkbox"/> patternVariable</li> <li><input checked="" type="checkbox"/> Referential Integrity</li> <li><input checked="" type="checkbox"/> Viewpoint Constraints</li> <li><input checked="" type="checkbox"/> Viewpoint Validation Rules</li> </ul> </li> </ul> </li></ul>	<ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> DWF_D_01 - Union qualifier integrity</li> <li><input checked="" type="checkbox"/> DWF_D_02 - Union qualifier cardinality</li> <li><input checked="" type="checkbox"/> DWF_D_03 - Discriminant Union Property Qualifier</li> <li><input checked="" type="checkbox"/> DWF_D_04 - Composition Cardinality Rule</li> <li><input checked="" type="checkbox"/> DWF_D_05 - Association Ends IsPrimitive check</li> <li><input checked="" type="checkbox"/> DWF_D_06 - Property isPrimitive type check</li> <li><input checked="" type="checkbox"/> DWF_D_08 - Union Discriminant</li> <li><input checked="" type="checkbox"/> DWF_D_09 - Association navigable members check</li> <li><input checked="" type="checkbox"/> DWF_D_10 - Aggregation/Composition navigability check</li> <li><input checked="" type="checkbox"/> DWF_D_11 - Association member AggregationKind check</li> <li><input checked="" type="checkbox"/> DWF_D_12 - Property type check</li> <li><input checked="" type="checkbox"/> DWF_D_13 Consistency of Qualifier values</li> <li><input checked="" type="checkbox"/> DWF_D_14 - Association members check</li> <li><input checked="" type="checkbox"/> DWF_D_15 - Class self generalization</li> <li><input checked="" type="checkbox"/> DWF_D_16 - Class cycles check (inner package)</li> <li><input checked="" type="checkbox"/> DWF_D_17 - Union Discriminant Data Value</li> <li><input checked="" type="checkbox"/> DWF_D_18 - Collection type check</li> <li><input checked="" type="checkbox"/> DWF_D_19 - Class Multiple Inheritance</li> <li><input checked="" type="checkbox"/> DWF_D_20 - Constraint expression type check 1</li> <li><input checked="" type="checkbox"/> DWF_D_21 - DataType Pattern Validation</li> <li><input checked="" type="checkbox"/> DWF_D_22 - Boolean Type Incorrect literals values</li> <li><input checked="" type="checkbox"/> DWF_D_23 - NumericValue Unit Consistency</li> <li><input type="checkbox"/> DWF_D_24 - DataValue Pattern Validation</li> <li><input checked="" type="checkbox"/> DWF_D_25 - Dependency Cycle Detection</li> <li><input checked="" type="checkbox"/> DWF_D_26 LiteralBooleanValue Type</li> <li><input checked="" type="checkbox"/> DWF_D_27 - EnumerationLiteral Type</li> <li><input checked="" type="checkbox"/> DWF_D_28 - DataValue Reference check</li> <li><input checked="" type="checkbox"/> DWF_D_29 - Association members AggregationKind check 2</li> <li><input type="checkbox"/> DWF_D_30 - Boolean Value Family Type check</li> <li><input checked="" type="checkbox"/> DWF_D_31 - Description is well formed XML</li> <li><input type="checkbox"/> DWF_D_32 - Collection Value Family Type check</li> <li><input type="checkbox"/> DWF_D_33 - String Value Family Type check</li> <li><input type="checkbox"/> DWF_D_34 - Complex Value Family Type check</li> <li><input type="checkbox"/> DWF_D_35 - Enumeration Value Family Type check</li> <li><input type="checkbox"/> DWF_D_36 - Numeric Value Family Type check</li> <li><input type="checkbox"/> DWF_D_37 - Expression Value Family Type check</li> <li><input type="checkbox"/> DWF_D_38 - Association Target Class Property key value check</li> <li><input checked="" type="checkbox"/> DWF_D_39 - Data Value Name Check</li> <li><input checked="" type="checkbox"/> DWF_D_40 - Association Consistency between layers</li> <li><input type="checkbox"/> DWF_D_41 - class with an abstract attribute/role</li> <li><input type="checkbox"/> DWF_D_42 - Packages visibility problem</li> </ul>

There is also an *Interfaces* group:

**Constraints**

Constraint categories:	Select constraints to enable:
<ul style="list-style-type: none"> <li>▪ Melody Advance           <ul style="list-style-type: none"> <li>▪ Design               <ul style="list-style-type: none"> <li>□ Completeness</li> <li><input checked="" type="checkbox"/> Consistency</li> <li>□ Coverage</li> </ul> </li> <li>▪ Well-Formedness               <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> Capabilities</li> <li><input checked="" type="checkbox"/> Components</li> <li>□ Data</li> <li>□ Dataflows</li> <li><input checked="" type="checkbox"/> Interfaces</li> <li><input checked="" type="checkbox"/> Scenarios</li> <li><input checked="" type="checkbox"/> State Machines</li> <li>□ Use Cases</li> </ul> </li> </ul> </li> <li>▪ Integrity</li> <li>▪ Quality</li> <li>&gt; Transition           <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> pattern</li> <li><input checked="" type="checkbox"/> patternParameter</li> <li><input checked="" type="checkbox"/> patternVariable</li> <li><input checked="" type="checkbox"/> Referential Integrity</li> <li><input checked="" type="checkbox"/> Viewpoint Constraints</li> <li><input checked="" type="checkbox"/> Viewpoint Validation Rules</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> DWF_I_01 - ExchangeItemParameter kind consistency</li> <li><input checked="" type="checkbox"/> DWF_I_03 - ExchangeItemElement Direction check</li> <li><input checked="" type="checkbox"/> DWF_I_04 - Components must implements/uses accessible interfaces</li> <li><input checked="" type="checkbox"/> DWF_I_05 - Component Communication Link kind</li> <li><input checked="" type="checkbox"/> DWF_I_06 - Interface location check</li> <li><input checked="" type="checkbox"/> DWF_I_07 - Interface location level</li> <li><input checked="" type="checkbox"/> DWF_I_08 - Component port required and provided interface check</li> <li><input checked="" type="checkbox"/> DWF_I_09 - CommunicationLink kind and ExchangeItem kind</li> <li><input checked="" type="checkbox"/> DWF_I_10 - CommunicationLink protocol and ExchangeItem kind</li> <li><input checked="" type="checkbox"/> DWF_I_11 - ExchangeItemAllocation AllocatedItem Check</li> <li><input checked="" type="checkbox"/> DWF_I_12 - Exchange item allocation layer consistency</li> </ul>

Some rules are nevertheless still missing and get progressively integrated on user suggestion.

## 6.4 Basic Best Practices

The model content and diagrams should be homogeneous between model contributors. Hence define the rules to be applied by each contributor:

- To make modelling efficient (less rework)
- To make good diagrams (diagram layout conventions)

These practices should therefore be shared between all the team members from the start of a project, and documented in order to allow consistent model maintenance over the lifecycle.

### 6.4.1 Creating new model elements

Before creating a new Type, check that it does not already exist!

Take care: model checking can detect duplicates only if they have the same name. And remember also that two elements created in different packages are considered different, even if they have the same name.

### 6.4.2 Naming Conventions

Capitalize the first letter of class names.

Begin property names with a lowercase letter. Multi-word names are often formed by concatenating the words and using lowercase for all letters except for upcasing the first letter of each word but the first.

A possible rule for Primitive Classes and Simple Types is to prefix their name with "T", in order to easily distinguish them from a standard class.

### 6.4.3 Package Structuration

A package should never contain more than 20 elements. If it the case, break it down into sub-packages.

In order to avoid complex package structuration define guidelines from the the project start on how data should be categorized into packages (e.g. packages can be structured according to data main usage: command & control, maintenance...)

### 6.4.4 Diagram Layout

In the CDB, a good practice is to use only rectilinear line styles.

Put generalized classes above specialized ones.

Do not clutter your diagrams: not more than 12 classes or types in the same CDB.

### 6.4.5 Using color codes on diagrams

This guideline applies to any diagram and any type of model.

In case color codes are really needed to identifish specific concerns visually on a diagram, this should be managed in a formal way :

- Identifying formally a property or property value which holds the semantic concept under consideration,
- Developping a viewpoint that will ensure unicity and consistency of this attribute across all model elements,
- Using this same viewpoint to manage the color that is displayed on a diagram.

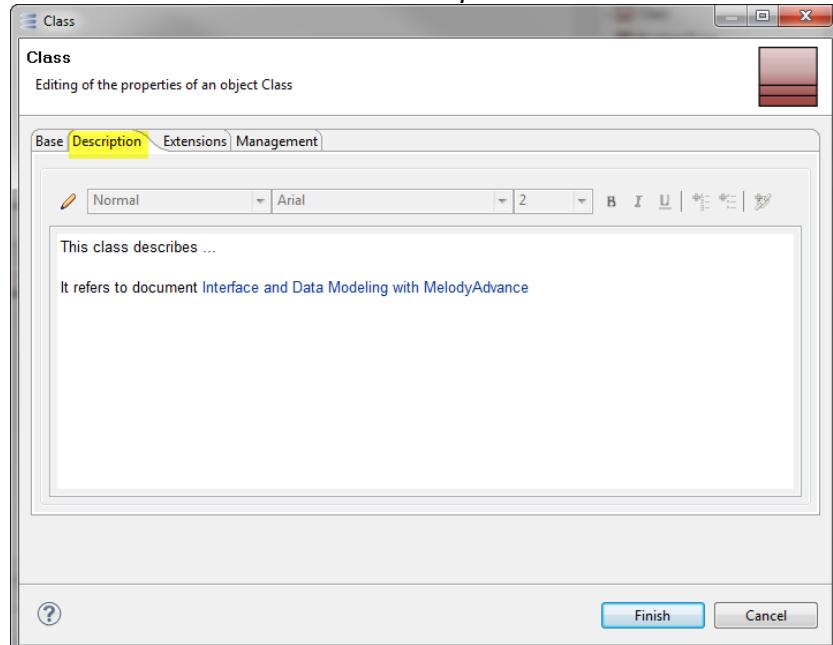
Using viewpoints will prevent from manual maintenance of the color codes which would be likely to generate inconsistencies and errors with critical impacts.

### 6.4.6 Document and annotate the model

Do not forget to document extensively your modeling elements and diagrams.

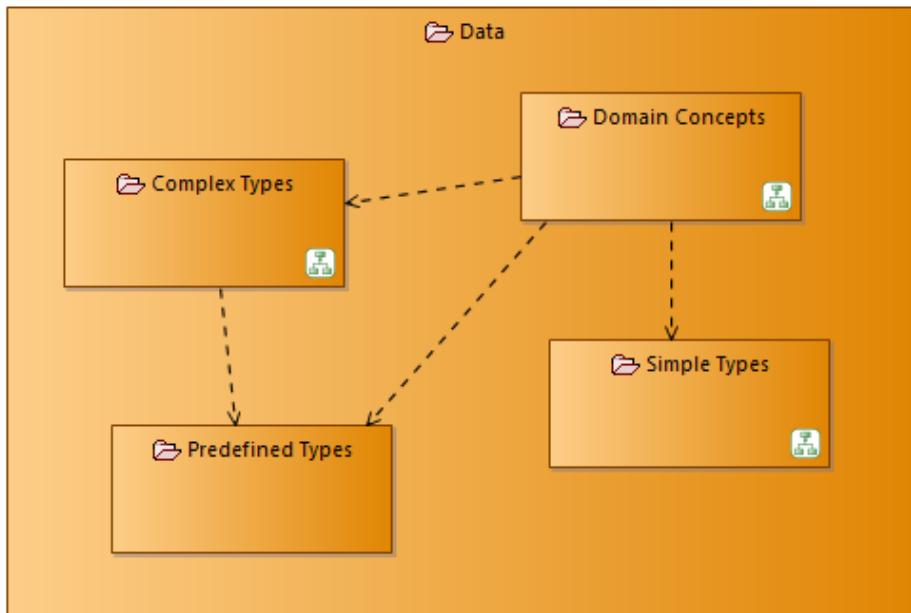
You can use both rich text and links to diagrams, external files, etc.

This should be done via the *Description* tab:

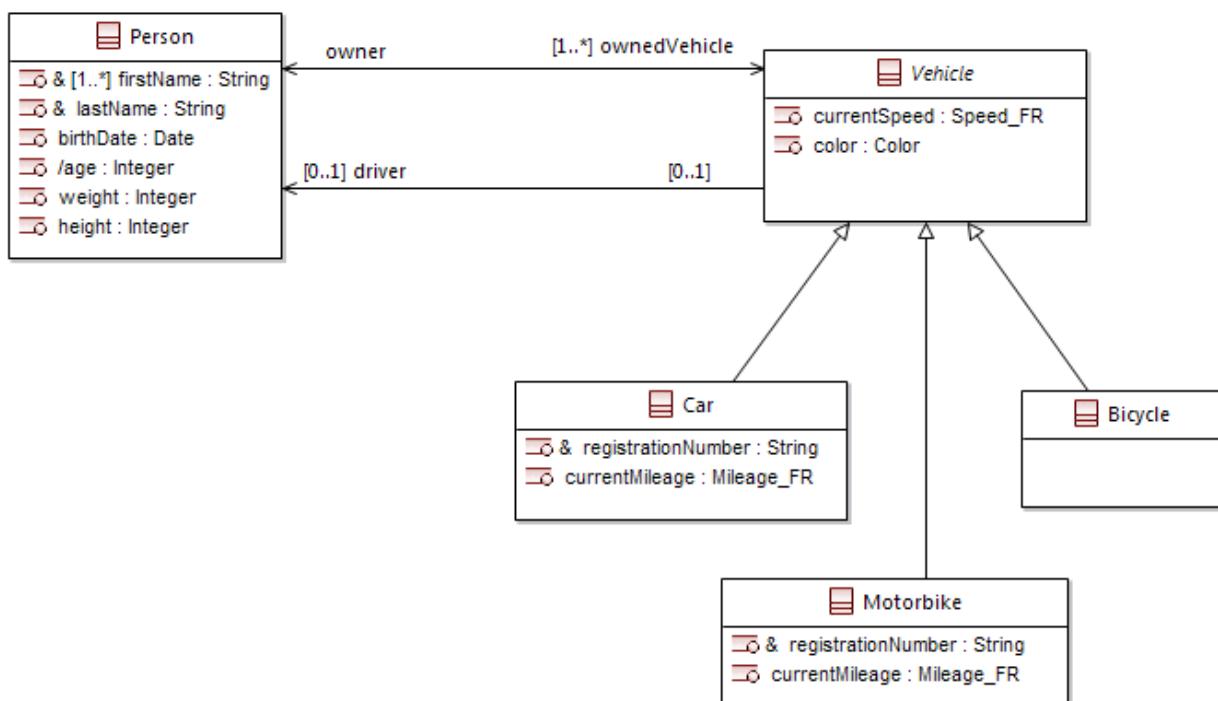


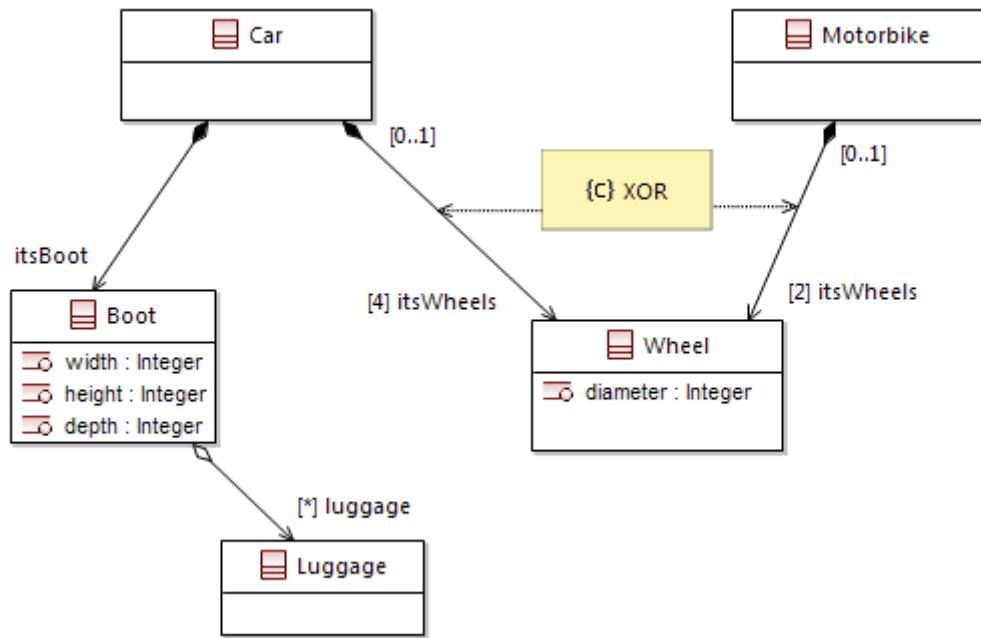
## 7 APPENDIX A – CAR DATA MODEL EXAMPLE

### 7.1 Package Dependencies



### 7.2 Domain Concepts





### 7.3 Complex and Simple Types

