Hans Karlsen

MDriven The book

Doing effective Business – by taking control of Information

Hans Karlsen, Stockholm, Sweden
2016-01-23

What is Object Constraint Language

This is from Wikipedia:

The **Object Constraint Language** (**OCL**) is a declarative language for describing rules that apply to Unified Modeling Language (UML) models developed at IBM and now part of the UML standard. Initially, OCL was only a formal specification language extension to UML.[1] OCL may now be used with any Meta-Object Facility (MOF) Object Management Group (OMG) meta-model, including UML.[2] The Object Constraint Language is a precise text language that provides constraint and object query expressions on any MOF model or meta-model that cannot otherwise be expressed by diagrammatic notation. OCL is a key component of the new OMG standard recommendation for transforming models, the Queries/Views/Transformations (QVT) specification.

Different ways MDriven relies on on OCL

As constraint definition on a class	
As description of derivation rule on derived attributes	
As description of derivation of derived associations	
As ViewModel columns and Nesting definitions	
As definitions for Visible and Enable state for ViewModel columns	
As expression of style information on ViewModel columns	
As expression for object presentation on classes	
Action Enable expression	
State machine Guards	

OCL expression must be without side effects. It is a query language and as such it is not expected to change data as the language is applied.

In MDriven we do however want to change data when appropriate – so we use the exact same syntax as OCL in something we call EAL – ExtendedActionLanguage.

We use EAL in MDriven here:

Action execute expression	
Actions in ViewModel columns Execute expression	
Class method implementations	
StateMachine Effects	

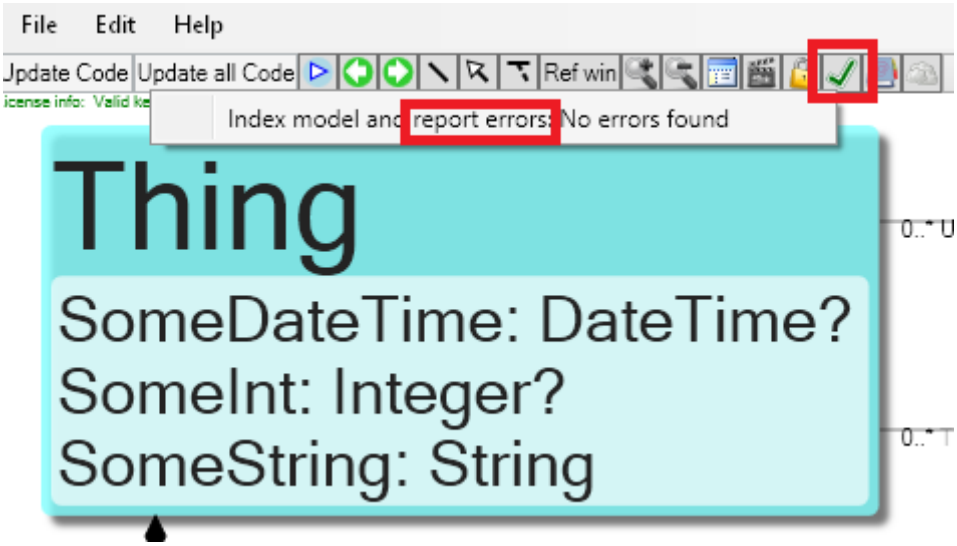
Normally the OCL expressions are executed in memory – but it is a common need to query large quantities of data in the database. Normally SQL is used in databases. To avoid having multiple query languages in MDriven we provide a subset of the OCL language that we can translate to SQL.

We call this subset for OCLps – where ps stands for Persistent Storage.

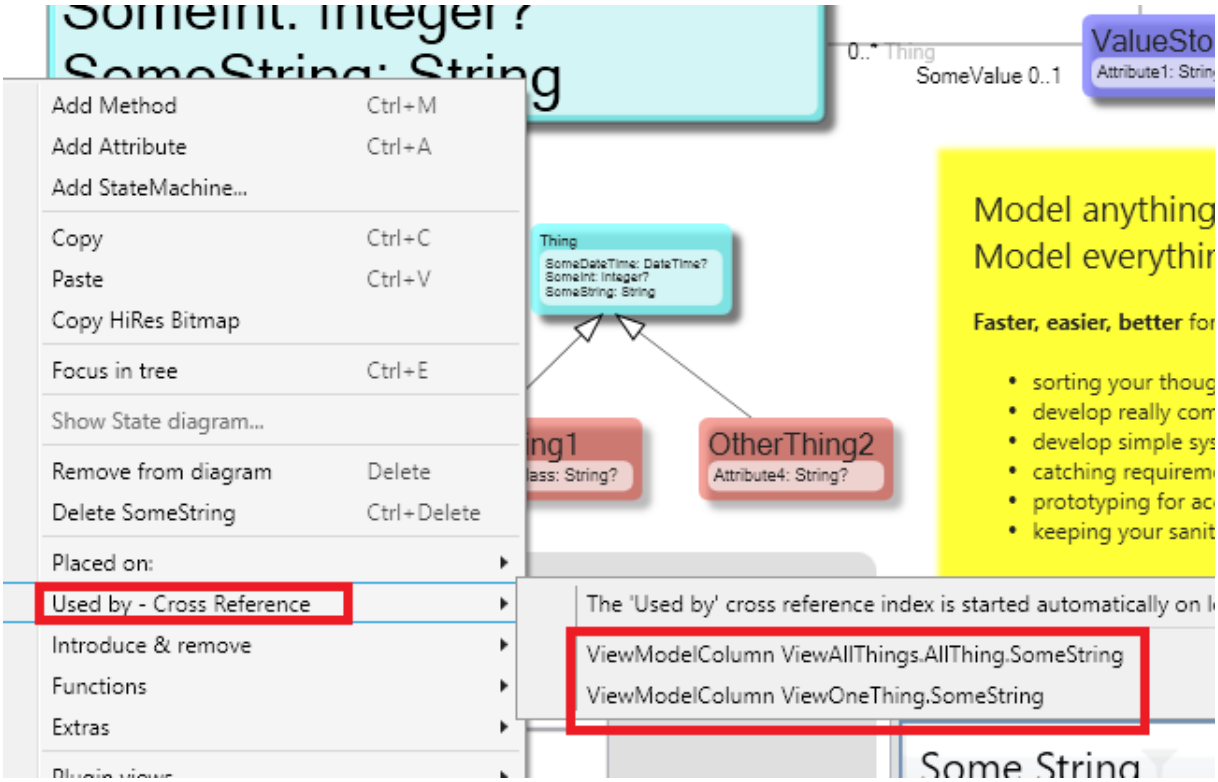
We use OCLps in MDriven here:

SearchExpressions Nestings in ViewModels	
All ViewModel Columns starting with PSExpression_	

In MDriven all the 3 types of OCL (OCL, EAL, OCLps) are dynamically typed checked whenever the model is loaded, saved or if you initiate a model check manually by clicking the mode check:



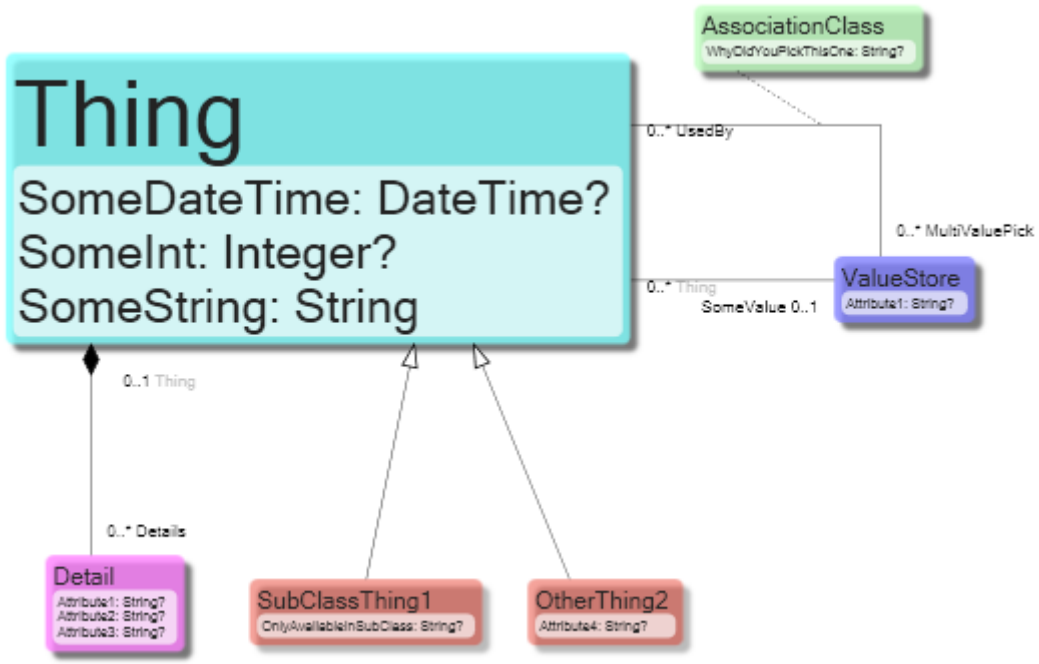
Running the ModelCheck also cross-reference you model so that you can see where things are used:



MDriven relies heavily on OCL and it is a very powerful tool to describe constraints, actions and transformations in your model.

OCL, EAL, OCLps Introduction

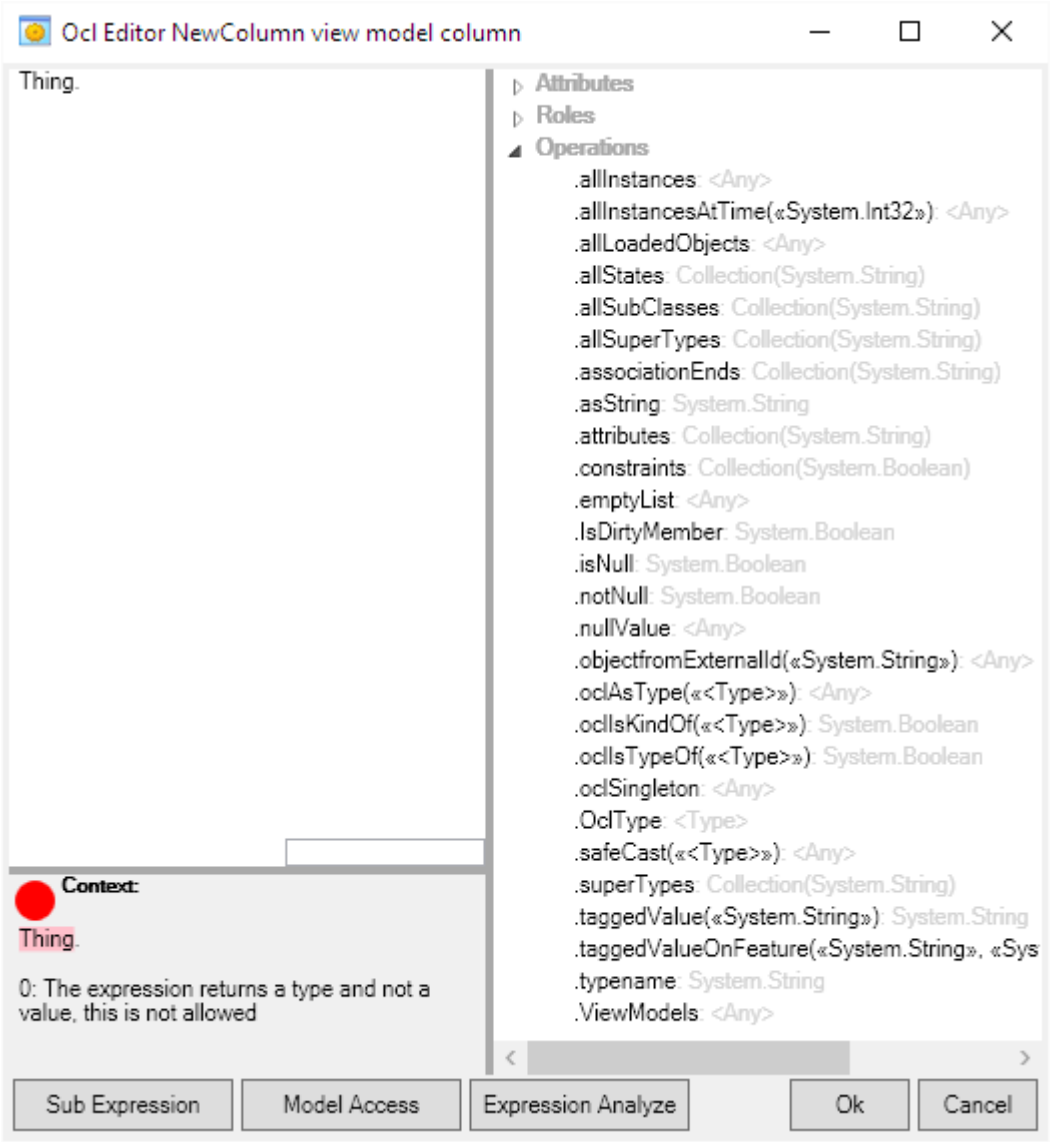
Your model is central to all expression you will handle. We will use this model to for the examples:



Thing.allinstances	Gives you a list of all Things
Things.allinstances->select(someInt>3)	Only things with someInt bigger than 3

Thing.allinstances->select((someInt>3) and (someInt<6))	Only things with someInt bigger than 3 but less than 6. Notice the extra parenthesis to or the Boolean expressions together
Things.allinstances->select(x|x.someInt>3)	Here we introduce the loop variable x. We separate the definition of x from the usage of x with the pipe sign "|". Loop variables are optional but if names are unique – but you will need to use them to give precision or to if you want to perform operations on the loop context itself.
Things.allinstances.Details	Gives a list of all detail objects that are connected to a Thing. The Detail objects that float around without a Thing will not be in the list
Things.allinstances.Details.Attribute1	A list of nullable strings from the contents from the details attribute1. Note that OCL is null-tolerant – you do not need to check if the Details exists of not – the language handles null checks for you
SubClassThing1.allinstances.Details	Inherited features of classes are directly accessible
Thing.allInstances->select(x|x.safeCast(SubClassThing1).OnlyAvailableInSubClass='hello')	Filtering on Specialization is done with an operator SafeCast. This is null safe so for all objects that do not fit the profile the expression returns false

This was description of the allinstances operator. It is a common operator. To find all available you can open the OCL-Editor and type in a class:

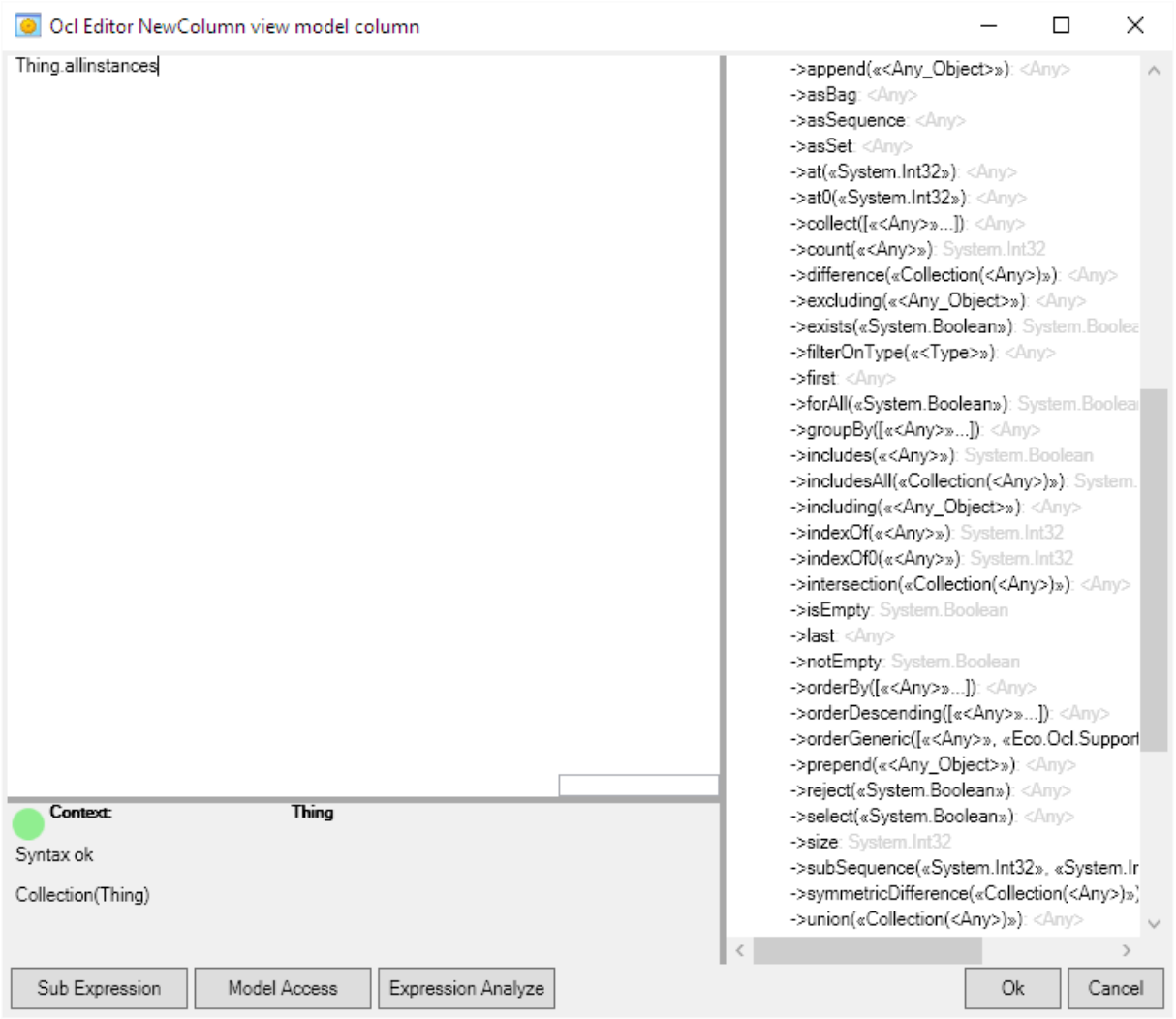


The operations listed do this:

Allinstances	All the objects of the class
allinstancesAtTime	Using versioning you can get all the instances that existed at a time
allLoadedObjects	All the currently loaded instances
AllStates	Meta information about available states in state machines the class may contain
allSubClasses	Meta information on all the sub classes this class has
AllSuperTypes	Meta information on all the super classes – in inheritance order the class has
associationEnds	Meta information on all the associationEnds
Asstring	The string representation of the class – the asString operation is available on everything
Attributes	Meta information about what attributes the class has
Contraints	Meta information on what constraints the class has

Emptylist	Returns an empty list typed to hold objects of the class
IsDirtyMember	
isNull	
nullValue	A typed null value
objectFromExternalId	An external identity will be resolved to the object
oclAsType	The type of the class
oclIsKindOf	This is to if a class is a subclass or a the class itself and not unrelated
oclIsTypeOf	Returns true if
oclSingleton	Classes that implements the Singleton pattern – by setting IsSingleton=true – will return the singleton instance with this operator
OclType	
safeCast	
SuperTypes	
TaggedValue	Meta information on tagged values set in the class
TaggedValueOnFeature	Meta information on Tagged values set on a named feature in the class
Typename	The type name as a string
ViewModels	A tuple with the ViewModels for this class a members

Once you have a collection of objects there are certain operators that are applicable to it. Again you can use the OCL-Editor to see what they are:



->append	Add another object last
->asBag	Collapses to one list
->asSequence	Collapses to one list
->asset	Remove doublets
->at	Get the objects at X where the first index is 1
->at0	Get the objects at X where the first index is 0
->collect	Iterate over the collection and build a tuple result
->count	Count how many meet a certain criteria
->difference	The difference between 2 collections
->excluding	The collection except this single object
->exists	Are there any objects that fulfill the criteria
->filterOnType	Only keep the ones of a certain type
->first	Return the first object
->forAll	Iterate all that fulfills the critera
->groupBy	Build collection of tuples grouped by some aspect
->includes	Does the collection include the object
->includesAll	Does the collection include the whole other collection
->including	

->IndexOf	The 1 based index of an object in the collection possibly -1 if not existing
->indexOf0	The 0 based index of an object in the collection possibly -1 if not existing
->intersection	The intersection of two collections
->isEmpty	Returns true if the collection is empty
->last	Returns the last object in the collection
->notEmpty	Returns true of the collection is not empty
->orderBy	Sorts the collection on one or more properties
->orderDescending	Sort the from biggest to smallest
->orderGeneric	Sorts the list of properties with interchangeable sort order: (expr1, OclSortDirection::ascending, expr2, OclSortDirection::descending...)
->prepend	Add an object in front of the list
->reject	Returns the objects not matching the criteria
->select	Returns the objects matching the criteria
->size	Returns the number of elements in the collection
->subsequence	Returns a smaller collection from a start to stop
->symmetricDifference	The symmetric difference between the collections; ie all the objects in collection1 or collection2 but not in both
->union	The set of objects in collection1 and objects in collection2

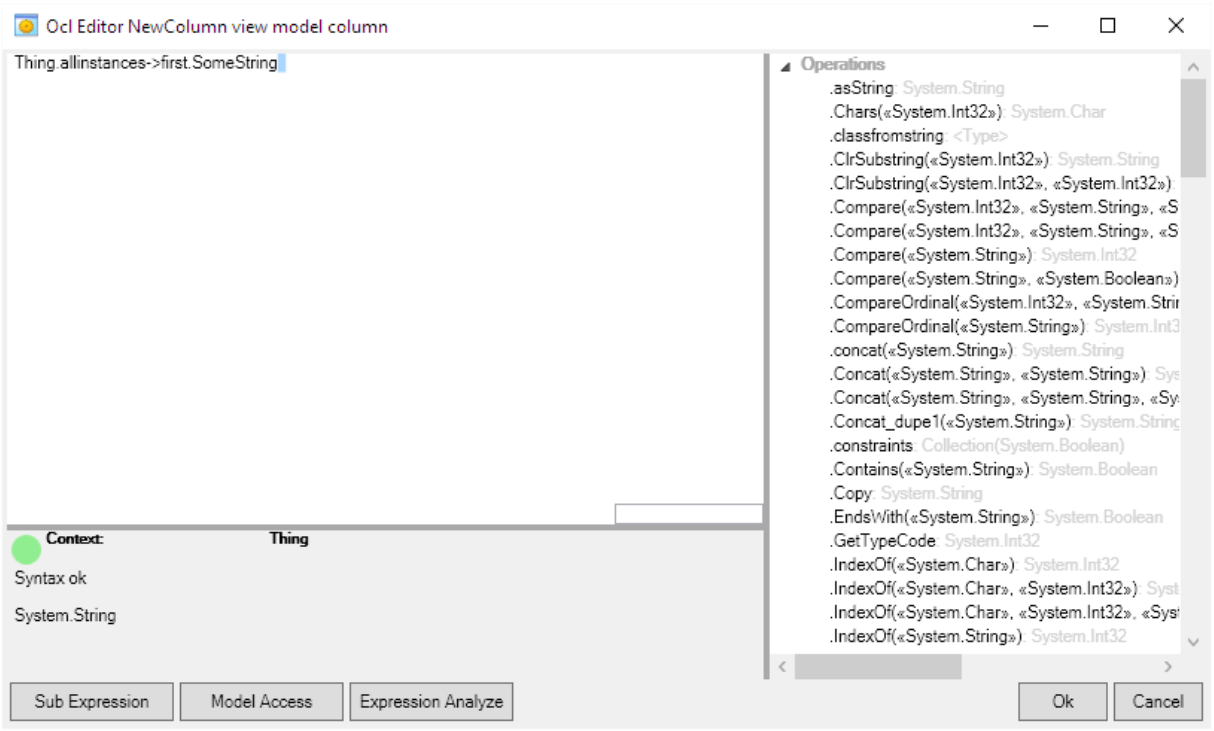
One important aspect of OCL that is worth noting is that it expands lists of lists to just a list. An example in plain English; Thing.allinstances.Details – this will come back as a set of details that are all the details from all the Things. If OCL had not expanded lists automatically one could have expected a set of sets containing the details per thing. But this is not the case. The automatically expansion of lists of lists is sometime referred to as flattening of a collection – referring to the reduction of topology in the result.

Some OCL examples

Bag{'5','1','2','2','3','4'}->ascommalist	5, 1, 2, 2, 3, 4
Bag{'5','1','2','2','3','4'}->union(Bag{'1','2','2','3','6'})->ascommalist	5, 1, 2, 2, 3, 4, 6
Bag{'5','1','2','2','3','4'}->union(Bag{'1','2','2','3','6'})->asset->ascommalist	5, 1, 2, 3, 4, 6
Bag{'5','1','2','2','3','4'}->union(Bag{'1','2','2','3','6'})->asset->orderby(a|a)->ascommalist	1, 2, 3, 4, 5, 6
Bag{'5','1','2','2','3','4'}->intersection(Bag{'1','2','2','3','6'})->orderby(a|a)->ascommalist	1, 2, 2, 3
Bag{'5','1','2','2','3','4'}->Difference(Bag{'1','2','2','3','6'})->orderby(a|a)->ascommalist	4, 5

Bag{'5','1','2','2','3','4'}- >SymmetricDifference(Bag{'1','2','2','3','6'})- >orderby(a|a)->ascommalist	4, 5, 6

If you are in the context of a simple type like string, double, int, datetime or Boolean MDriven will expose the simple operations that are available in the .net Framework. Testing this in the OCL-Editor:



In this case it is a string that is the result – and we can to string operations like compare, indexof, split etc.

The numeric types float, double, decimal and int are sort of apples of the same tree and MDriven expose ways to go from all numeric types to decimal. The operator is called toDecimal.

Certain important constructs

Some constructs are more returning than others as an everyday business developer with MDriven. Your favorite ways to express yourself may be different from mine but these are some of my returning expressions:

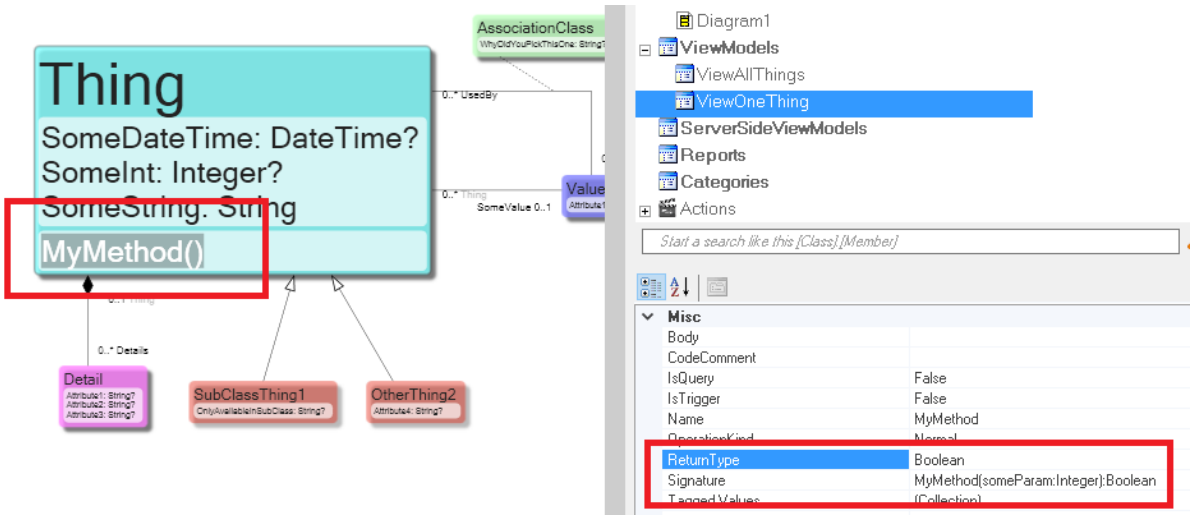
let z= Things.allinstances-
>select(x|x.someInt>3)->size
in
(
If z>4 then
'There are more than 4
Things with SomeInt>3'
else
'There are '+z.asstring+'
Things with SomeInt>3'
endif
	I use the "let" construct to assign a result of an expression to a temporary variable. This so I do not need to repeat myself in the testing of z>4 and the z.asstring

)	
Thing.allinstances->groupby(x|x.SomeValue)	Groupby , this expression has the type Collection(SomeValue:ValueStore+List:Collection(Thing)) so I get a list of SomeValue and for each a list of the things that use it
Thing.allinstances->collect(x|x.SomeValue.UsedBy.SomeInt->sum, x.SomeValue.UsedBy->collect(y|x, y.Details))	Nested collecting. This expression get the type Collection(Part1:System.Int32+Part2:Collection(Thing:Thing+Details:Collection(Detail))) The ability to nest collections is very powerful. In this case I start with all Things – grab the SomeValue valueStore– check what other things has this set via the association MultiValuePick and for these I sum up all SomeValue plus grab the Details. This kind of multi level collect-usage is very handy when summarizing deep object hierarchy's on different levels
' '.Chars(0)	This expression returns System.Char. Since OCL has not literal way to input a single character – it is always interpreted as string – this trick will help when calling certain .net functions that takes characters as arguments

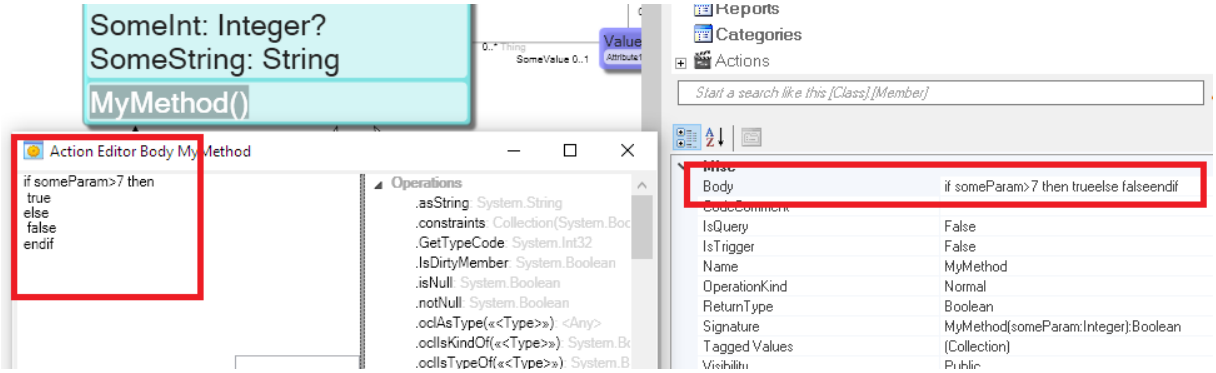
if true then
'this returns a string'
else
0.asstring
endif
	All return paths must result in the same type. Since OCL is a functional language we must be consistent. This is one way to get the expression correct ; add.asstring after the zero

Thing.allinstances->select(someint>3) ValueStore.allinstances->select(usedby->notEmpty).Thing Like this: Thing.allinstances->select(someint>3) ->intersection(ValueStore.allinstances->select(usedby->notEmpty).Thing)	Working with OCLps – expressions that will be translated into SQL and executed in a database- it is sometimes easier to do one expression per complex constraint and at the end intersect all the expressions together.
self	When you are in the context of an object you can use the variable self to access properties of this

You may define methods in classes to and implement these with OCL:



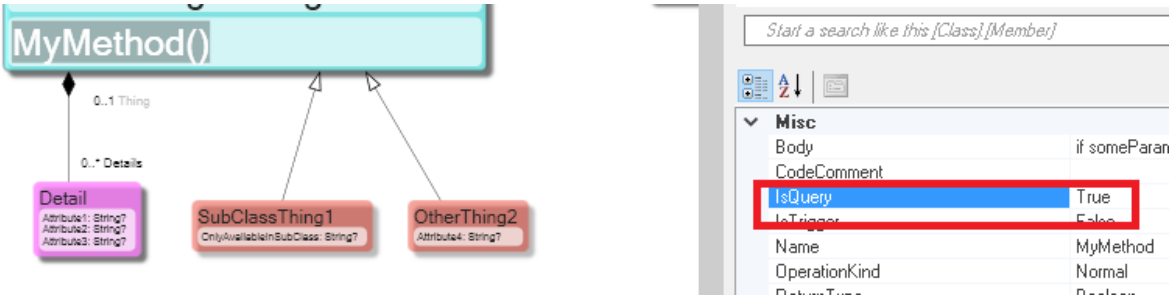
You will in the OCL implementation in the Body-property:



Notice that since this was a method MDriven will treat you OCL as EAL – something that is allowed to have side effects.

In this case our method do not have any side effects and I may want to be able to use this method in OCL.

But trying to use it in OCL will not succeed. Methods with side effects are not recognized by OCL . There is a flag on the Method definition called IsQuery and if this is set we “promise” that it does not have intentional side effects. Now it is seen by OCL:



We can then use our IsQuery method in any expression in OCL. Thing.allinstances->select(x|x.MyMethod(x.SomeInt))

EAL differences

When using EAL one often want to stack expressions after each other. To allow for this EAL has introduced an expression separator: The semicolon ";". This means that you can build method bodies that do several things like this:

```
self.SomeDateTime := DateTime.Now;
self.SomeInt := 27;
self.SomeString := self.SomeDateTime.ToString('yyyy-MM-dd')
```

In EAL we use := to assign but = to compare.

In EAL we can also create new objects Thing.Create

Worth noting is that the expression separator ";" can only be used between statements. So writing this 'a string';0 is of type integer. But writing this 'a string';0; is of unknown type and hence wrong – the last statement that the parser expect after the last ; is not found.

OCLps differences

OCLps is a subset of OCL. No side effects, and you cannot use your methods even if they are marked with IsQuery. The collect, groupby and other operators that return tuples are not supported. The reason is that the main use of OCLps is to return a list of identities based some criteria's from select or difference or the like. Once MDriven has the set of identities we will load the corresponding objects – then you can take over with normal OCL.

Summary OCL

I often get the question if OCL is capable of doing everything we need to do in a line of business application. The answer is that as long as the arguments and result is representable in your model – yes it will do anything. Sometimes you have external or ambient data not accessible from the model – then you cannot use OCL – until you make that data available.

Not only can you do everything you need – it also comes out in small easily interpreted snippets of text that very much looks just like the requirements you are set to implement.

I like to compare OCL and modeling with Calculus. In math you can discuss numbers and operators on those number in plain language – but you seldom do since it will be error prune and require you to use a lot of words for even simple things. Instead everyone actually doing math uses calculus notation to write up expressions. The expressions are often reduced to the smallest possible – so that they are easily understood and ready to be used for a purpose.

Use OCL for the same reason but not on only numbers but on all your designed information. Imagine a world without a good way to declaratively work with math. In this world we would probably not have been able to do much cool technology. The ability to convey compact math between people is very good for mankind. I am certain that a good compact way to convey rules on information is equally important – if not even more usable – for mankind.