

# **Creating an ECO business model**

## Table of Contents

<b>Overview</b>	<b>1</b>
<b>Prerequisites and goals</b>	<b>2</b>
<b>Creating an ECO model</b>	<b>3</b>
<b>Creating a class diagram</b>	<b>4</b>
<b>Adding attributes to the class</b>	<b>5</b>
<b>Adding associations between classes</b>	<b>8</b>
<b>Adding object validation</b>	<b>11</b>
<b>Generating source code</b>	<b>13</b>
Inspecting the source code	13
<b>Derived members</b>	<b>16</b>
Creating OCL derived members	16
Creating code derived members	17
Creating derived settable members	18
<b>Default string representation</b>	<b>20</b>
<b>Summary</b>	<b>21</b>
<b>Index</b>	<b>a</b>

# 1 Overview

Although it is possible to add your business classes to your UI application it is recommended that you instead create a class library. This will reduce the possibility of cross contaminating source code in your business and presentation layers, which might prevent the same business model from being used by a secondary user interface such as ASP .NET.

# 2 Prerequisites and goals

## Prerequisites

To successfully follow this document the user should have ECO installed. An understanding of UML would be an advantage.

## Goals

By the end of this document you will be able to

- Create a basic ECO model.
- Added ECO classes.
- Add UML attributes (aka properties) to those classes.
- Add associations between classes to allow them to hold references to each other.
- Add object validation expressions using constraints.
- Inspect generated source code and understand the additional .NET attributes generated.

## 3 Creating an ECO model

1. File->New->Other.
2. Select the ECO node in the TreeView and select the ECO project wizard, then click OK.
3. Select "Package in DLL"
4. Name the project "CapableObjects.QuickStart.BusinessModel".
5. Click "OK".

Your project is now created. First we need to rename the package to something more meaningful.

6. Select the [Model view] tab.



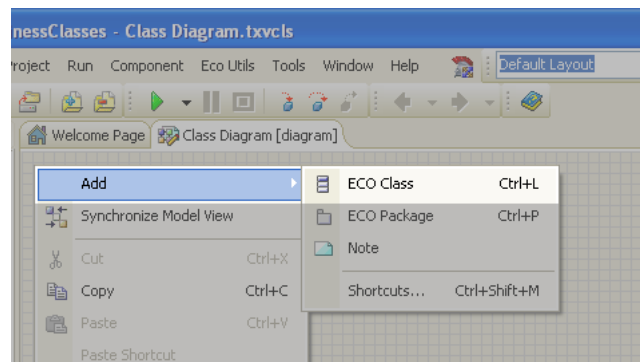
7. Right-click on the TreeView node entitled "Package\_1" and select "Rename".
8. Change the name of the package to "QuickStartPackage".

You now have a project containing an ECO business model that when compiled will create a DLL which may be used in multiple projects, next some business classes will be added to the model.

# 4 Creating a class diagram

Next a class will be added to the model.

1. Right-click on the QuickStartPackage node in the Model View and select Add->Add ECO Class Diagram, a drawing surface for the new class diagram will appear in place of the source code editor window.
2. Right-click the drawing surface and select Add->ECO Class.



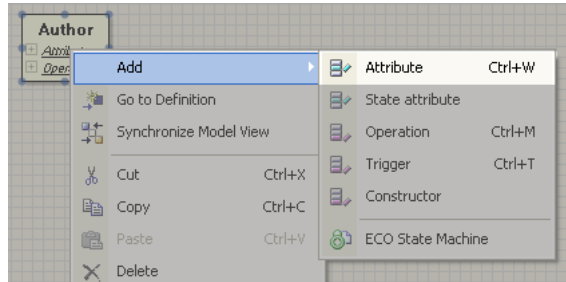
3. When the menu option is selected a new class element will appear on the diagram drawing surface. By default the name of the class will be in edit mode, you can set the name of the class by immediately typing, if the class name is not in edit mode you can invoke it simply by single-clicking the name on the diagram and then beginning to type, or you can select the class and change its name in the Object Inspector window.
4. Rename the class "Author".



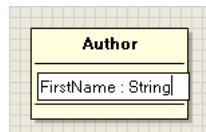
## 5 Adding attributes to the class

In the UML class properties are known as "attributes". When describing UML the term "attribute" will be used, when discussing source code the term "property" will be used instead.

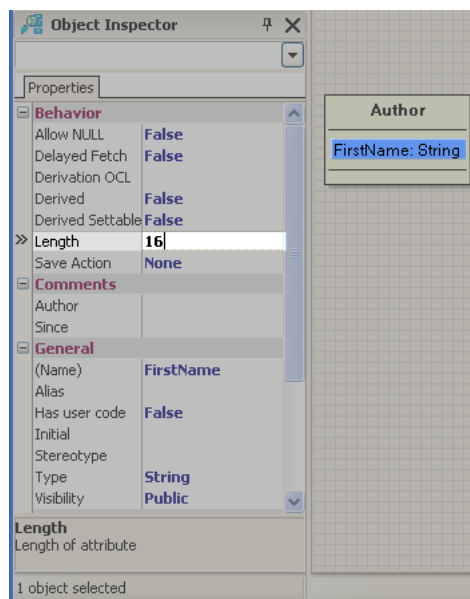
1. Right-click the Author class and select Add->Attribute. A UML attribute is the equivalent of a property in a class.



2. Again the name of the added item will be in edit mode so it is possible to set the name and type of the attribute by typing in the name separated by a colon. Type in "FirstName : String".



3. The new attribute may be further specified by modifying its properties in the Object Inspector, change "Length" to "16".



4. Now add the following additional UML attributes to the Author class:

Name	Type	Length
EmailAddress	String	64
LastName	String	32
Password	String	32
Salutation	String	16
ID	AutoInc	

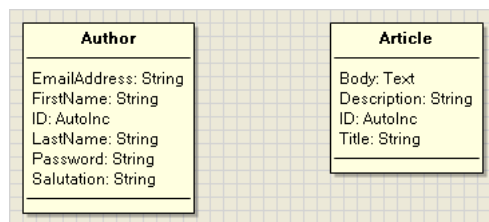
5. Now create a new class named "Article" and add the following UML attributes:

Name	Type	Length
Title	String	64
Description	String	255
Body	Text	-1
ID	AutoInc	

You may notice that the type of the "Body" attribute is "Text" rather than "String". This is a custom attribute type which will map in code to a string property on the class, if ECO is instructed to generate the database automatically it will create a column capable of storing a text/memo value. A length of -1 has been used to illustrate that there is no length limit. Custom types include:

Name	Delphi type	C# type	Description
AutoInc	Integer	int	Creates a table column that has a DB assigned auto-incrementing value.
Blob	TBytes	byte[]	Creates a table column that is capable of storing large binary data.
Image	TBytes	byte[]	Creates a table column that is capable of storing large binary data.
Text	string	string	Creates a table column that is capable of storing large textual data.

Your class diagram should now look something like this:



6. Create another class named "ArticleType" and add the following UML attributes:

Name	Type	Length
ID	AutoInc	
Name	String	32

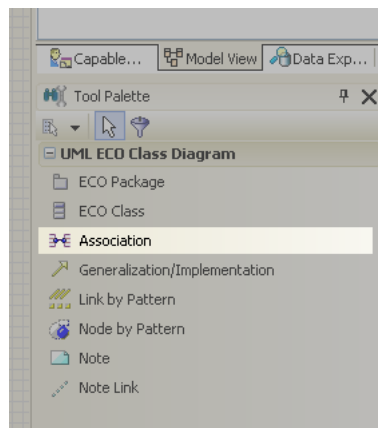
## 6 Adding associations between classes

An association is a way of allowing ECO classes to hold references to each other. Instead of adding a UML attribute to a class to hold a reference to another object a class association should be used because:

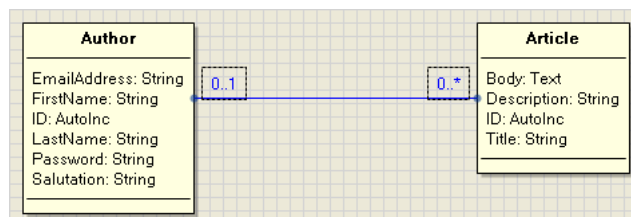
1. If the association is two-way (a property added to both classes on each end of the association) then the properties' object references at both ends will be kept in sync' with each other.
2. It is possible to specify delete rules so that deleting an object on one end of the association has a specified effect on objects at the other end of the association; cascade delete, prohibit delete, or unlink.
3. Object instances at the other end of an association will be automatically fetched from the database when referenced via an association; PurchaseOrder.Lines[0] would automatically fetch the first OrderLine in the PurchaseOrder's "Lines" association if it had not already been retrieved.
4. If ECO is instructed to create the database then many-to-many associations will result in a "link table" being created in the database.

Whenever one of your business classes needs to hold a reference to an instance of another ECO class you should always use an association. To add an association to the current model perform the following steps:

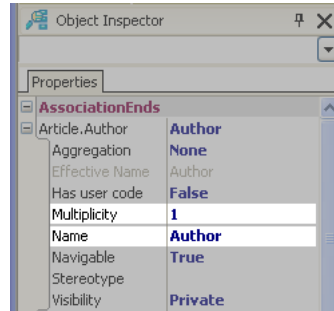
1. On the tool palette select "Association".



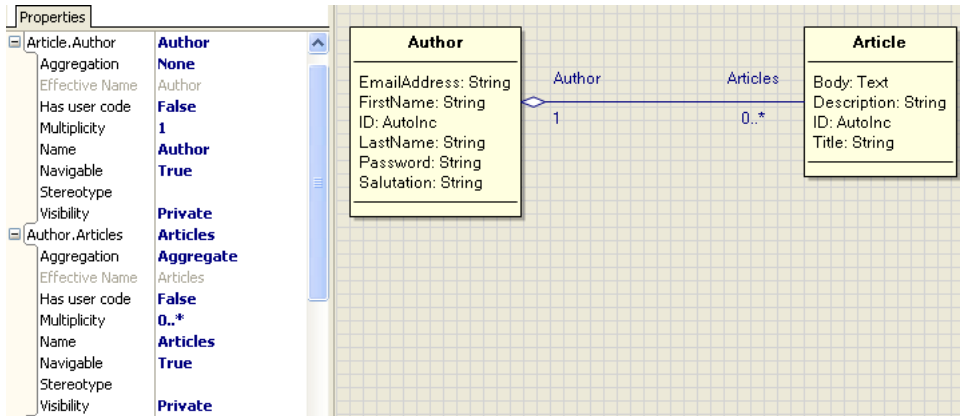
2. Now click on the Author class, hold the left mouse button down, and then move the mouse over to the Article class (as if you were dragging and dropping the Author class onto the Article class). As you move your mouse you will see an association line being drawn, to complete the operation release the left mouse button whilst the cursor is over the Article class, the association line will then connect to both classes.



3. In the Object Inspector expand the property entitled "Article.Author" by clicking the [-] symbol to the left of it. Set the nested "Name" property to "Author" and set the multiplicity to 1.



4. Now expand the property entitled "Author.Articles" and set the nested "Name" property to "Articles".
5. Click the drop down button to the right of the embedded "Aggregation" property, and from the list select "Aggregated". After rearranging the role names and multiplicity labels on the diagram drawing surface your diagram should now look something like this.



**Aggregation**

UML aggregation is a visual indication to the programmer, in ECO it is also a way of specifying delete rules.

Aggregation	Appearance	Delete behavior
None		Objects at the aggregated end of the association (Article) are unlinked from the deleted object.
Aggregate		The object at the non-aggregated end of the association (Author) is prohibited from being deleted if any aggregated objects (Article) are associated with it.
Composite		Objects at the composite end of the association are considered to be "part of" the parent object, deleting the parent object (Author) will also delete its owned parts (Article).

Note that this is the default behavior, it is still possible to specify a different delete action for association ends in the Object Inspector.

**Exercise for the reader**

Now add an association between the Article and ArticleType classes. Use the correct aggregation kind so that an ArticleType may not be deleted if any articles are associated with it. It should be possible to reference a single ArticleType instance from multiple Article instances.

# 7 Adding object validation

ECO allows you to specify a list of OCL expressions against the classes in your model, during runtime it is possible to obtain a list of modeled constraints and evaluate their expressions in order to determine whether or not the current object instance passes validation. The main benefit of having validation constraints in the model is that it becomes very easy to ensure consistent validation is available in your application no matter which part of your application performs the validation; the same validation rules will apply to your business model whether you have developed a form, ASP .NET, or web service interface.

To add constraints to your model follow these steps:

1. Select the [Model view] tab.
2. Expand the CapableObjects.QuickStart.BusinessClasses node in the TreeView.
3. Expand the embedded QuickStartPackage node.
4. Single-click on the Article class.
5. Click the [...] button to the right of the "Constraints" property in the Object Inspector.
6. In the editor form that appears click the "Add" button, a new empty constraint will appear in the list on the right of the form.
7. In the "Name" property for the new constraint type

```
Title must be at least 5 characters long
```

8. In the "OCL Expression" property type

```
self.Title.Length >= 5
```

Note: There is an OCL editor dialog available by clicking the [...] button to the right of the OCL Expression property to help you to create valid OCL expressions.

Now add the following additional constraints to the Article class:

Name	OCL Expression
Description must be at least 50 characters long	self.Description.Length >= 50
Body must be at least 50 characters long	self.Body.Length >= 50
Author required	self.Author->NotEmpty

Note: "self" is case-sensitive.

Now add the following constraints to the Author class, you may want to copy / paste the OCL expression for the last entry to ensure it is correct:

Name	OCL Expression
Salutation must be at least 2 characters	self.Salutation.Length >= 2
First name must be at least 2 characters	self.FirstName.Length >= 2
Last name must be at least 2 characters	self.LastName.Length >= 2
Password must be at least 6 characters	self.Password.Length >= 6
Email address required	self.EmailAddress.Length > 0
Invalid email address	(self.EmailAddress.length = 0) or (self.EmailAddress.regExpMatch("\\w+([-.]\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*"))

The Author constraints will reveal that the "Email address required" is invalid if no email address is entered, and that the "Invalid email address" is invalid if an email address is entered but it is not a valid email address. The regular expression was obtained from <http://www.regexplib.com>.

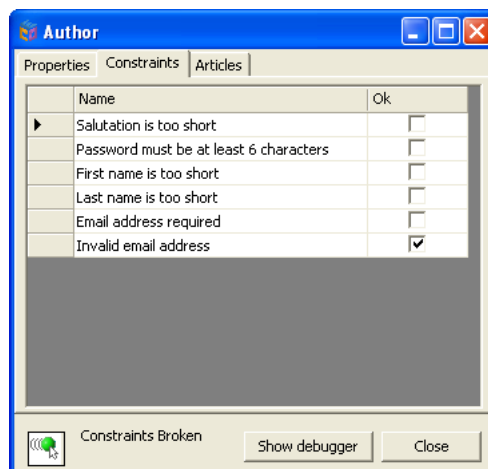
Note: Use the [...] button next to the OCL Expression property to invoke the OCL editor. If any error is made entering the OCL expression it will be easy to spot.

Now add the following constraints to the ArticleType class.

Name	OCL Expression
Name must be unique	ArticleType.allInstances->select(name.sqlLikeCaseInsensitive(self.name))->size = 1
Name required	self.Name.Length > 0

### Testing the constraints

During prototyping later in this series, at the bottom of the auto-generated Author form you will see a label that reads "Constraints broken". Double-click this label and you will be presented with a list of constraints.



## 8 Generating source code

A simple business model has now been created, however, it currently only exists as a design. In order to implement the design source code must first be generated. At the top of the [Model View] tab click the "Update ECO source" button. Now that the source has been generated, compile the project to produce its DLL file. ECO designers use reflection to read model information during application development, so it is advisable to ensure the project containing the model is compiled once you have finish making modifications and have updated your model's source code.

### 8.1 Inspecting the source code

Open the source code that has been generated. One of the first classes you will see is the class for the package itself.

```
[UmlElement('Package', Id='21871a40-77a6-4143-a27b-8f6553bf33c')]
[UmlMetaAttribute('ownedElement', TypeOf(Author))]
[UmlMetaAttribute('ownedElement', TypeOf(Article))]
[EcoCodeGenVersion('2.0')]
QuickStartPackage = class
public
type
  [UmlElement('Association', Id='feb9645c-4b9a-4815-9978-b7aa387c2fc2')]
  AuthorAuthorArticleArticles = class
  end;
end;
```

The class is identified as a Package using the UmlElement .NET attribute. Each of the classes belonging to this package are listed above the class and identified using the UmlMetaAttribute .NET attribute. Within the class itself there are nested classes, there will be one per association within your model. In this case the association from Author to Article was automatically named AuthorAuthorArticleArticles. An embedded class is created for each association in order to provide a place for ECO to generate .NET attributes holding information such as whether the association is persistent or transient (does not get saved to the persistent storage).

Above the Author class itself you will see a collection of .NET attributes describing meta information from the model.

```
[UmlElement(Id='64bd2b42-f7e4-41c0-bacb-10c6f8ff9b6d')]
[UmlMetaAttribute('constraint', 'Salutation must be at least 2 characters=' +
  'self.Salutation.Length >= 2')]
[UmlMetaAttribute('constraint', 'Last name must be at least 2 characters=' +
  'elf.LastName.Length >= 2')]
[UmlMetaAttribute('constraint', 'Password must be at least 6 characters=' +
  'lf.Password >= 6')]
[UmlMetaAttribute('constraint', 'Email address is invalid=(self.EmailAddre' +
  'ss.length = 0) or'#13#10'(self.EmailAddress.regExpMatch('\w+([-.]\\w+)' +
  '*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*'))')]
[UmlMetaAttribute('constraint', 'First name must be at least 2 characters=' +
  'self.FirstName.Length >= 2')]
[UmlMetaAttribute('constraint', 'Email address required=self.EmailAddress.' +
  'Length > 0)']
```

The class is uniquely identified using the UmlElement .NET attribute with a GUID. In addition you will see the constraints that were added earlier in this document. These .NET attributes are also used to hold other information such as ECO state

machines, so this list will undoubtedly grow as your model approaches completion.

The information within the class itself is contained within a region named "ECO generated code". Typically you would leave this collapsed to make reading the source code easier. The first part of the class defines an embedded class named "Eco\_LoopbackIndices". This class contains a set of constants that identify the members of the Author class by index. This index is used when you need to obtain an ECO IProperty reference for a class member so that you can read model information at runtime for example.

```

Author = class(System.Object, ILoopBack)
{$REGION 'ECO generated code'}
public
type
  Eco_LoopbackIndices = class
  public
  const
    Eco_FirstMember = 0;
  const
    Eco_MemberCount = (Eco_FirstMember + 6);
  const
    Salutation = Eco_FirstMember;
  const
    FirstName = (Eco_LoopbackIndices.Salutation + 1);
  const
    Password = (Eco_LoopbackIndices.FirstName + 1);
  const
    EmailAddress = (Eco_LoopbackIndices.Password + 1);
  const
    LastName = (Eco_LoopbackIndices.EmailAddress + 1);
  const
    Articles = (Eco_LoopbackIndices.LastName + 1);
  end;

```

The following section contains some ECO support source code:

```

public
  function AsIObject: IObjectInstance;
  procedure set_MemberByIndex(index: Integer; value: TObject); virtual;
  function get_MemberByIndex(index: Integer): TObject; virtual;

```

- **AsIObject:** Provides access to ECO framework functions for any object. For example `SomePerson.AsIObject.Delete` would mark the object for deletion. The `IObject` interface (which is the superclass of `IObjectInstance`) is used throughout the ECO framework. `IObject` is ECO's way of referring to object instances.
- **set\_MemberByIndex / get\_MemberByIndex:** OCL expressions will read object values directly from its cache. When the developer needs to implement code within the property they will mark the property `HasUserCode=True` in the model. Any property that is marked in such a way will automatically have additional source code generated within these methods to enable the OCL evaluator to read/write values using the property accessors directly rather than having to restore to reflection.

Each UML attribute will generate a property on the class, along with a read/write method. The "get" method will return the cached value from the `EcoSpace`, the "set" method will set the value within the `EcoSpace` cache. Finally a property is declared with the information from the model, information such as the maximum allowed length of the property.

```

function get_Salutation: String;
procedure set_Salutation(Value: String);
[UmlElement(Id='8aea05e2-f76b-47f7-9bf3-ef2d1d2c8736',
Index=Eco_LoopbackIndices.Salutation)]

```

```
[UmlTaggedValue('Eco.Length', '16')]  
property Salutation: String read get_Salutation write set_Salutation;
```

Each association will also generate a property on the class returning either a single instance of another class type, or a collection of instances. In this example the Author.Articles role (association end) has been declared. You will see additional model information such as

1. The name of the association. This is the association class embedded within the package class.
2. The multiplicity of the role.
3. The aggregation used.

```
function get_Articles: IList<Article>;  
[UmlElement('AssociationEnd', Index=Eco_LoopbackIndices.Articles, Id='83' +  
'057ffb-f630-4de8-9996-02dd8bf587e8')]  
[UmlMetaAttribute('association',  
TypeOf(QuickStartPackage.AuthorAuthorArticleArticles), Index=1)]  
[UmlMetaAttribute('multiplicity', '0..*')]  
[UmlMetaAttribute('aggregation', AggregationKind.Aggregate)]  
property Articles: IList<Article> read get_Articles;
```

Again this list may grow as your model becomes more complete.

Finally, you may notice the following two constructors in the class:

```
constructor Create(content: IContent); overload;  
constructor Create(serviceProvider: IEcoServiceProvider); overload;
```

The first constructor is executed when the object is recreated from the persistent storage. This occurs when a previously created object is saved, the application terminates, and then the object instance is retrieved when the application next runs.

The second constructor is executed when an entirely new instance is created rather than recreated from the persistence storage.

## 9 Derived members

Derived members allow the developer to create attributes or associations in the business model that are calculated from other members in the business model. These members may be calculated in the following ways:

Type	Description
OCL derived	An OCL expression is entered in the business model for the derived member, when the value of this member is requested the OCL expression is evaluated and a result is produced.
Code derived	The developer creates a method on the class with a specific name and parameters, when the value of this member is requested the method is executed and returns the calculated value.
Reverse derived	For reading a value this is exactly the same as a normal code derived member. In addition the programmer is able to implement an additional method which is responsible for storing the reverse-engineered values elsewhere. For example, setting a reverse derived member "Age: Integer" would adjust the year in "DateOfBirth : DateTime".

When the value of a derived member is first requested ECO will either evaluate the specified OCL expression or execute the method written by the developer and then return the result. This result will then be automatically cached within the EcoSpace and any subsequent requests will return the same value from the cache, saving the application from having to recalculate the value each time; when the calculations involved are intensive this can result in a vastly improved user experience.

To ensure the cached value does not become stale ECO employs a subscription approach (observer pattern). A change notification request is placed on each element that is used to derive the result. When an element value changes the cached value is discarded, requesting the derived value after its cached value has been discarded will result in the value being recalculated and cached again. Note that derived values are only ever calculated on demand and if there is no previously calculated cache value stored.

Note that derived UML attributes in the class appear with a forward slash in front of their name, this acts as a visual indicator to show that the value is calculated on demand rather than being stored in the persistent storage.

### 9.1 Creating OCL derived members

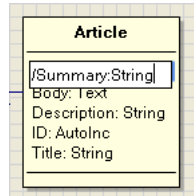
1. Open the design surface for the class diagram.
2. Right-click the Article class and add a new attribute.
3. Name the attribute "Summary" and specify its type as "String".
4. Set its "Derived" property to "True".
5. Set its "Derivation OCL" property to

```
self.Title + ' by ' + self.Author.asString
```

7. Click the "Update ECO source code" button at the top of the [Model View] tab.
8. Compile the project.
9. Save all changes.

When creating an OCL derived association ECO will automatically place the required subscriptions on each of the elements used to create the resulting value.

Note: A short-cut to mark a new attribute as derived is to start the name with a forward slash when typing it's name into the in-place editor.



## 9.2 Creating code derived members

1. Open the design surface for the class diagram.
2. Right-click the Author class and add a new attribute.
3. Name it "FullName" and specify its type as "String".
4. Set its "Derived" property to "True", but this time do not specify a "Derivation OCL" value.
5. Click the "Update ECO source code" button on the [Model View] tab.
6. Open the file QuickStartPackage.pas.
7. Locate a method named "FullNameDeriveAndSubscribe" in the Author class.

```
function Author.FullNameDeriveAndSubscribe(reevaluateSubscriber:
Eco.Subscription.ISubscriber;
resubscribeSubscriber: Eco.Subscription.ISubscriber): TObject;
begin
end;
```

The method should return an object (a string in this case) and provides the developer with two parameters, both of which are of the type ISubscriber. These parameters are used to subscribe to each element used whilst calculating the result value, this ensures that ECO cached value for the derived member is discarded if any of the elements used should change.

Parameter	Purpose
reevaluateSubscriber	This subscriber should be used for attributes on a class such as FirstName, LastName, and attributes of associated instances such as self.PurchaseOrderLines[!].LineValue.
resubscribeSubscriber	This subscriber should be used for subscribing to lists of objects, such as self.PurchaseOrderLines.

A subscription is placed on a member of a class by retrieving an IProperty reference for the member. This is done using the following steps

1. Get an IObject reference from the AsIObject method.
2. Use the "Properties" property to get an IPropertyCollection.
3. Find the relevant IProperty by using the GetByLoopbackIndex method.

4. Use the `SubscribeToValue` method to place the subscription.

Add the following code to the method, this code will first place subscriptions and then return the calculated result. Note that the `resubscribeSubscriber` parameter is not used here because there is only a single object involved in the calculations. Within each class an embedded class named "Eco\_LoopbackIndices" is defined which holds a constant for each of the class's members.

```
function Author.FullNameDeriveAndSubscribe(reevaluateSubscriber:
Eco.Subscription.ISubscriber;
resubscribeSubscriber: Eco.Subscription.ISubscriber): TObject;
begin
    //Place subscriptions
    AsIOBJECT.Properties.GetByLoopbackIndex(Eco_LoopbackIndices.Salutation).SubscribeToValue
(reevaluateSubscriber);
    AsIOBJECT.Properties.GetByLoopbackIndex(Eco_LoopbackIndices.FirstName).SubscribeToValue
(reevaluateSubscriber);
    AsIOBJECT.Properties.GetByLoopbackIndex(Eco_LoopbackIndices.LastName).SubscribeToValue(
reevaluateSubscriber);

    //Calculate the result
    Result := Salutation + ' ' + FirstName + ' ' + LastName;
end;
```

## 9.3 Creating derived settable members

A derived settable member is the same as a code derived member except it allows the user to modify the calculated value. When an attempt is made to modify the value a specific method will be executed, in which the developer is responsible for "reversing" the calculation.

Here are some more scenarios in which you may wish to use this feature:

1. Allowing a user to enter their current age, and then using this to calculate which year they were born.
2. Allowing a user to adjust the gross amount on an invoice resulting in the net and tax amounts being adjusted automatically.
3. Allowing `DepartureTime`, `ArrivalTime`, and `TripDuration` to be modified and for all three to be kept in sync.

To allow the user to modify a derived attribute follow these steps:

1. Select the class's "FullName" attribute in the Object Inspector.
2. Set its "Derived settable" property to "True".
3. Click the "Update ECO source code" button at the top of the [Model View] tab.
4. Open the source code for the Author class in the editor.
5. Search for the method "set\_FullName" and implement it using the following code.

### [Delphi]

```
procedure Author.set_FullName(Value: string);
const
    Separators: array[1..1] of char = ('/');
var
    NameParts: array of string;
begin
```

```
NameParts := Value.Split(Separators);
if Length(NameParts) = 3 then
begin
  if NameParts[0] <> '' then
    Salutation := NameParts[0];
  if NameParts[1] <> '' then
    FirstName := NameParts[1];
  if NameParts[2] <> '' then
    LastName := NameParts[2];
end;
end;
```

The user may now enter an author's name by modifying the FullName in the format "Salutation/FirstName/LastName" instead of having to set focus to three separate controls and enter each part individually. If there are not exactly three parts to the user entry (two forward slashes) then the input is ignored. If any of the values entered are empty then the current value will be left. For example, the value "//Morris" would change only the last name.

# 10 Default string representation

In the previous example "Creating an OCL derived member", in the OCL expression you will see the text "self.Author.asString". Each modeled business class has a "Default String Representation" property which is an OCL expression that should be evaluated for when the OCL identifier "asString" is evaluated. This would mean that the OCL previously used for Article Summary

```
self.Title + ' by ' + self.Author.asString
```

would result in a string looking something like this

```
MyFirstArticle by CapableObjects.QuickStart.BusinessModel.Author
```

This is because by default the Default String Representation will return the name of the class. Instead the model will be changed to show the author's full name using our code-derived "FullName" derived attribute.

1. Open the design surface for the class diagram and select the Author class, alternatively you may select the Author class in the [Model View] tab.
2. In the Object Inspector set the "Default String Representation" to "self.FullName".
3. Select the Article class and set the "Default String Representation" to "self.Summary".
4. Click the "Update ECO Source Code" button at the top of the [Model View] tab.
5. Compile the project and save all changes.

# 11 Summary

This article has described how to use the integrated modeler to create classes, UML attributes, and associations. This model may now not only be used in various applications but may also be combined with other packages or even extended upon by other packages. These are all topics that will be covered later in this series.

# Index

## A

- Adding associations between classes 8
- Adding attributes to the class 5
- Adding object validation 11

## C

- Creating a class diagram 4
- Creating an ECO model 3
- Creating code derived members 17
- Creating derived settable members 18
- Creating OCL derived members 16

## D

- Default string representation 20
- Derived members 16

## G

- Generating source code 13

## I

- Inspecting the source code 13

## O

- Overview 1

## P

- Prerequisites and goals 2

## S

- Summary 21