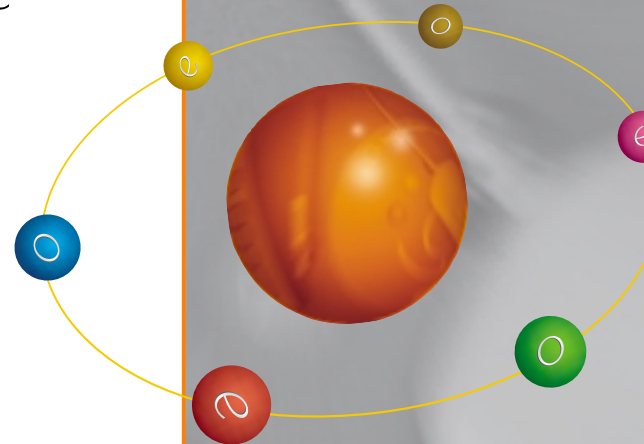


Objecteering/UML

Objecteering/VB Developer User Guide

Version 5.2.2



Objecteering

Software

www.objecteering.com

Taking object development one step further

Information in this document is subject to change without notice and does not represent a commitment on the part of Objecteering Software. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the purchaser's personal use, without the express written consent of Objecteering Software.

© 2003 Objecteering Software

Objecteering/UML version 5.2.2 - CODOBJ 001/001

Objecteering/UML is a registered trademark of Objecteering Software.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

UML and OMG are registered trademarks of the Object Management Group. Rational ClearCase is a registered trademark of Rational Software. CM Synergy is a registered trademark of Telelogic. PVCS Version Manager is a registered trademark of Merant. Visual SourceSafe is a registered trademark of Microsoft. All other company or product names are trademarks or registered trademarks of their respective owners.

Contents

Chapter 1: Introduction to the Objectteering/Visual Basic module	
Overview of the Objectteering/Visual Basic module	1-3
Structure of the Objectteering/Visual Basic user guide	1-4
Essential Objectteering/UML concepts	1-5
Essential Visual Basic concepts	1-7
Correspondence between Objectteering/UML and Visual Basic	1-8
Chapter 2: Using the Objectteering/Visual Basic module	
Working with the Objectteering/Visual Basic module	2-3
The properties editor for the Objectteering/Visual Basic module	2-6
Chapter 3: Objectteering/Visual Basic First Steps	
Getting started	3-3
Generating code	3-5
Visualizing generated code	3-8
Editing generated code	3-11
Running and compiling generated Visual Basic code	3-13
Chapter 4: Objectteering/UML elements and Visual Basic equivalence	
Classes	4-3
Attributes	4-11
Associations and aggregations	4-17
Collections.....	4-22
Operations.....	4-25
Operation parameters	4-32
Enumerations	4-37
MIDL/Visual Basic/Objectteering equivalence	4-40
Chapter 5: Objectteering/Visual Basic : Generalization, interface and types	
Generalization, interfaces and polymorphism	5-3
Chapter 6: The Objectteering/Visual Basic interface	
The Objectteering/Visual Basic interface	6-3
Chapter 7: Parameterizing the Objectteering/Visual Basic module	
Overview of module parameterization	7-3
Parameter sets.....	7-4
Index	

Chapter 1: Introduction to the
Objecteering/Visual Basic
module

Overview of the Objectteering/Visual Basic module

Overview

Welcome to the *Objectteering/VB Developer* user guide!

The *Objectteering/Visual Basic* module is used to generate Visual Basic code based on UML models and annotations.

The *Objectteering/Visual Basic* module is designed for Visual Basic designers and programmers who want to automatically translate their UML design into Visual Basic code. A certain level of knowledge in Visual Basic, UML and Objectteering/UML is presumed.

Functions

Using the *Objectteering/Visual Basic* module, it is possible to:

- ◆ generate Visual Basic class modules from UML classes
 - ◆ generate the content (methods, properties, ...) of these modules
 - ◆ generate enumerations, types, and so on
-

Structure of the Objectteering/Visual Basic user guide

The *Objectteering/Visual Basic* user guide is structured as follows:

- ◆ *Chapter 1: An introduction to the Objectteering/Visual Basic module*
 - ◆ *Chapter 2: Working with the Objectteering/Visual Basic module*
 - ◆ *Chapter 3: Objectteering/Visual Basic First Steps*
 - ◆ *Chapter 4: A description of Objectteering/UML elements and their equivalents in Visual Basic*
 - ◆ *Chapter 5: A chapter on generalization, interfaces and polymorphism*
 - ◆ *Chapter 6: A chapter describing the Objectteering/Visual Basic interface*
 - ◆ *Chapter 7: Parameterizing the Objectteering/Visual Basic module*
-

Essential Objectteering/UML concepts

Overview

In this section, the user can refresh his memory regarding the Objectteering/UML concepts of tagged values, notes, stereotypes and generation templates.

Tagged values

Tagged values are used to annotate elements, in order to add a particular semantic meaning to the element in question. For example, the addition of the *{persistent}* tagged value to a class makes this class persistent.

The availability of tagged values depends on which Objectteering/UML modules have been chosen, as well as on the nature of the element edited.

It is possible to add from 0 to n parameters to a tagged value.

Conceptually speaking, tagged values are used to annotate predefined UML types. For example, the *{persistent}* tagged value is a boolean annotation, which can be added to a class. It is not found in the code generated for this class, but will, for example, drive an SQL code generator.

Notes

Notes are particular cases of tagged values, to which a single text type parameter is associated. Objectteering/UML uses notes to store things like:

- ◆ documentation on the annotated element
- ◆ operation code

Stereotypes

Stereotypes are used to specialize a type of element, so as to limit or specialize its definition.

For example, an operation can be stereotyped «create». If this is the case, the operation will then have the properties, constraints and behavior of a constructor.

Conceptually speaking, stereotypes are used to define new UML predefined element sub-types. For example, the «create» stereotype defines a new type of operation.

Generation templates

The *Objectteering/Visual Basic* module has been realized using the so-called "*generation template*" technique. This technique is used to describe the generator in a simple hierarchical way, in order that it be:

- ◆ Easily understandable
 - ◆ Easily parameterizable
 - ◆ Easily maintained
-

Essential Visual Basic concepts

Overview

In this section, the user can refresh his memory on certain basic Visual Basic concepts.

The project in Visual Basic

A Visual Basic project is defined as a set made up of forms, modules and at least one precompiled resource (*.res) file. Classes and modules define the application's objects and their functions. The project (project file (*.vbp)) also includes a list of OCX components, which are to be included for the generation (link) of exe code.

Unlike a simple model, a class module has a pseudo-constructor (*Initialize*) and a pseudo-destructor (*Terminate*). The same is true for Visual Basic Forms, which also have a pseudo-constructor (Form_Load) and a pseudo-destructor (Form_Unload). None of these methods take parameters.

Visual Basic components

Visual Basic components are installable libraries, which are used to link functions (OLE or OLE servers) to a project. In Visual Basic, almost everything is OLE. Components can be presented in very different forms: OLB, OCX, ActiveX (internet OCX), DLL, PKG, and so on. Each component can be considered as an object. A file of components (OCX or other) can contain several objects.

Basic checks (those defined in the tool box, generally presented on the left of the screen) are also contained in a (one of the) OLE server(s) (VBxxx.dll).

Correspondence between Objectteering/UML and Visual Basic

Overview

A Visual Basic component more or less corresponds to a package in UML.

Since a component can be broken down into several modules, classes, forms and other integrated components, the range is managed on a component (a Visual Basic project). Unlike Objectteering/UML (UML), a Visual Basic component cannot be abstract and must always, therefore, be instantiated (here, this concerns an OLE server).

Figure 1-1 shows "Visual Basic-style" encapsulation by ActiveX type OLE servers.

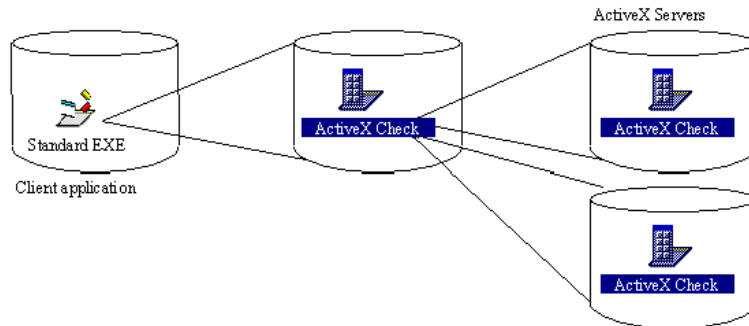


Figure 1-1 " Visual Basic-style" encapsulation using ActiveX type OLE servers

A Visual Basic component is very similar to a DLL and can be loaded as and when (late binding) or directly integrated into the client application (direct instantiation of an ActiveX component).

Visual Basic is a functional language like C (furthermore, it can very easily be interfaced with C).

In Visual Basic, containers called "*Collections*" exist, and an iterator is generated on each collection by a loop:

```
« For Each <obj> In <collectionInst> .. Next ».
```

Chapter 2: Using the
Objectteering/Visual Basic
module

Working with the Objecteering/Visual Basic module

Introduction


Before the *Objecteering/Visual Basic* module can be used, the following steps must be carried out:

- 1 - Create a working UML modeling project.
- 2 - Select the module.

Creating a working UML modeling project

For information on how to create a UML modeling project, please refer to the "*Creating or opening a UML modeling project*" section in chapter 3 of the *Objecteering/UML Modeler* user guide.

Selecting the VBModule module for the new UML modeling project

Launch the *Objecteering/UML Modeler* editor on your newly-created UML modeling project. The  "UML modeling project modules" icon launches the window used to select the module (as shown in Figure 2-1).

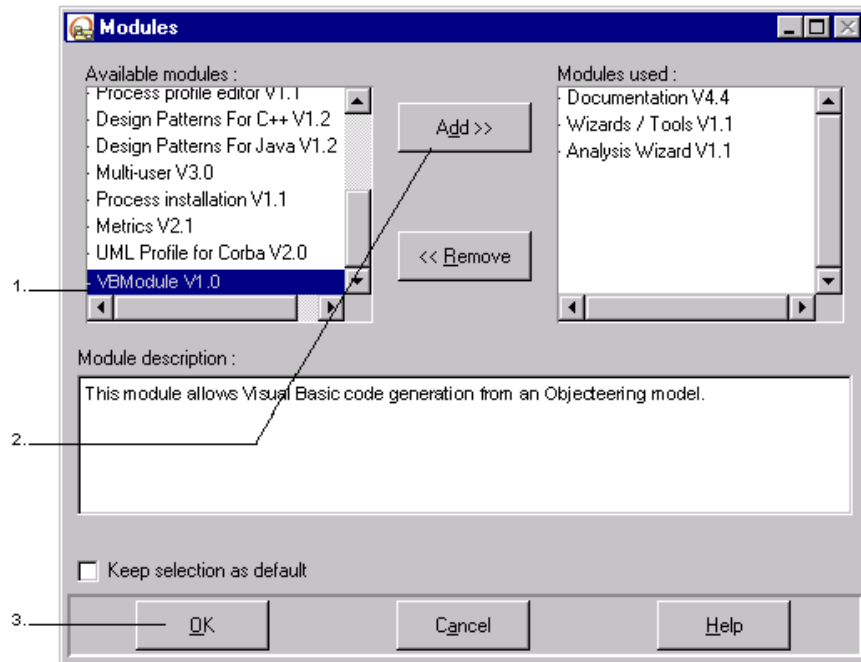


Figure 2-1. Selecting the Visual Basic module

Steps:

- 1 - Select the "VBModule" module from the available modules list on the left-hand side of the screen.
- 2 - Click on the "Add" button. The "VBModule" module then appears in the right-hand "Modules used" column.
- 3 - Click on "OK" to confirm. If the "Keep selection as default" box is checked, the "VBModule" module will automatically be available during future Objectteering/UML sessions.

For further information on this operation, please refer to the "*Selecting modules in the current UML modeling project*" section in chapter 3 of the *Objectteering/Introduction* user guide.

The properties editor for the Objectteering/Visual Basic module

The "VB" tab of the properties editor for a package

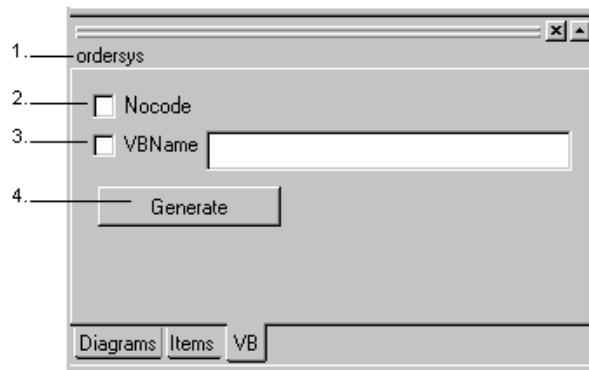


Figure 2-2. The "VB" tab of the properties editor on a package

Key:

- 1 - This indicates the name of the package selected in the explorer.
- 2 - This field is used to add the *{NoCode}* tagged value.
- 3 - This field is used to add the *{VBName}* tagged value.
- 4 - The "Generate" button is used to launch VB code generation on the selected package.

The "VB" tab of the properties editor for a class

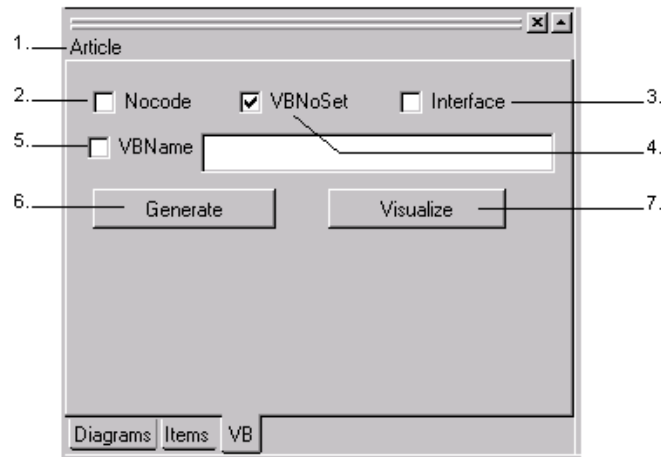


Figure 2-3. The "VB" tab of the properties editor on a class

Key:

- 1 - This indicates the name of the class selected in the explorer.
- 2 - This field is used to add the *{NoCode}* tagged value.
- 3 - This field is used to add the *{VBNoSet}* tagged value.
- 4 - This field is used to add the *<<interface>>* stereotype.
- 5 - This field is used to add the *{VBName}* tagged value.
- 6 - The "Generate" button is used to generate VB code for a class.
- 7 - The "Visualize" button is used to visualize the generated code.

The "VB" tab of the properties editor for an operation

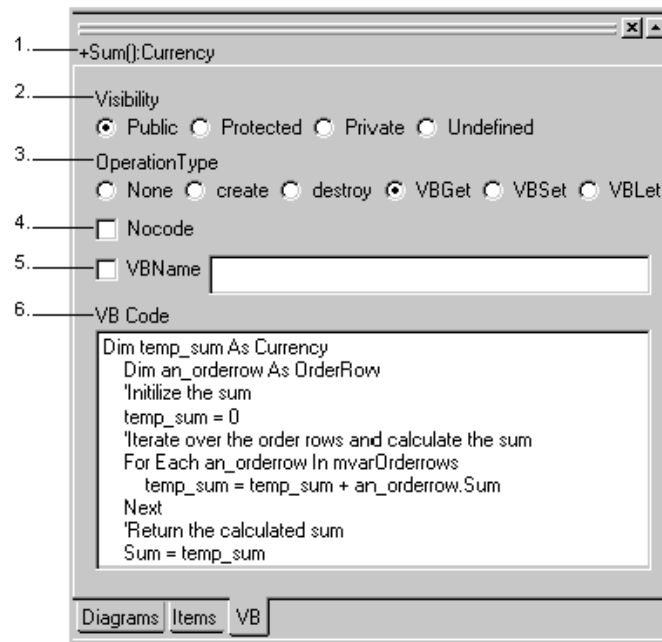


Figure 2-4. The "VB" tab of the properties editor on an operation

Key:

- 1 - This indicates the name of the operation selected in the explorer.
- 2 - The "Visibility" buttons are used to select the visibility of the operation.
- 3 - The "Operation type" buttons are used to select the type of the operation.
- 4 - This field is used to add the {NoCode} tagged value to the operation.
- 5 - This field is used to add the {VBName} tagged value to the operation.
- 6 - This field is used to enter or modify the operation's code.

The "VB" tab of the properties editor for an attribute

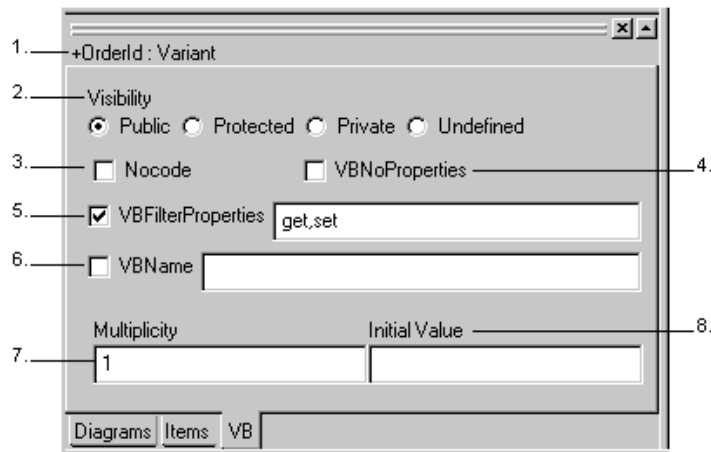


Figure 2-5. The "VB" tab of the properties editor on an attribute

Key:

- 1 - This indicates the name of the attribute selected in the explorer.
- 2 - The "Visibility" buttons are used to select the visibility of the attribute.
- 3 - This field is used to add the `{NoCode}` tagged value to the attribute.
- 4 - This field is used to add the `{VBNoProperties}` tagged value to the attribute.
- 5 - This field is used to add the `{VBFILTERProperties}` tagged value to the attribute, by checking the tickbox and adding the value of the filter (*get* and/or *set*) separated by a comma.
- 6 - This field is used to add the `{VBName}` tagged value to the attribute, by checking the tickbox and entering a name in the text field.
- 7 - This field is used to specify the value of the attribute's multiplicity
- 8 - This field is used to specify the initial value of the attribute.

The "VB" tab of the properties editor for an association

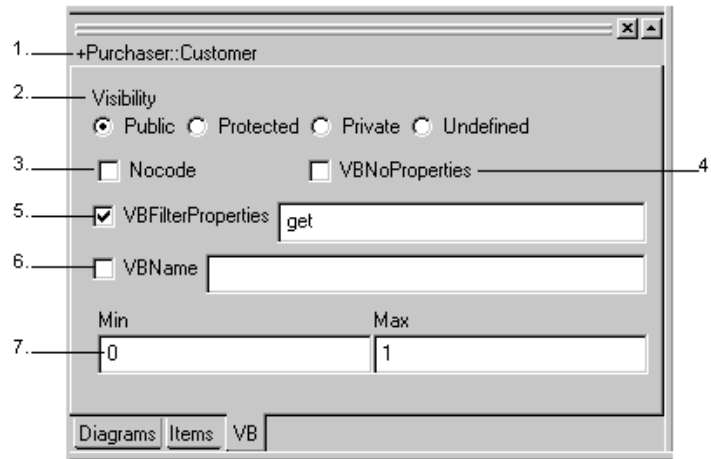


Figure 2-6. The "VB" tab of the properties editor on an association

Key:

- 1 - This indicates the name of the association selected in the explorer.
 - 2 - The "Visibility" buttons are used to select the visibility of the association.
 - 3 - This field is used to add the *{NoCode}* tagged value to the association.
 - 4 - This field is used to add the *{VBNoProperties}* tagged value to the association.
 - 5 - This field is used to add the *{VBFilterProperties}* tagged value to the association, by checking the tickbox and adding the value of the filter (*get* and/or *set*) separated by a comma.
 - 6 - This field is used to add the *{VBName}* tagged value to the association, by checking the tickbox and entering a name in the text field.
 - 7 - The "Min" and "Max" fields are used to specify the value of the association's multiplicity.
-

Chapter 3: Objecteering/Visual Basic First Steps

Getting started

Introduction

In these First Steps, we are going to use the "*Ordersys*" demonstration UML modeling project, in order to present the different features of the *Objectteering/Visual Basic* module step by step.

Initializing the First Steps UML modeling project

After having created a new UML modeling project named "Vbproject" and selected the *Objecteering/Visual Basic* module for this new UML modeling project, the next step is to import the Visual Basic First Steps UML modeling project (Figure 3-1).

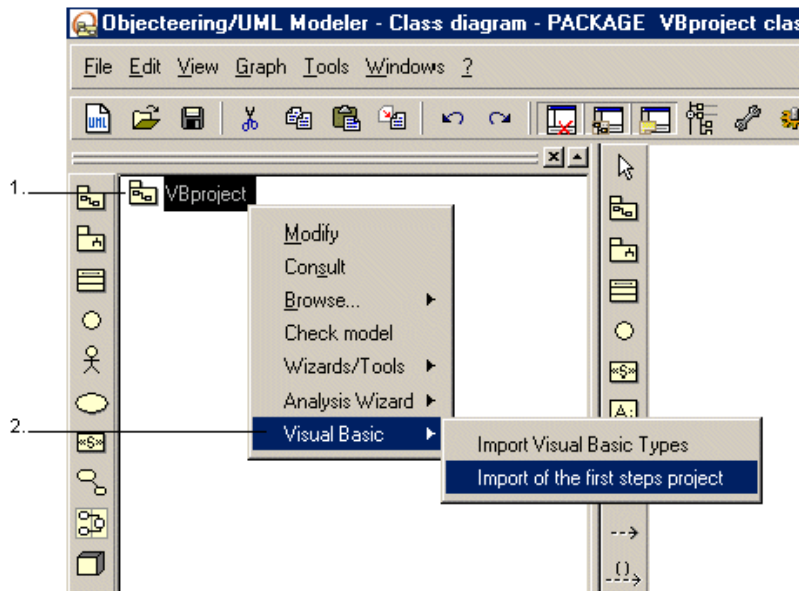


Figure 3-1. Importing the Visual Basic first steps project

Steps:

- 1 - Right-click on the UML model root (in this case, "Vbproject"), in order to display the context menu.
- 2 - Run the "Visual Basic/Import of the first steps project" command.

The *Objecteering/Visual Basic* first steps project is then imported into your newly created UML modeling project.

Generating code

Creating a generation work product

For the purposes of these first steps, this operation is not necessary, since all the generation work products are already present in the first steps project you have just imported.

Note: The creation of a generation work product is an essential step in the code generation procedure. For further information on the creation of generation work products, please refer to the "*The Objectteering/Visual Basic interface*" in chapter 6 of this user guide.

Generating code

We are now going to generate the code for the "ordersys" package (Figure 3-2).

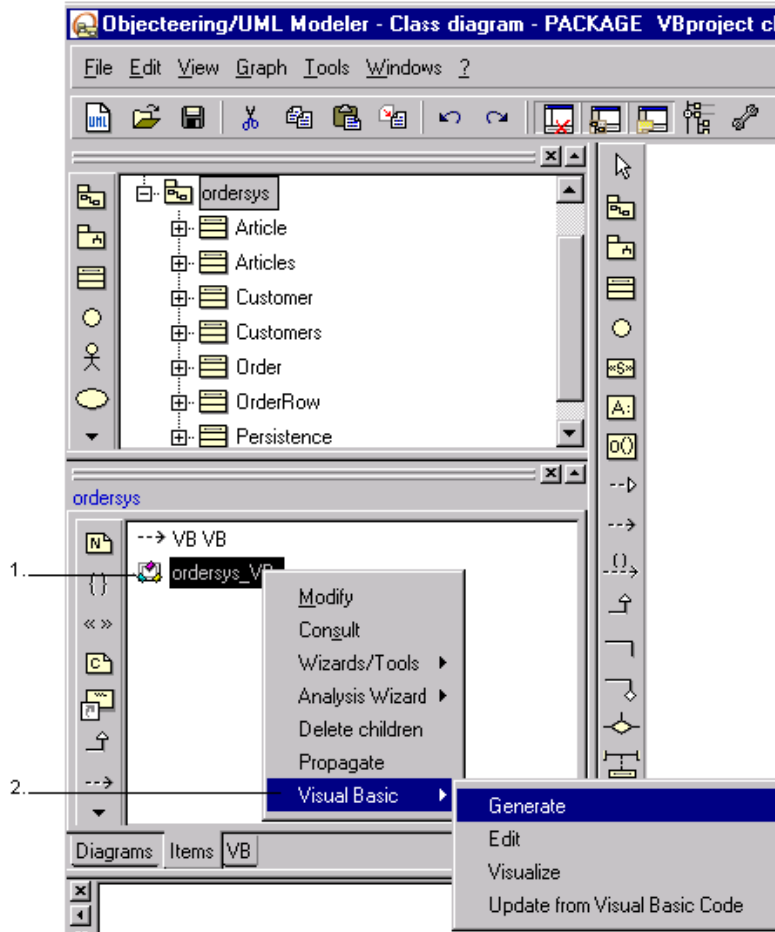


Figure 3-2. Generating Visual Basic code on the "ordersys_VB" generation work product

Steps:

- 1 - Right-click on the "*ordersys_VB*" generation work product in the "*Items*" tab of the properties editor to display the context menu.
- 2 - Run the "*Visual Basic/Generate*" command.

Visual Basic code is then generated in the generation directory specified when the generation work product is created. In this case, the generation work product was previously created, and the "*C:\Projects\vb\ordersys*" generation directory defined. All the code will be generated in this directory.

Visualizing generated code

To visualize the code generated during the previous step, simply carry out the steps indicated below (Figure 3-3).

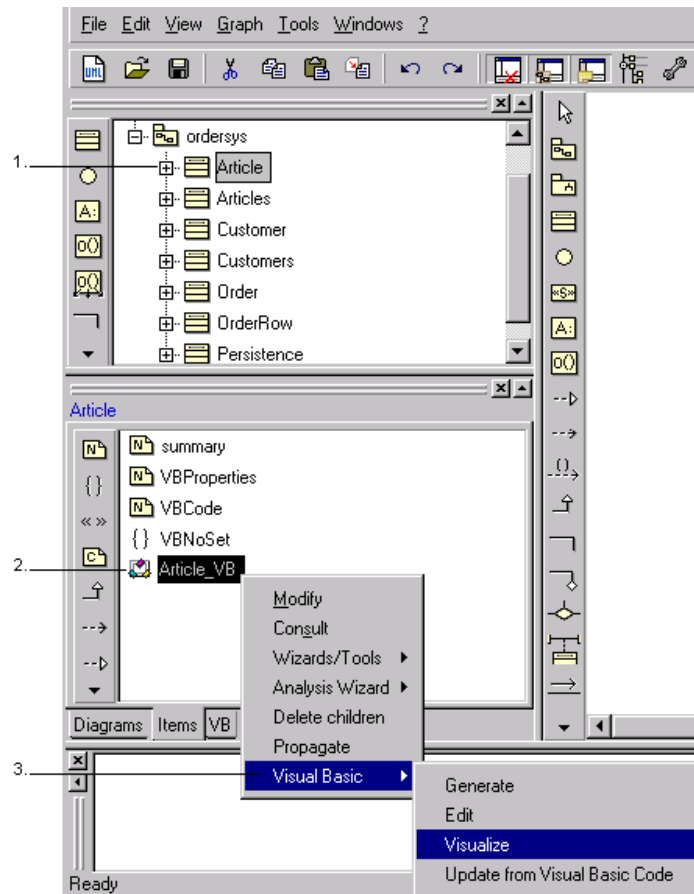


Figure 3-3. Visualizing the code generated on the "Article" class

Steps:

- 1 - Select the "Article" class in the explorer.
- 2 - Right-click on the "Article_VB" generation work product in the "Items" tab of the properties editor to display the context menu.
- 3 - Run the "Visual Basic/Visualize" command.

This command opens a window containing the VB code generated on the "Article" class (Figure 3-4).

```

C:\Projects\vb\ordersys\Article.cls

'START OF MODIFIABLE ZONE@OBJID@15627@270271284:
2488@T
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1 'True
  Persistable = 0 'NotPersistable
  DataBindingBehavior = 0 'vbNone
  DataSourceBehavior = 0 'vbNone
  MTSTransactionMode = 0 'NotAnMTSObject
END
Attribute VB_Name = "Article"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
Attribute VB_Ext_KEY = "SavedWithClassBuilder6" , "Yes"
Attribute VB_Ext_KEY = "Top_Level" , "Yes"
'END OF MODIFIABLE ZONE@OBJID@15627@270271284:
2488@E
' Class Article : An article in stock, uniquely identified by an article
number.
'HARD CODED ZONE

'START OF MODIFIABLE ZONE@OBJID@15629@270271284:
2491@T
  
```

Figure 3-4. The Visual Basic code generated on the "Article" class

Chapter 3: Objectteering/Visual Basic First Steps

If you double-click on the blue text, a dialog box containing this text then opens, in which you can modify the code (Figure 3-5).

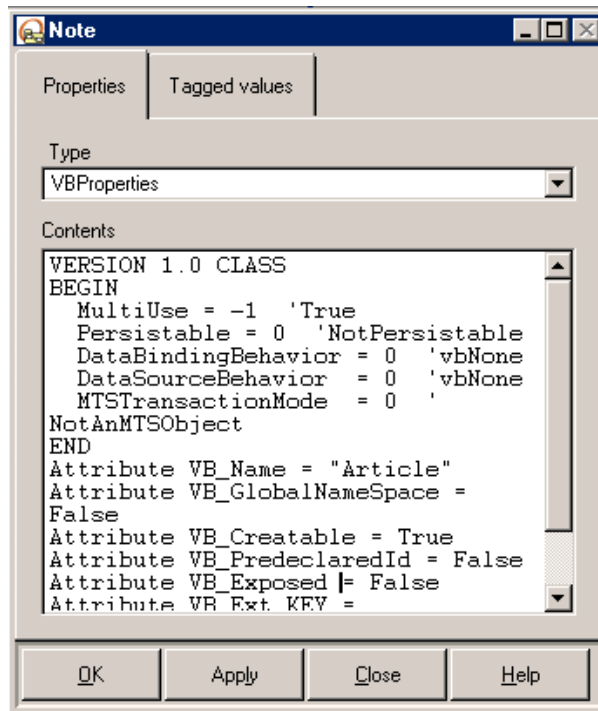


Figure 3-5. Editing generated code

Editing generated code

It is also possible to edit the code generated by following the steps shown in Figure 3-6.

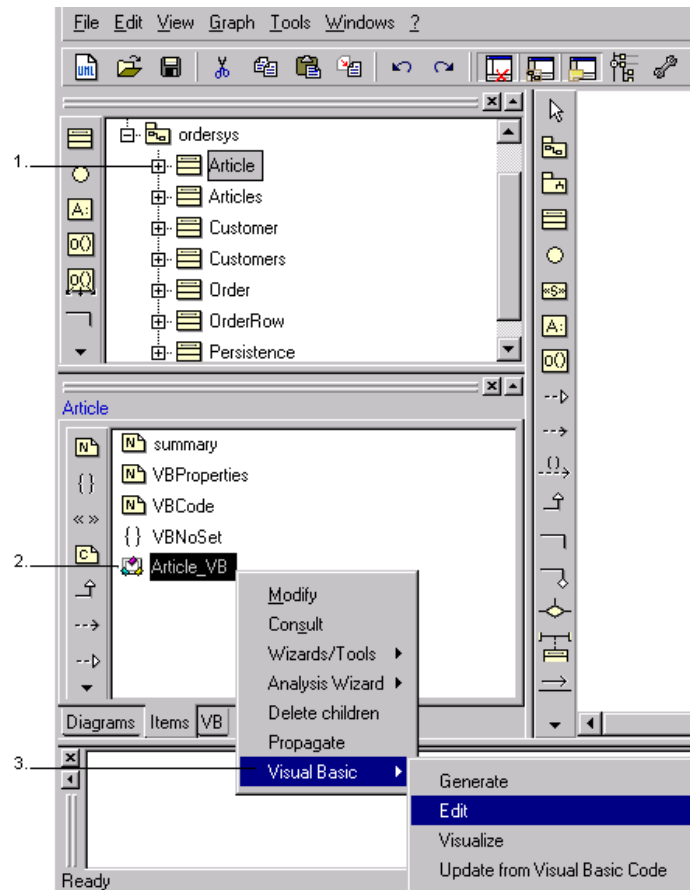


Figure 3-6. Editing generated code

Chapter 3: Objectteering/Visual Basic First Steps

Steps:

- 1 - Select the "*Article*" class in the explorer.
- 2 - Right-click on the "*Article_VB*" generation work product in the "*Items*" tab of the properties editor to open the context menu.
- 3 - Run the "*Visual Basic/Edit*" command.

This command opens Visual Basic (if the access path has been defined at module configuration level), in order to edit the generated "*Article.cls*" file.

Note: You can also update previously generated Visual Basic code, by right-clicking on the generation work product and running the "*Visual Basic/Update from Visual Basic code*" command.

Running and compiling generated Visual Basic code

Before running the code, you should first retrieve the different files and directories which are necessary to the execution of this First Steps project, which are found in the "C:\Program Files\Objectteering\modules\VBModule\1.0\Samples\ordersys" directory.

The seven files, "Db.bas", "dlg_orde.frm", "dlg_ordr.frm", "fish.ico", "ordersys.vbp", "ordersys.vbw" and "ordrsysm.bas", as well as the entire "database" directory found in this directory, should be copied into the "C:\Projects\vb\ordersys" generation directory.

The Visual Basic code must be run in Visual Basic itself.

To open the "ordersys" project, launch Visual Basic 6.0 and run the "*Fichier/Ouvrir un projet*" command. Select "ordersys.vbp" in the generation directory.

This command opens the "ordersys" project directly in Visual Basic.

Chapter 3: Objecteering/Visual Basic First Steps

Before running the program, you should select an option from the "Projet" menu, as shown in Figure 3-7.

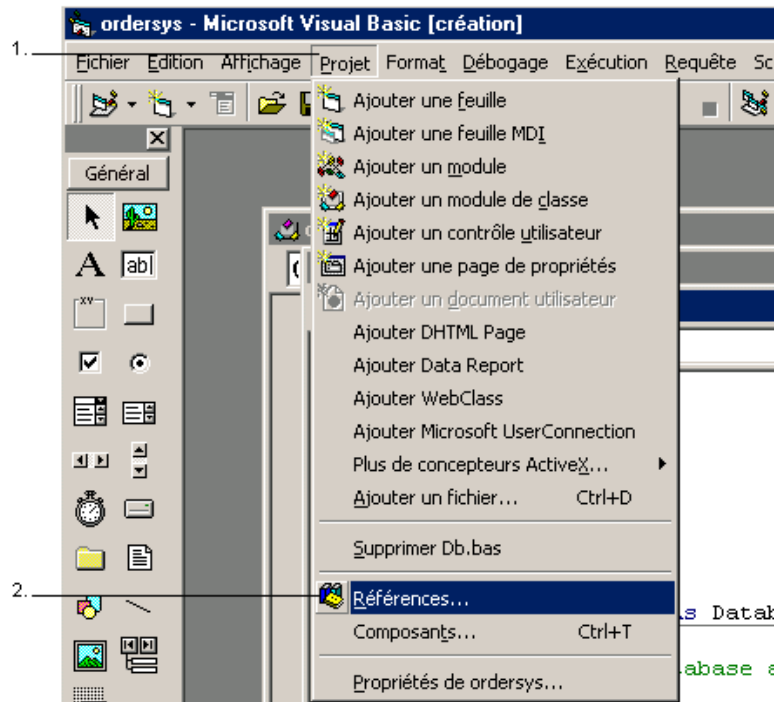


Figure 3-7. Running options

Steps:

- 1 - Click on the "Projet" menu in the Visual Basic menu bar.
- 2 - Select the "Références" option.

This command opens a new dialog box, used in the selection of several references (Figure 3-8).

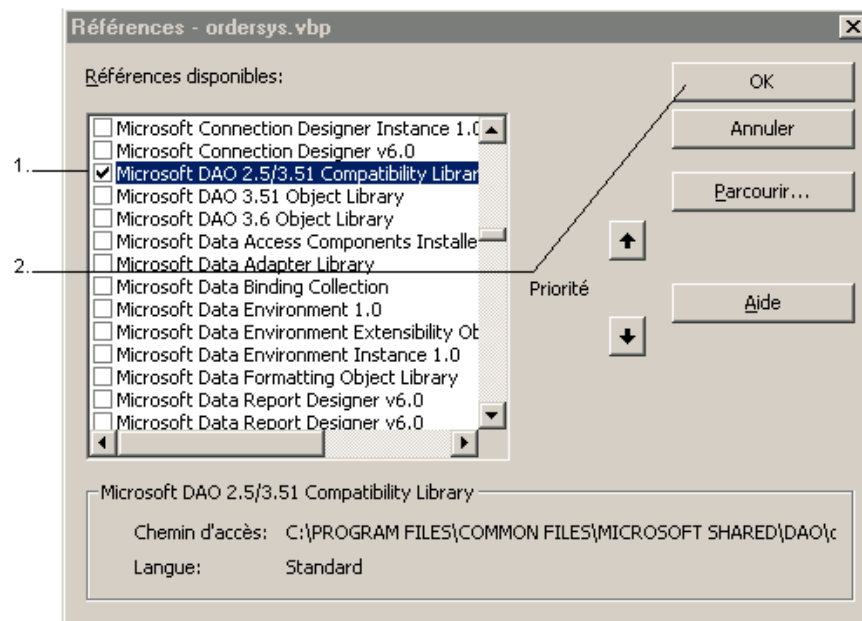


Figure 3-8. Choosing a reference

Steps:

- 1 - Select the "Microsoft DAO" reference.
- 2 - Click on the "OK" button to confirm.

Chapter 3: Objectteering/Visual Basic First Steps

Now, all that remains to be done is to run the program, by carrying out the steps illustrated in Figure 3-9.

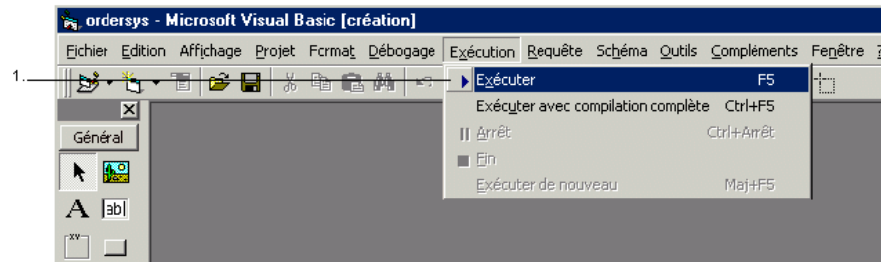


Figure 3-9. Running

Steps:

- 1 - Either press F5 or click on "Exécution/Exécuter".

Chapter 4: Objecteering/UML
elements and Visual Basic
equivalence

Classes

Introduction

A class represents a pattern for an object's creation and has an extended description in the model. A class represents its instances, and can have attributes and operations, as well as dependency links, associations and generalization links.

Objectteering/UML class dialog box

The dialog box used to modify information on a class is shown in Figure 4-1:

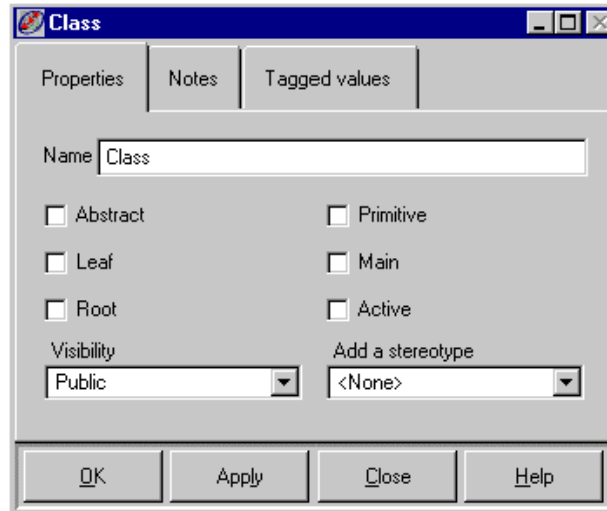


Figure 4-1. The "Class" dialog box in Objectteering/UML

Figure 4-2 shows the Visual Basic menu used to list those objects which may be added to a Visual Basic project:

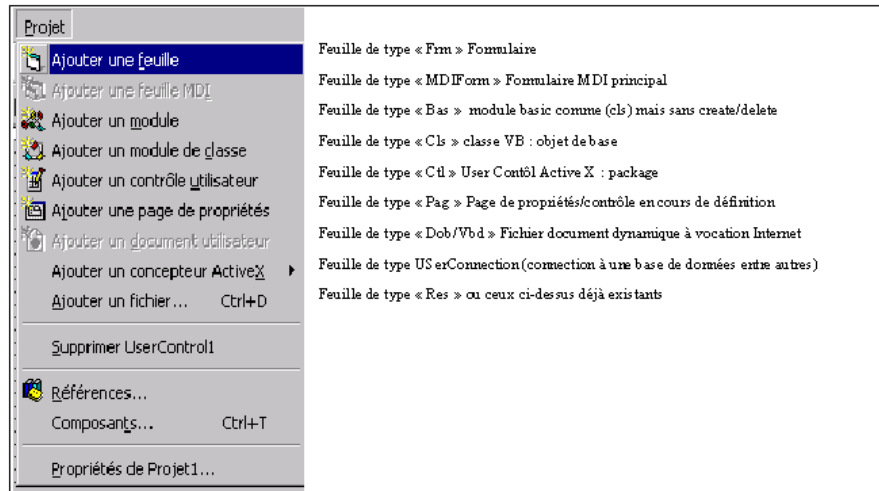


Figure 4-2. The Visual Basic menu used to add objects to a Visual Basic project

A class is generated as a Visual Basic class module (.cls file).

Generation templates

The general form of a generated class module file is as follows:

```
'VB properties declaration zone, for example
VERSION 1.0 CLASS
BEGIN
    MultiUse = -1 'True
    Persistable = 0 'NotPersistable
    DataBindingBehavior = 0 'vbNone
    DataSourceBehavior = 0 'vbNone
    MTSTransactionMode = 0 'NotAnMTSObject
END
Attribute VB_Name = "Article"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
Attribute VB_Ext_KEY = "SavedWithClassBuilder6" ,"Yes"
Attribute VB_Ext_KEY = "Top_Level" ,"Yes"
'END OF MODIFIABLE ZONE

' Class <class name> : <class documentation>, for example:
' Class Article : An article in stock, uniquely identified
by ' an article number.
...
'START OF MODIFIABLE ZONE@OBJID@33654@270271284:2241@T@24
' specific VB code
Option Base 0 ' for example
'END OF MODIFIABLE ZONE@OBJID@33654@270271284:2241@E@27
...
'-----
' Enumerations
'-----
'Enumerations declarations
...
'-----
' Types
'-----
'Types declarations
...
'-----
' ATTRIBUTES DECLARATION SECTION
'-----
'Attributes declarations
...
'-----
' ASSOCIATIONS DECLARATION SECTION
```

Chapter 4: Objectteering/UML elements and Visual Basic equivalence

```
'-----  
'Associations declarations  
...  
'-----  
' ATTRIBUTES ACCESSORS SECTION  
'-----  
'Attributes accessors (property get/let/set) declarations  
...  
'-----  
' ASSOCIATIONS ACCESSORS SECTION  
'-----  
'Associations accessors (property get/let/set) declarations  
...  
'-----  
' OPERATIONS SECTION  
'-----  
'Operations declarations  
...  
'-----  
' CONSTRUCTOR  
'-----  
'Constructor declaration  
...  
'-----  
' DESTRUCTOR  
'-----  
'Destructor declaration
```

Description of Objectteering/UML dialog box and equivalence in Visual Basic

The following table provides a description of the Objectteering/UML element, as well as its equivalent in Visual Basic.

Objectteering/UML	Description	Equivalent in Visual Basic
Name	Name of the class	The module's " <i>Name</i> " property
Primitive	Determines whether or not a class is primitive.	No Visual Basic equivalent
Abstract	An Abstract class is defined in a very general way, and has no direct instances.	No Visual Basic equivalent
Root	This is said of a class which is not derived from any other class	No direct Visual Basic equivalent Highest level Visual Basic Class Modules
Main	A main class is a class whose unique instance represents the application.	No direct Visual Basic equivalent In a Visual Basic project, the activation (and consequently the instantiations which result) is carried out via a module (Sub Main) or a Form unique to client applications. This can be envisaged when forms and modules are implemented.

Objectteering/UML	Description	Equivalent in Visual Basic
Visibility	Visibility can be either public or private. A public class is accessible from any package which uses the current package. A private class can only be accessed from the current package or a specializing package.	Visibility impacts "functions" and data within modules and not the actual modules themselves. By default, modules (objects) are public in the project/component.
Leaf	Defines a class which cannot have sub-classes (specializing classes).	No direct Visual Basic equivalent.
Active	Defines a class whose instances are active	An Event in Visual Basic is not an object, but rather a User CallBack. It is local to the module in which it is generated.

Notes and tagged values used to drive the generation of classes

The following tagged values are used when generating Visual Basic code:

The ... tagged value	is used to ...
{VBExtern} (on a class or a package)	tell the generator not to generate visual basic code on this unit.
{VBName} (on a class or a package)	replace the name of the unit by the value of the tagged value when generating VB code. For a package, this is the corresponding project name.
{VBNoSet}	tell the generator NOT to generate a collection for this class. (see collections chapter).

The following notes are used when generating Visual Basic code:

The ... note	is used to ...
VBProperties	<p>generate the first piece of VB code of a cls file. This part typically contains some code such as:</p> <pre> VERSION 1.0 CLASS BEGIN MultiUse = -1 'True Persistable = 0 'NotPersistable DataBindingBehavior = 0 'vbNone DataSourceBehavior = 0 'vbNone MTSTransactionMode = 0 'NotAnMTSObject END Attribute VB_Name = "Class1" Attribute VB_GlobalNameSpace = False Attribute VB_Creatable = True Attribute VB_PredeclaredId = False Attribute VB_Exposed = False Attribute VB_Ext_KEY = "SavedWithClassBuilder6" , "Yes" Attribute VB_Ext_KEY = "Top_Level" , "Yes" </pre> <p>This text is not intended to be modified by hand but through the Visual Basic windows.</p>
VBCode	<p>generate some specific Visual Basic code that either you don't want to be or you can't have modeled. You may modify this code in VB or Objectteering/UML.</p>
Summary	<p>generate the class documentation at the top of the cls file. It may be modified in Objectteering/UML only.</p>
Description	<p>generate documentation for the class if the summary note cannot be found.</p>

Attributes

Introduction

Attributes are generated in the form of Visual Basic properties, and are accompanied by their accessors (Get/Let/Set).

For example, for the Variant type ArticleId attribute, the following is obtained:

```
' -----
' ATTRIBUTES DECLARATION SECTION
' -----
' ArticleId : A unique article identifier. Also the article
' number used for identifying articles in stock.
Public mvarArticleId As Variant
...
' -----
' ATTRIBUTES ACCESSORS SECTION
' -----
Public Property Let ArticleId (ByVal vData As Variant)
On Error GoTo Article_ArticleId__exception
    mvarArticleId = vData
    Exit Property
Article_ArticleId__exception:
    Resume Article_ArticleId__end
Article_ArticleId__end:
    Exit Property

End Property

Public Property Get ArticleId () As Variant
On Error GoTo Article_ArticleId__exception
    If IsObject(mvarArticleId) Then
        Set ArticleId = mvarArticleId
    Else
        ArticleId = mvarArticleId
    End If
    Exit Property
Article_ArticleId__exception:
    Resume Article_ArticleId__end
Article_ArticleId__end:
    Exit Property

End Property
```

Objecteering/UML attribute dialog box

The dialog box used to modify information on an attribute is shown in Figure 4-3:

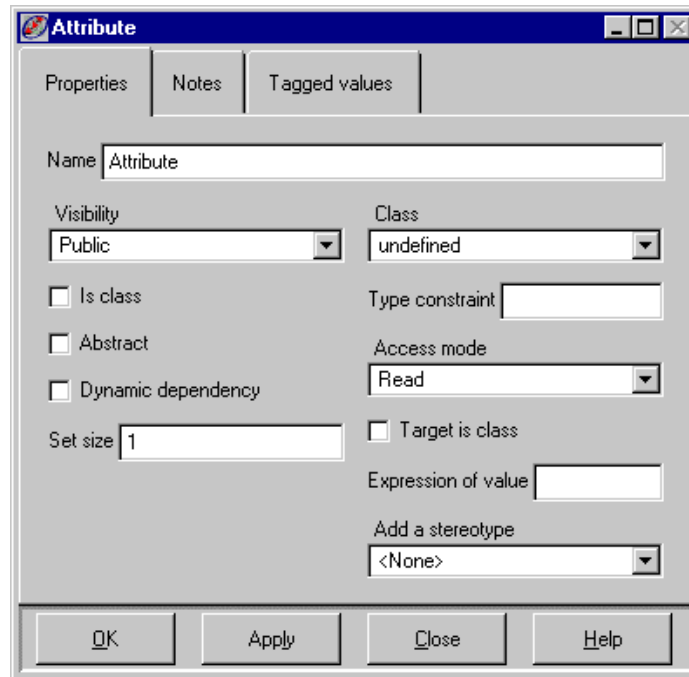


Figure 4-3. The "Attribute" dialog box in Objecteering/UML

Description Objectteering/UML dialog box and equivalence in Visual Basic

Objectteering/UML	Description	Equivalent in Visual Basic
Name	Name of the attribute	Name of the Visual Basic variable The variable name is made up of "mvar"+Name and the name of the Get/Let accessors will be Name (see the example above)
Visibility	Visibility of the attribute (none, public, protected or private).	None: the association is not generated. Private : Explicit declaration with Private+ Private accessors Public : Explicit declaration with Public+ Public accessors. Protected : Explicit declaration with Private + Friend accessors.
Class	Determines the attribute's type class. A help list selects "primitive" classes, as well as classes available through the current class.	Attribute type (please refer to the chapter on types)
Is Class	Specifies a class member, i.e. one shared by all instances of this class.	No equivalent. Please refer to notes below.
Dynamic Dependency	Determines whether or not the attribute is a dynamic dependency, i.e. whether its value is calculated dynamically through an expression. This also corresponds to "derived" attributes.	No equivalent. Will later be implemented by a property get function.

Objecteering/UML	Description	Equivalent in Visual Basic	
Expression of value	For a dynamic dependency, this field contains the expression of the dynamic calculation in the target language. Otherwise, this is the default value of the attribute.	No equivalent. Will subsequently be implemented in the Initialize method.	
Type Constraint	Provides an indication of the instantiation of the attribute's elementary class. For example, in the case of a string attribute, TypeConstraint determines the size of the string (*, 10, etc.).	For strings only (by default, almost unlimited if the size is not specified).	
Access Mode	Determines the type of authorized access to the attribute (none, read, write, read/write)	No equivalent. Can subsequently be used to find out which accessors (GET/LET) should be implemented on the attribute.	
Target is Class	Indicates that the type is a metaclass.	No Visual Basic equivalent.	
Set size	If its value is 1, the attribute is not a set, otherwise the size of the set is indicated (* for sets of unlimited size).	1	The attribute is not a set.
		>1 fixed size	Specified size : Dim A(T) As ...
		>1 variable size	Collection. Please refer to the chapter on collections

Notes and tagged values used to drive the generation of attributes

The following tagged values are used when generating Visual Basic code:

The ... tagged value	is used to ...
{VBLong} on an integer or a real type attribute	tell the generator the attribute is a Long or a Double (see types correspondances).
{VBTypeExpr}	replace the type of the unit by the value of the tagged value when generating VB code. For example, you could have an attribute with this tagged value valued at "StdFont" to generate a StdFont attribute.
{VBNoProperties}	tell the generator NOT to generate any properties accessors (get/set/let) on this attribute.
{VBFilterProperties } }	tell the generator that those properties that are listed in the tagged value parameters and only these ones do not have to be generated. For example, use {VBFilterProperties = Set} to generate only the set/let accessor.
{Nocode}	tell the generator not to generate code on this attribute.
{VBName}	replace the unit name by the value of the tagged value in the generated code.

The following notes are used when generating Visual Basic code:

The ... note	is used to ...
Description	generate the attribute documentation at the top of its declaration. It may be modified in Objecteering/UML only.

Notes

A function or a class member declared `Friend` will be visible throughout the current component, but not outside it. This does not exactly correspond to the description of *protected*, which means "visible in the class and its sub-classes".

A fixed length string declared as being *public* cannot be used in (object) class modules.

Where no type has been specified in Visual Basic, the `VARIANT` default type is used (please refer to the chapter on Objectteering/Visual Basic type equivalence).

A class attribute cannot be declared as being `STATIC`. Its range is specific to the module, unless specified by `PUBLIC` (a variable in a function can be static). In the future, this will result in the generation of a "class" pseudo-member through a global property of a module (.bas).

We recommend that you use `Collection` objects (containers) rather than dynamic tables for sets. Please refer to the "*Collections*" section in the current chapter of this user guide for further information.

Associations and aggregations

Introduction

Associations and aggregations are implemented by Visual Basic properties in the same way as attributes. They are accompanied by their accessors (Get/Set).

The fundamental difference between attributes and associations is that an attribute is typed by a base type, whilst an association is directed towards another class and is, therefore, "typed" by this class.

For example, for the navigable association from Order towards Customer, whose role is Purchaser, the following is obtained in Order:

```
' -----
' ASSOCIATIONS DECLARATION SECTION
' -----

' Purchaser : the customer that purchased the order
Public mvarPurchaser As Customer
...
' -----
' ASSOCIATIONS ACCESSORS SECTION
' -----
Public Property Set Purchaser(ByVal vData As Customer)
On Error GoTo Order_Purchaser__exception
    Set mvarPurchaser = vData
    Exit Property
Order_Purchaser__exception:
    Resume Order_Purchaser__end
Order_Purchaser__end:
    Exit Property
End Property

Public Property Get Purchaser() As Customer
On Error GoTo Order_Purchaser__exception
    Set Purchaser = mvarPurchaser
    Exit Property
Order_Purchaser__exception:
    Resume Order_Purchaser__end
Order_Purchaser__end:
    Exit Property
End Property
```

Objecteering/UML association dialog box

The dialog box used to modify information on an association is shown in Figure 4-4:

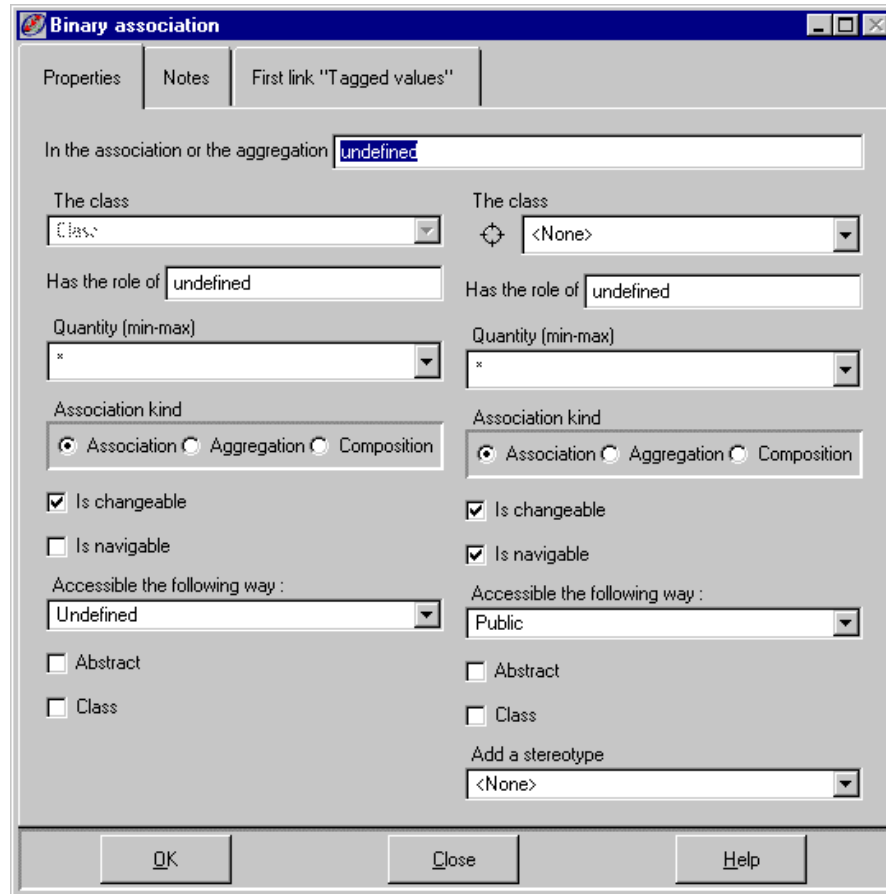


Figure 4-4. The "Binary association" dialog box in Objecteering/UML

Description of Objectteering/UML dialog box and equivalence in Visual Basic

Objectteering/UML	Description	Equivalent in Visual Basic	
In the association or aggregation	Name of the association.	No Visual Basic equivalent	
The class	Name of classes which are the extremities of the link.	Visual Basic property type	
Has the role of	The role played by the class in the link vis-a-vis the other class.	Visual Basic property name The name of the variable is made up of "mvar"+Role and the name of the accessors. Get/Set will be Role (see the example above)	
Quantity (min-max)	Interval cardinality in "min*max" form, where "min" designates the minimum number of instances of the other class, and "max" designates the maximum number of instances of the other class. The "*" symbol designates an unlimited number.	0-1 or 1-1	Generation of a Class type property
		0-N or 1-N (N fixed)	Generation of a DIM type property (N) As Class
		0-* or 1-*	Generation of a collection

Objectteering/UML	Description	Equivalent in Visual Basic
Association kind	Determines whether or not the representatives of the class concerned are : Composition : strong composition Aggregation : shared composition. Association : standard association	In all three cases, a Visual Basic property is generated. Subsequently, it is possible, for aggregates, to create and destroy corresponding objects in Class_Initialize and Class_Terminate.
Is Changeable	The instance at the end of this association can be modified.	No Visual Basic equivalent.
Is Navigable	The instance at the end of this link can be accessed from this association's opposite class	The Visual Basic property which corresponds to the association will only be generated if it is navigable.
Accessible the following way	Visibility of the member (none, public, protected or private).	None : the attribute is not generated. Private : Explicit declaration with Private+ Private accessors Public : Explicit declaration with Public + Public accessors. Protected : Explicit declaration with Private + Friend accessors.
Abstract	Determines whether or not the role is abstract.	No Visual Basic equivalent.
Class	Determines whether or not the role is class.	Please refer to the notes on attributes

For relationships of 0-* or 1-* type, it is preferable to use a Collection object (Container with integrated iterator) rather than a dynamic table, in order to preserve object encapsulation.

Notes and tagged values used to drive the generation of associations

The following tagged values are used when generating Visual Basic code:

The ... tagged value	Is used to ...
{VBTypeExpr}	replace the type of the unit by the value of the tagged value when generating VB code. For example, you could have an association with this tagged value valued at "StdPicture" to generate a StdPicture association.
{VBNoProperties}	tell the generator NOT to generate any properties accessors (get/set) on this association.
{VBFilterProperties}	tell the generator that only those properties that are listed in the tagged value parameters have to be generated. For example, use {VBFilterProperties = Set} to generate only the set accessor.
{Nocode}	tell the generator not to generate code on this association.
{VBName}	replace the unit name by the value of the tagged value in the generated code.

The following notes are used when generating Visual Basic code:

The ... note	is used to ...
Description	generate the association documentation at the top of its declaration. It may be modified in Objectteering/UML only.

Collections

Introduction

For every XXX class, an XXXs collection is implemented.

This collection is generated in a separate file named XXXs.cls for the XXX class.

For example, for the navigable association from Order towards the OrderRow class (cardinality 0-*), whose role is Orderrows, an OrderRows class is generated and the following is obtained in Order :

```
'-----
' ASSOCIATIONS DECLARATION SECTION
'-----
' Orderrows : rows of the order
Private mvarOrderrows As Orderrows
...
'-----
' ASSOCIATIONS ACCESSORS SECTION
'-----
Private Property Set Orderrows(ByVal vData As Orderrows)
On Error GoTo Order_Orderrows__exception
    Set mvarOrderrows = vData
    Exit Property
Order_Orderrows__exception:
    Resume Order_Orderrows__end
Order_Orderrows__end:
    Exit Property
End Property

Private Property Get Orderrows() As Orderrows
On Error GoTo Order_Orderrows__exception
    Set Orderrows = mvarOrderrows
    Exit Property
Order_Orderrows__exception:
    Resume Order_Orderrows__end
Order_Orderrows__end:
    Exit Property
End Property
```

Class module code

This class module typically contains the following code for the Orders collection of the Order class:

```
'COLLECTION FOR Class Order
Private mCol As Collection
Public Sub Add(vItem As Order, Optional sKey As String)
    If Len(sKey) = 0 Then
        mCol.Add vItem
    Else
        mCol.Add vItem, sKey
    End If
End Sub

Public Property Get Item(vntIndexKey As Variant) As Order
    Set Item = mCol(vntIndexKey)
End Property

Public Property Get Count() As Integer
    Count = mCol.Count
End Property

Public Sub Remove(vntIndexKey As Variant)
    mCol.Remove vntIndexKey
End Sub

Public Property Get NewEnum() As IUnknown
    Set NewEnum = mCol.[_NewEnum]
End Property

Private Sub Class_Initialize()
    Set mCol = New Collection
End Sub

Private Sub Class_Terminate()
    Set mCol = Nothing
End Sub
```


Notes

If you do not wish this class module to be generated, you may add the {VBNoSet} tagged value to the class. In this case, you must deal yourself with the class collection code, either by writing your own collection class (named XXXs) or by using an existing one and by using the {VBTypeExpr = <collection class name>} on your n-ary association.

The "Add", "Item" and "Remove" functions allow you to add an item to the collection, get an item from the collection or remove an item from the collection.

The "Count" function returns the number of items in the collection.

The "NewEnum" function helps in enumerating the collection in a "for each" loop. You MUST reference the OLE automation lib in your VB project in order to include "IUnknown".

Operations

Introduction

Class operations are implemented by functions in Visual Basic class modules.

Objectteering/UML operation dialog box

The dialog box used to modify information on an operation is shown in Figure 4-5:

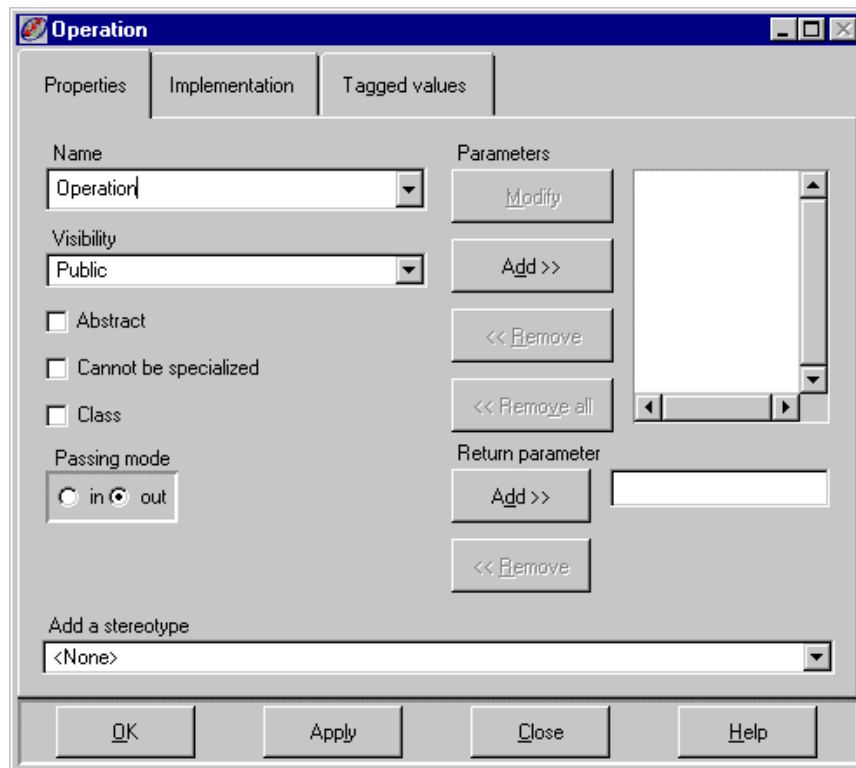


Figure 4-5. The "Operation" dialog box in Objectteering/UML

Description of Objectteering/UML dialog box and equivalence in Visual Basic

Objectteering /UML	Description	Equivalent in Visual Basic	
Name	Name of the operation	Name of the operation (please refer to the notes below)	
Visibility	Visibility of the member (none, public, protected or private).	Public	Public operation
		Protected	Friend : Operation which is public in the component but not outside.
		Private	Private
Abstract	Determines abstract operations, i.e. operations not implemented at this level.	No equivalent in Visual Basic.	
Cannot be specialized	Determines an operation which cannot be redefined in a subclass.	All Visual basic operations are non-derivable.	
Class	Defines a "class" operation, i.e. shared by all its instances.	Should use a global function in a separate Bas module (as for class attributes).	
Passing Mode	The operation's passing mode (in or inout). Determines whether the object is modified (inout) or not (in) by the operation.	By default, <i>inout</i> . (no <i>in</i> alone in Visual Basic).	

Notes

By default, Visual Basic code contains modular functions. These functions may be separated from one another using syntax, which depends on the name of the "object" dealt with.

For example, Class = File ; Operation = Destroy could give the following in Visual Basic:

```
Sub File_Destroy()  
    ' proceeding  
End Sub
```

Syntax : <Object> _ <method> (<List params>)

This (Sub) != function (Function) procedure (like in Pascal) is run as follows:

```
...  
File_Destroy  
...
```

Code generation

The Visual Basic code for operations is stored in Objectteering/UML in the form of "Visual Basic" notes named VBCode.

For example, in the "Customer" class, the "Fetch" method code is generated as follows:

```
' Fetch : Fetches a customer, from the database, given a
customer id.
Public Sub Fetch (ByVal Id As Integer)
On Error GoTo Customer_Fetch__exception

'START OF MODIFIABLE ZONE@OBJID@36050@270271284:2509@T@152
  'Set the Article Id query parameter and get the customer
  mvarStorage.theQueryDef.Parameters!Id = Id
  mvarStorage.Execute
  If mvarStorage.RecordsExists Then
    Customer
    CustomerId:=mvarStorage.theRecordSet!CustomerId,
    Name:=mvarStorage.theRecordSet!Name,
    Address:=mvarStorage.theRecordSet!Address
  Else
    Clear
  End If
'END OF MODIFIABLE ZONE@OBJID@36050@270271284:2509@E@161
Exit Sub
Customer_Fetch__exception:
  Resume Customer_Fetch__end
Customer_Fetch__end:
  Exit Sub
End Sub
```

The code contained in the "VBCode" note is generated between "START OF ZONE..." and "END OF ZONE ..." markers. It can then be modified directly in Visual Basic. The modified code can be reintroduced into the "VBCode" note in Objectteering/UML via an update command. (Please refer to chapter 6, "The Objectteering/Visual Basic interface" of this user guide for further information.)

The automatic generation of certain method parts as debug code has been hard coded. Furthermore, whether or not this code is generated depends on certain module parameters. The error code itself should be improved, in order to really handle exceptions, instead of just resuming execution.

Stereotypes, notes and tagged values used to drive the generation of operations

The following stereotypes are used when generating Visual Basic code:

The stereotype ...	is used to ...
Create	represent a class constructor and is implemented by the Class_Initialize method, whatever its name. This operation takes no parameters. Furthermore, attribute and association initialization code will automatically be generated in the Class_Initialize method.
Destroy	represent a class destructor and is implemented by the Class_Terminate method, whatever its name. Furthermore, attribute and association destruction code will be generated in the Class_Terminate method.
VBGet	generate a Property Get method instead of a function.
VBSet	generate a Property Set method instead of a procedure (Sub).
VBLet	generate a Property Let method instead of a procedure (Sub).

The following tagged values are used when generating Visual Basic code:

The ... tagged value	Is used to ...
{Nocode}	tell the generator not to generate code on this association.
{VBName}	replace the unit name by the value of the tagged value in the generated code.

The following notes are used when generating Visual Basic code:

The note ...	is used to ...
Summary	generate the operation documentation at the top of its declaration. It may be modified in Objecteering/UML only.
Description	generate the operation documentation if no summary note can be found.
VBCode	generate the Visual Basic code. This note contains the real application code of the operation. It may be modified either in Objecteering/UML or in the Visual Basic IDE.

Operation parameters

Introduction

UML operation parameters are transformed into their Visual Basic equivalent.

For example, for a private operation, "*M*", with an integer type in parameter, "*PI*", and a char type out parameter, "*PO*", which returns a boolean, the following is obtained:

```
Public Function M(ByVal PI As Integer, PO As Byte) As  
Boolean  
...  
End Function
```

Objecteering/UML parameter and return parameter dialog boxes

Figures 4-6 and 4-7 show the Objecteering/UML dialog boxes used to enter or modify parameters and return parameters :

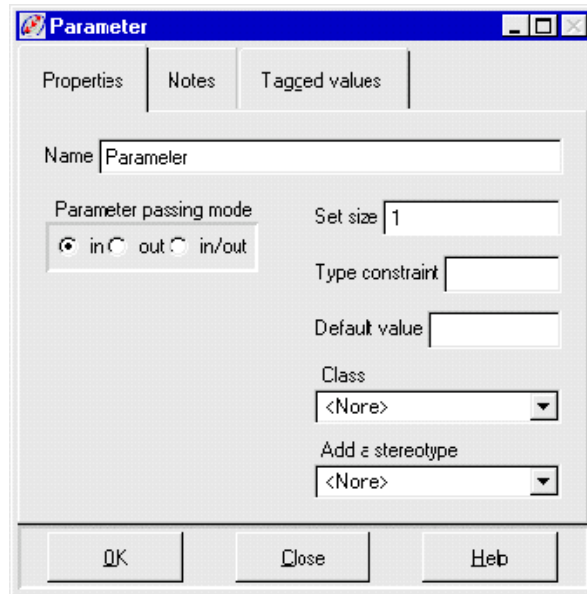


Figure 4-6. The "Parameter" dialog box in Objecteering/UML

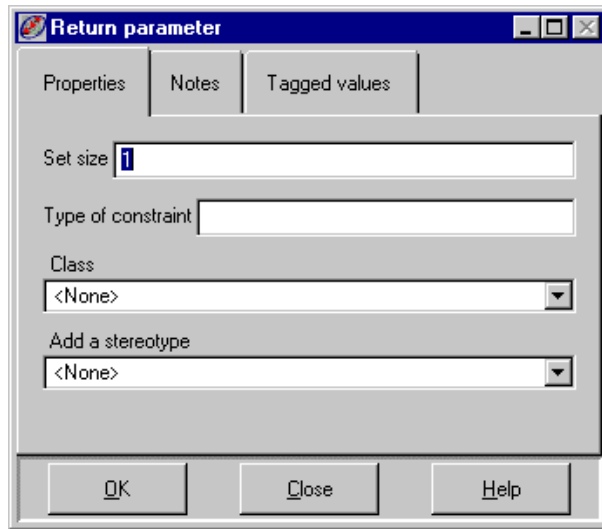


Figure 4-7. The "Return parameter" dialog box in Objecteering/UML

Description of Objectteering/UML dialog box and equivalence in Visual Basic

Objectteering/UML	Description	Equivalent in Visual Basic	
Name	The name of the element.	Name of the parameter	
Parameter Passing Mode	Determines whether the parameter value is provided by the caller (in or in/out), or modified by the triggered operation (in/out or out).	In	In parameter (ByVal)
		Out	Out parameter (ByRef by default)
		InOut	InOut parameter (ByRef by default)
Set size	If the value is other than 1, the parameter is a set of indicated size (* if unlimited, constant or integer).	1	the parameter is not a set
		>1	Collection
Type Constraint	Construction parameter for the parameter class (size of character string, for example).	For strings (by default, almost unlimited if no size if specified).	
Default Value	The parameter's default value (when needed).	Is generated if the parameter is declared at the end of the argument list AND the parameter is tagged VBOptional	
Class	Defines the class to which the parameter belongs.	Parameter type.	

Tagged values used to drive the generation of parameters

The following tagged values are used when generating Visual Basic code:

The ... tagged value	is used to ...
{VBTypeExpr}	replace the type of the unit by the value of the tagged value when generating VB code. For example, you could have a parameter with this tagged value valued at "StdPicture" to generate a StdPicture parameter.
{VBLong} on an integer or real parameter	generate a Long or Double parameter instead of a Integer or Single one. See types correspondances.
{VBOptional}	generate an optional VB parameter.
{VBName}	replace the unit name by the value of the tagged value in the generated code.

Notes

All types can be contained in a Variant.

In Visual Basic, parameters are passed by default via Reference.

A VARIANT can contain all types of Visual Basic data. An object can be passed to an operation through an OBJECT type parameter.

Enumerations

Introduction

Enumerations are generated in the form of VB enumerates. For example, a UML Color enumeration containing the values Red, Green and Blue will be mapped in Visual Basic by:

```
'-----  
' Enumerations  
'-----  
' Enumeration Color : basic RGB colors  
Public Enum Color  
    Blue  
    Green  
    Red  
End Enum
```

Objectteering/UML enumeration type dialog box

Figure 4-8 shows the Objectteering/UML dialog box used to enter or modify enumerations:

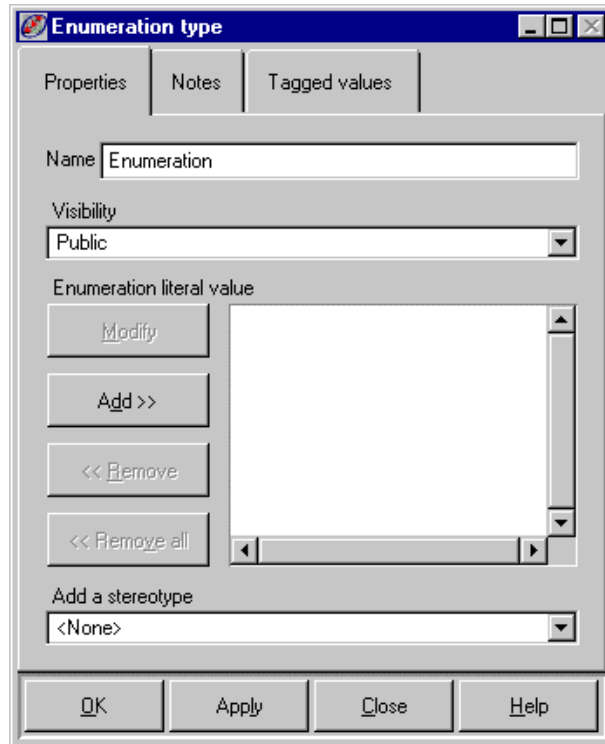


Figure 4-8. The "Enumeration type" dialog box in Objectteering/UML

Description of Objectteering/UML dialog box and equivalence in Visual Basic

Objectteering/ UML	Description	Equivalent in Visual Basic	
Name	The name of the enumeration	The name of the enumeration	
Visibility	Visibility of the member (none, public, protected or private).	Public	Enumeration declared Public (visible throughout the project)
		Other	Enumeration declared Private (visible only in the module)
Enumeration Literal Value	List of literal values of the enumeration. A constant value can be associated to the name using the syntax <name>=<value>	Name of the Visual Basic enumeration member.	

MIDL/Visual Basic/Objectteering equivalence

Visual Basic base types are mapped as shown in the table which follows:

Visual Basic	MIDL	Objectteering/UML
Integer	Short	integer
Long	Long	integer { VBLong }
Single	Float	real
Double	Double	real { VBLong }
Byte	Unsigned char	char
Boolean	Boolean or VARIANT_BOOL	boolean
String	BSTR (OLEstring)	string
Variant	VARIANT	undefined or VB::Variant
Date	DATE	VB::Date
Currency	CURRENCY or CY	VB::Currency
Object	IDispatch *	VB::Object
IUnknown	IUnknown	VB::IUnknown

Note: integer {VBLong} indicates the integer type, annotated with the {VBLong} tagged value. VB::Date indicates the predefined Date type as described in the VB package in Objectteering/UML.

If you wish to use a predefined Visual Basic type which does not exist in Objectteering/UML, you may use one of the following methods:

- ◆ Add a data type, an enumerate or a class to the VB package, and then use it.
- ◆ Create your own VB types package to which you will add your external data types, enumerates and classes. Your other packages must have a use link to this package, in order to use its types.
- ◆ Add a data type, an enumerate or a class to your package, mark this new type as external ({VBExtern} tagged value) and then use it.

Used data types will be translated as simple types (Let properties) whose name will be their modeled name or the content of the {VBName} tagged value if it exists.

Chapter 5: Objecteering/Visual Basic :
Generalization, interface
and types

Generalization, interfaces and polymorphism

Introduction

Pseudo-generalizations implemented as polymorphism at the functional level exist through Visual Basic "interfaces". These pseudo-generalizations are a sub-set of COM (Component Object Model) definitions.

Generalization, interfaces and polymorphism

These interfaces are "abstract classes", which generate a derivable component prototype (DLL or ActiveX, formerly OLE Server). The abstract component thus has the functions of the interface (pseudo-IDL), which allow their redefinition in "derived" module(s). No code is generic in interfaces (everything must be re-implemented in derived classes). Nevertheless, interface objects can be used, by creating one or several instances of derived objects and then allocating the chosen instance to an interface object. The Collection object is used to manage interface objects, as terminal objects (iterator).

Interfaces are COM composed elements, used to make a class polymorphous. They are defined by implementing an "abstract class", which will be inserted into the list of GUIDs in the registry base as being an "object model".

This model is simply a skeleton to be implemented by the user interface.

Intuitive mapping using UML consists of considering Visual Basic interfaces as UML classes stereotyped «interface». In UML, the implementation of an interface by a module is denoted by a realization link from the class towards the interface.

Example of simple use

Figure 5-1 shows an example of simple use:

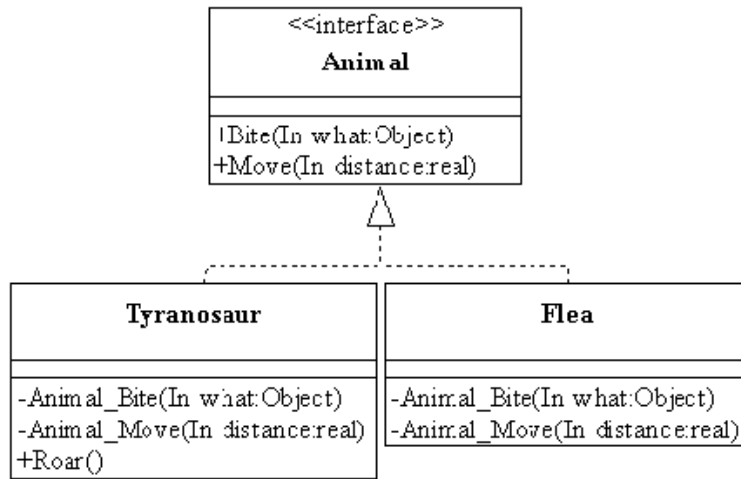


Figure 5-1. An example of simple use

The functions of *Animal* have no code. However, the *Flea* and *Tyrannosaur* objects have the *Animal* object's code. Polymorphism is obtained, by adding functions to derived objects (see *Roar()*). Instantiation then allows one or other of the objects to be used, as shown in the following example :


```

Dim tyr as New Tyrannosaur
Dim fle as New Flea ` mandatory instantiation
Dim obj as Object ` creation of a reference
Set obj = tyr
obj.Bite(fle) : obj.Roar()
Set obj = fle
obj. Bite(tyr) : obj.Move(10)
    
```

Chapter 6: The Objecteering/Visual
Basic interface

The Objecteering/Visual Basic interface

Creating a generation work product

In Objecteering/UML, a Visual Basic generation work product can be associated to a package or a class, using the  "Create a Visual Basic generation work product" icon, which is found in the "Items" of the properties editor (as shown in Figure 6-1).

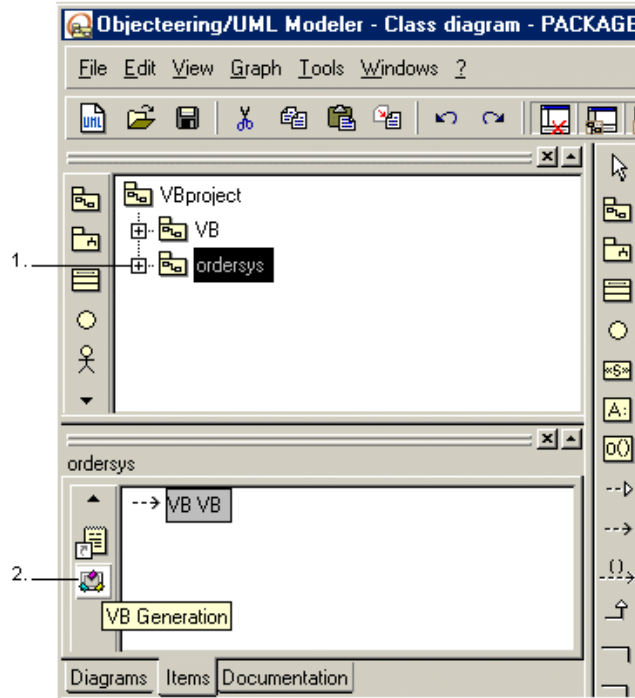



Figure 6-1. Associating a Visual Basic generation work product to a package

Steps:

- 1 - Select the "ordersys" package in the explorer.

- 2 - Click on the  "Create a Visual Basic generation work product" icon in the "Items" tab of the properties editor. The following dialog box then appears.

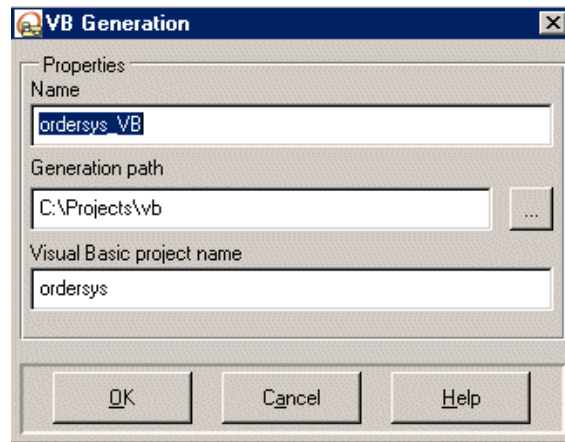


Figure 6-2. The "VB Generation" dialog box

The ... field	is used to ...
Name	specify the name of the generation work product.
Generation	specify the location of the files created when generation is run.
Visual Basic project name	specify the name of the Visual Basic project.

Note: If you select, for example, C:\projects\VB as the generation path for the "ordersys" package, the cls files will be generated in C:\projects\VB\ordersys.

The work product created appears in the "Items" tab of the properties editor.

Now carry out the following steps:

- 1 - Select the newly created generation work product, and open its context menu, by clicking on the right mouse button.
- 2 - Generate the Visual Basic code by running the "Generate" menu item available in the context menu on the Visual Basic generation work product.

Note: Using the "Generate" command at package level generates VB code for all classes in the package.

- 3 - Use Visual Basic to create your Visual Basic project with the same name and in the same directory as the package (c:\projects\VB\ordersys\ordersys.vbp in our example). In Visual Basic, use the "Project/Add File..." menu item to add all the generated cls files. This step only has to be carried out once, but each time you add a new class in Objecteering/UML, you will have to add it to your VB project. Exit Visual Basic.

Note: You can use the {VBName = ...} tagged value on a package to change the VB project name. For example, if you put the {VBName = My_VB_Project} tagged value on the "ordersys" package, the code will be generated in C:\projects\VB\My_VB_Project, instead of C:\projects\VB\ordersys, and your VB project should be stored in C:\projects\vb\My_VB_Project.vbp.

- 4 - Edit your VB project in Visual Basic. You can modify the code in the "START OF MODIFIABLE ZONE..." and "END OF MODIFIABLE ZONE ..." zones.
- 5 - Update the Objecteering/UML contents from the modification you made to your VB code, by running the "Update from VB Code" menu command, available in the context menu on the Visual Basic generation work product.

Note: You can visualize code generated for a class, by running the "Visualize" command, available in the context menu on the Visual Basic generation work product.

Chapter 7: Parameterizing the
Objectteering/Visual Basic
module


Overview of module parameterization

Introduction

The *Objecteering/Visual Basic* module provides the user with the possibility of customizing the following parameters:

- ◆ the editor used when editing generated code
- ◆ VB code generation parameters

The "Edit configuration" window

The window through which the *Objecteering/Visual Basic* module can be configured is opened either by clicking on the  "Modify module parameter configuration" icon or by clicking on the "Tools/Modify configuration..." menu in the Objecteering/UML menu bar.

In the "Edit configuration" window, the behavior of VB code generation can be modified, through the following elements:

- ◆ Generation directory
 - ◆ Generation options
 - ◆ UML profile containing the J rules
 - ◆ Generation template
-

Parameter sets

The "External edition" parameter set

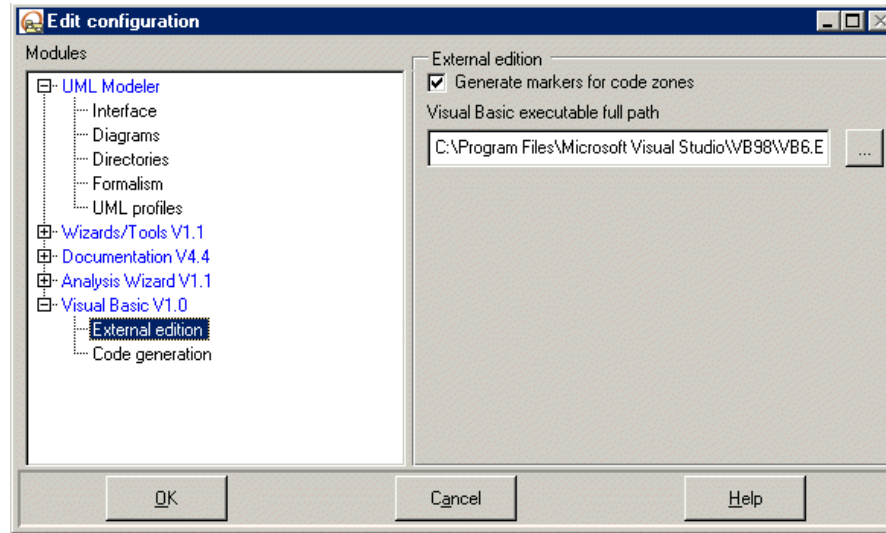


Figure 7-1. The "External edition" set of parameters for the Visual Basic module

The ... parameter	indicates ...
Generate identifiers	whether or not the markers used to retrieve text entered using an external text editor should be generated.
Visual Basic executable full path	the command used to launch an editor to modify the generated code.

The "Code generation" parameter set

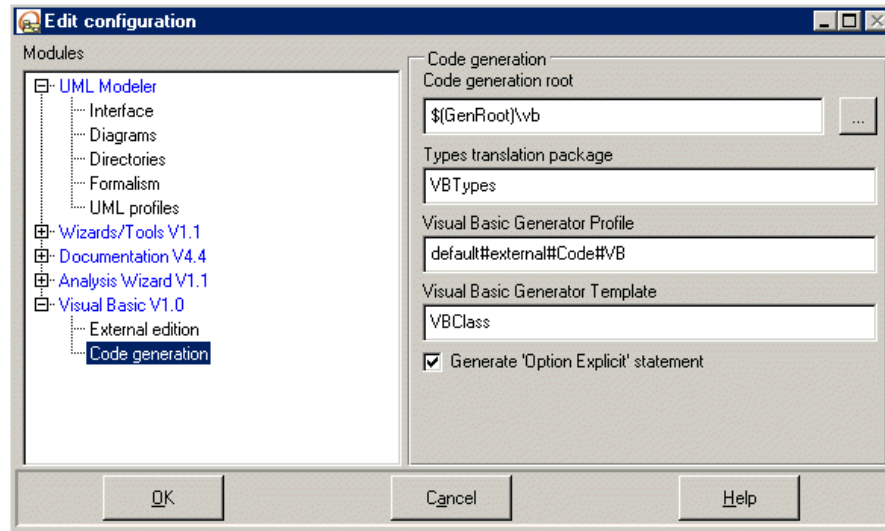


Figure 7-2. The "Code generation" set of parameters for the Visual Basic module

The ... parameter	indicates ...
Code generation root	the root in which .cls files are generated.
Types translation package	the name of the project used for translation types and the generation of accessors.
Visual Basic generator profile	the UML profile used to generate VB code.
Visual Basic generator template	VB code generation template used for classes.
Generate "Option explicit" statement	that the option will be generated in the file header.

Index

- .cls files 4-10
- .res files 1-7
- .vbproj files 1-7
- {NoCode} tagged value 2-6, 4-15, 4-21, 4-31
- {persistent} tagged value 1-5
- {VBExtern} tagged value 4-9, 4-41
- {VBFilterProperties} tagged value 2-9, 4-15, 4-21
- {VBLong} tagged value 4-15, 4-36, 4-41
- {VBName} tagged value 2-6, 4-9, 4-15, 4-21, 4-31, 4-36, 4-41
- {VBNoProperties} tagged value 2-9, 4-15, 4-21
- {VBNoSet} tagged value 2-7, 4-9, 4-24
- {VBOptional} tagged value 4-36
- {VBTypeExpr} tagged value 4-15, 4-21, 4-36
- <<create>> stereotype 1-6, 4-30
- <<Destroy>> stereotype 4-30
- <<VBGet>> stereotype 4-30
- <<VBLet>> stereotype 4-30
- <<VBSet>> stereotype 4-30
- Accessors 4-11
- ActiveX 1-7, 1-8, 5-3
- ActiveX type OLE servers 1-8
- Associations 4-3
- Attributes 4-3
- Choosing a reference in Visual Basic 3-15
- Class attributes 4-16
- Class module code 4-23
- Class modules 1-7, 4-4, 4-25
- Classes 4-3
- Code generation 4-29
- Code generation parameters 7-5
- Code generation procedure 3-5
- Collections 1-8, 4-16
- COM 5-3
- Component Object Model 5-3
- Creating a generation work product 3-5, 6-3
- Creating a UML modeling project 2-3
- Creating a working UML modeling project 2-3
- Customizing module parameters 7-3
- Defining new UML predefined element sub-types 1-6
- Dependencies 4-3
- Description note 4-10, 4-15, 4-21, 4-31
- DLL 1-7, 1-8, 5-3
- Editing generated code 3-9
- Enumerations 1-3
- Explorer 3-9, 3-12
- External edition parameters 7-4
- Friend 4-16
- Generalization 5-3
- Generalizations 4-3
- Generating code 3-6
- Generating module content 1-3
- Generating Visual Basic class modules from UML classes 1-3
- Generation directory 3-7, 7-3
- Generation options 7-3
- Generation template 7-3
- Generation templates 1-6, 4-5
- Generation work products 3-5, 3-9, 3-12
- Importing the Visual Basic first steps project 3-4

- Initializing the First Steps UML modeling project 3-4
- Instances 4-3
- Interfaces 5-3
- Internet OCX 1-7
- Late binding 1-8
- Note types 4-16
- Notes 1-5, 4-9, 4-15, 4-21, 4-24, 4-28, 4-30, 4-36
 - Description 4-10, 4-15, 4-21, 4-31
 - Summary 4-10, 4-31
 - VBCode 4-10, 4-29, 4-31
 - VBProperties 4-10
- Notes and tagged value for driving generation of attributes 4-15
- Notes and tagged values for driving generation of associations 4-21
- Notes and tagged values used to drive the generation of classes 4-9
- Notes and tagged values used to drive the generation of operations 4-30
- Notes and tagged values used to drive the generation of parameters 4-36
- Object encapsulation 4-20
- Objecteering/Introduction 2-5
- Objecteering/UML association dialog box 4-18
- Objecteering/UML attribute dialog box 4-12
- Objecteering/UML class dialog box 4-3
- Objecteering/UML dialog box and equivalence in Visual Basic 4-7, 4-13, 4-19, 4-27, 4-35, 4-39
- Objecteering/UML enumeration type dialog box 4-38
- Objecteering/UML Modeler 2-3
- Objecteering/UML operation dialog box 4-26
- Objecteering/UML parameter dialog box 4-33
- Objecteering/UML return parameter dialog box 4-33
- OCX 1-7
- OCX components 1-7
- OLB 1-7
- OLE 1-7
- OLE automation lib 4-24
- OLE Server 5-3
- OLE servers 1-7, 1-8
- Operations 4-3
- PKG 1-7
- Polymorphism 5-3
- Properties editor 3-7, 3-9, 3-12, 6-5
 - The "VB" tab of the properties editor for a class 2-7
 - The "VB" tab of the properties editor for a package 2-6
 - The "VB" tab of the properties editor for an association 2-10
 - The "VB" tab of the properties editor for an attribute 2-9
 - The "VB" tab of the properties editor for an operation 2-8
- Pseudo-constructors 1-7
- Pseudo-destructor 1-7
- Pseudo-destructors 1-7
- Pseudo-generalizations 5-3
- Running code 3-13
- Running options in Visual Basic 3-14
- Selecting the Objecteering/Visual Basic module 2-3
- Selecting the VBModule for a new UML modeling project 2-4
- SQL code generator 1-5
- Stereotypes 1-6, 4-30
 - <<create>> stereotype 1-6, 4-30

- <<Destroy>> stereotype 4-30
- <<interface>> stereotype 2-7
- <<VGet>> stereotype 4-30
- <<VLet>> stereotype 4-30
- <<VSet>> stereotype 4-30
- Summary note 4-10, 4-31
- Tagged values 1-5, 4-9, 4-15, 4-21, 4-30
 - {NoCode} tagged value 2-6, 4-15, 4-21, 4-31
 - {persistent} tagged value 1-5
 - {VBExtern} tagged value 4-9, 4-41
 - {VFilterProperties} tagged value 2-9, 4-15, 4-21
 - {VBLong} tagged value 4-15, 4-36, 4-41
 - {VBName} tagged value 2-6, 4-9, 4-15, 4-21, 4-31, 4-36, 4-41
 - {VBNoProperties} tagged value 2-9, 4-15, 4-21
 - {VBNoSet} tagged value 2-7, 4-9, 4-24
 - {VBOptional} tagged value 4-36
 - {VBTypeExpr} tagged value 4-15, 4-21, 4-36
- The Visual Basic menu used to add objects to a Visual Basic project 4-4
- Translating UML design into Visual Basic code 1-3
- Types 1-3
- UML annotations 1-3
- UML classes 1-3
- UML models 1-3
- UML operation parameters 4-32
- UML profile containing the J rules 7-3
- UML types 1-5
- VB code generation parameters 7-3
- VB enumerates 4-37
- VBCode note 4-10, 4-31
- VBProperties note 4-10
- Visual Basic base types 4-40
- Visual Basic class modules 1-3, 4-4, 4-25
- Visual Basic code 1-3
- Visual Basic commands
 - Edit 3-12
 - Generate 3-7, 6-5
 - Update from VB code 6-5
 - Update from Visual Basic code 3-12
 - Visualize 3-9, 6-5
- Visual Basic components 1-7, 1-8
- Visual Basic Forms 1-7
- Visual Basic generation work products 3-5, 6-3
- Visual Basic project 1-7
- Visual Basic properties 4-11