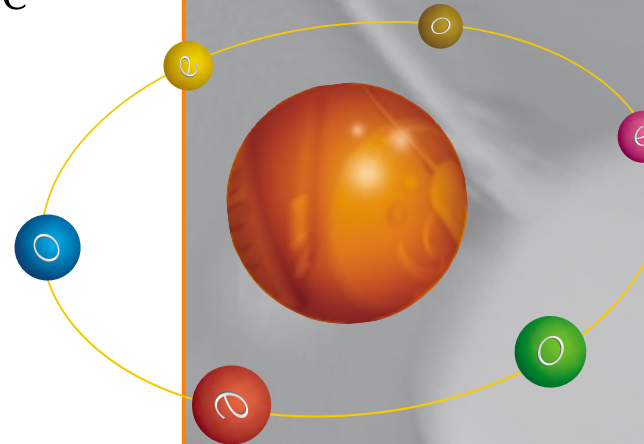


Objecteering/UML

Objecteering/C++ 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 001/003

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	
Overview of the Objectteering/C++ module	1-3
Structure of the Objectteering/C++ user guide	1-4
Overview of C++ development steps	1-5
Modeling	1-6
Generating files	1-7
Generating makefiles and compiling	1-8
Parameterizing the Objectteering/C++ generator	1-9
The properties editor for C++	1-10
Chapter 2: Using the Objectteering/C++ module	
Working with the Objectteering/C++ module	2-3
The properties editor and the Objectteering/C++ module	2-5
Chapter 3: First Steps	
Prerequisites	3-3
Creating a generation work product	3-6
Visualizing the generated code	3-11
Creating a compilation work product	3-16
Generating a makefile	3-20
Visualizing the makefile	3-22
Executing the makefile	3-25
Chapter 4: Compilation and generation work products	
The two work products in the Objectteering/C++ module	4-3
The attributes of a generation work product	4-5
The attributes of a compilation work product	4-7
The context menus of generation and compilation work products	4-16
Generating C++	4-22
Editing the code file	4-23
Updating the repository	4-26
Analyzing the compilation	4-28
Visualizing the file	4-30
Chapter 5: Tagged values and notes specific to C++	
Overview of tagged values and notes specific to C++	5-3
Tagged values	5-4
Notes	5-7

Chapter 6: Calling module commands in batch mode	
Calling the module's commands in batch mode - Overview	6-3
Calling commands	6-4
Invocable commands	6-5
Chapter 7: Module parameters	
Definition of module parameters	7-3
Detailed description of parameters	7-5
Chapter 8: Generating code for a package	
Overview of code generation on a package	8-3
Tagged values for a package	8-6
Chapter 9: Generating code for a class	
Overview of code generation on a class	9-3
Class generalization	9-5
Class invariant	9-7
Main class	9-9
Generic class: Template	9-11
Tagged values on a class	9-13
Notes on a class	9-15
Chapter 10: Generating code for an attribute	
Overview of code generation on an attribute	10-3
Declaration and initialization	10-6
Access methods	10-9
Tagged values on an attribute	10-11
Notes on an attribute	10-17
Chapter 11: Generating code for associations	
Overview of code generation on an association	11-3
Generation	11-5
Handling instances	11-8
Tagged values on an association	11-9
Chapter 12: Generating code for an operation	
Overview of code generation on an operation	12-3
Operation	12-4
Parameter	12-7
Detailed tagged values on an operation	12-10
Notes on an operation: Detailed description	12-12
Constraints on an operation: Detailed description	12-14
Detailed tagged values on a parameter	12-16
Notes on a parameter: Detailed description	12-20

Chapter 13: Generating code for a generalization, an enumerate and basic types	
Introduction	13-3
Generating basic types	13-4
Generating a new type.....	13-6
Generating an enumerate.....	13-7
Generating a generalization	13-9
Creating an instance.....	13-11
Chapter 14: Parameterizing basic types	
Parameterizing basic types.....	14-3
Package structure	14-6
BaseTypes	14-7
Translating set type associations, parameters and attributes.....	14-9
The translation class typeAttribute or typeAssociation	14-11
The translation class typeIOParameter or typeReturnParameter	14-13
DefaultTranslations	14-14
Chapter 15: Adapting C++ code generation	
Overview of C++ generation adaptation.....	15-3
Generation constant	15-6
Naming and filtering constants	15-7
Parameterizing the generation of different units	15-16
Package	15-17
Class	15-19
Attributes	15-24
Associations	15-30
Operations.....	15-37
Parameters.....	15-45
DataType.....	15-48
Header	15-49
ModelElement	15-50
Enumeration.....	15-52
Generalization.....	15-54
Member	15-56
Examples	15-58
Chapter 16: Adapting makefile generation	
Adapting the makefile	16-3
Accessing the compilation work product	16-7
Compilation work product attributes	16-8
Module parameters	16-9
Packages - part 1.....	16-11
Packages - part 2.....	16-12

Chapter 17: The Objectteering/UML library	
Overview of the Objectteering/UML library.....	17-3
CR_Boolean.....	17-5
CR_String.....	17-6
Set of primitive objects.....	17-8
Set of non-primitive objects	17-11
Chapter 18: Coverage of C++ by Objectteering/UML	
Objectteering/UML coverage of C++	18-3
Basic types.....	18-5
Built types.....	18-6
Nested declarations	18-8
Features	18-10
Friendly operators	18-12
Generalization between classes.....	18-14

Index

Chapter 1: Introduction

Overview of the Objecteering/C++ module

Welcome to the *Objecteering/C++* user guide!

The *Objecteering/C++* module covers the entire development process, from the modeling design phase right up to the final application (executable or library).

Using the *Objecteering/C++* module, it is possible to:

- ◆ create texts (notes) and annotations (tagged values) specific to C++
- ◆ generate C++ code from the model
- ◆ enter C++ code either in the model or directly in the files
- ◆ create "*makefiles*" that can be parameterized by the user
- ◆ compile

The *Objecteering/C++* module constantly guarantees the consistency between the generated C++ source, the binary and the model. It can also assist you during the development cycle, by helping you to correct the compilation errors in a graphic environment.

One major strength of the *Objecteering/C++* module is its powerful parameterization feature. The implementation of certain modeling elements can be defined through tagged values, and generation rules can even be enriched or redefined, in order to adapt the generated code to your style of programming.

Structure of the Objectteering/C++ user guide

This user guide has been designed for users of the *Objectteering/C++* module. It will guide you through the modeling and the realization of an application, and constitutes a reference manual for understanding and using dedicated tagged values or notes.

The *Objectteering/C++* user guide is divided into the following chapters:

- ◆ Chapter 1: Introduction
 - ◆ Chapter 2: Using the Objectteering/C++ module
 - ◆ Chapter 3: First Steps, providing a step by step insight into the workings of the *Objectteering/C++* module
 - ◆ Chapter 4: Overview of generation and compilation work products
 - ◆ Chapter 5: Overview of the available notes and tagged values
 - ◆ Chapter 6: Calling module commands
 - ◆ Chapter 7: Module parameters
 - ◆ Chapters 8 to 13: Generation of C++ code
 - ◆ Chapters 14 to 16: Adapting C++ generation
 - ◆ Chapter 17: The Objectteering/UML library
 - ◆ Chapter 18: Coverage of C++ by Objectteering/UML
-

Overview of C++ development steps

Presentation

Objectteering/UML is a graphic design tool, used to carry out C++ code generation. This tool is based on two essential elements: the generation work product and the compilation work product. The former maintains consistency between the generated code and the model, whilst the latter aims at compiling code to obtain the application. For further information on these work products, please refer to chapter 4, "*Compilation and generation work products*", of this user guide.

The development steps are shown in Figure 1-1.

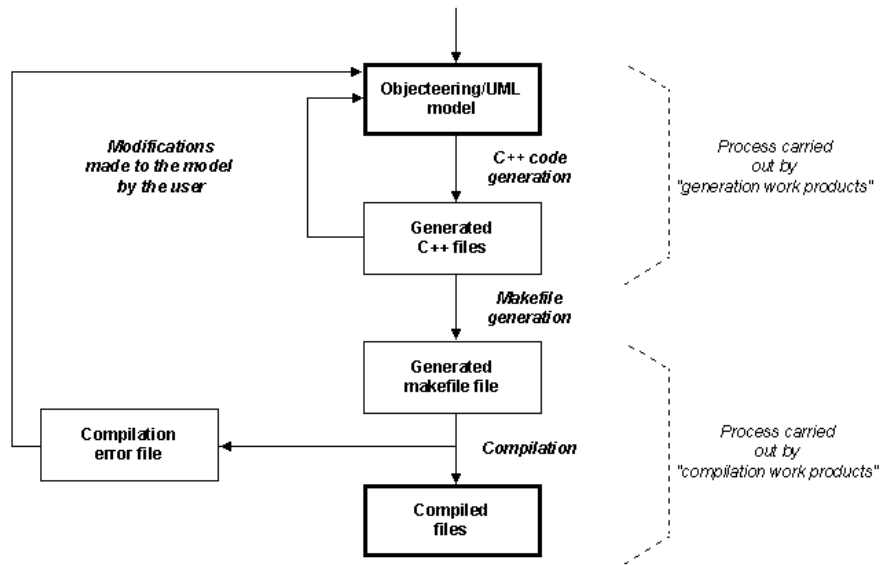


Figure 1-1. Development steps

Modeling

Modeling an application consists of:

- ◆ creating a UML model
- ◆ annotating the elements of this model according to the selected types of implementation

The *Objectteering/C++* module provides a set of notes and tagged values specific to C++, which can be used to annotate model elements. For further information on these notes and tagged values, please refer to chapter 4, "*Tagged values and notes specific to C++*", of this user guide.

Notes are used to enter the text that will be inserted in the generated code. For example, a C++ note attached to a method's body will contain the code of an algorithm for the function; the "*C++BodyHeader*" note inserts the "*include*" code into the header file.

Tagged values inform the generator of the implementation rules. For example, *{array}* on an association means that the implementation is a kind of array and *{virtual}* on an operation means that the operation will be a virtual function, etc.

Note: User assessments show that over 70% of the generated C++ code comes from the model, whilst the remaining 30% comes from additional "*notes*".

Generating files

The *Objectteering/C++* module generates a pair of files (*header* and *body*) for each package and its classes.

When modeling, it is not necessary to supply all the C++ notes for each model element. The *Objectteering/C++* module can deduce them from the model and add the necessary markers in the generated code file. This means that the user can edit the generated file using the editor of his choice. The *Objectteering/UML* repository is then updated automatically.

Editing can be carried out inside or outside the tool. In both modes, the update of *Objectteering/UML*'s repository is guaranteed. Cases such as simultaneous modification of the same file following two different modes are no longer ambiguous, as the user can choose which code must be taken into account, through a dedicated graphic editor. In this way, the generation work product guarantees the consistency between the model and code files.

Figure 1-2 shows when the user intervenes during the modeling and development phases.

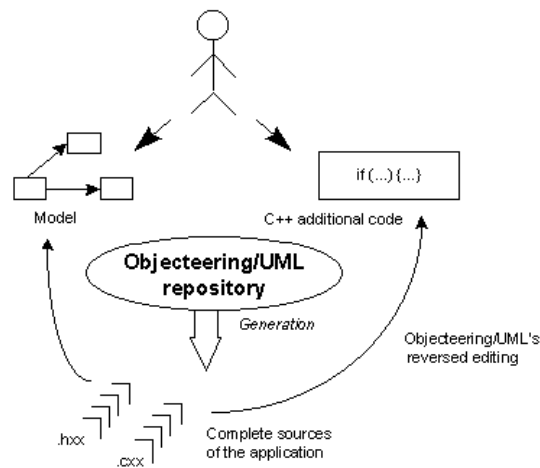


Figure 1-2. The generation produces comprehensive sources

Generating makefiles and compiling

The compilation work product generates a makefile for each package, calls the compiler and displays the results of the compilation in a dialog box. This dialog box is divided into two parts. The upper part displays incorrect code, whilst the lower part presents the error message emitted by the compiler. This graphic environment guides you intuitively to correct errors in the model and code. Your corrections are taken into account in the Objectteering/UML repository so that the final work product will be consistent with the model.

Parameterizing the Objecteering/C++ generator

The *Objecteering/C++* module provides various ways to parameterize the generated code in order to adapt it to your programming style.

- ◆ Parameterizing through tagged values: by annotating the model with tagged values, the user can specify code generation. The default case contains no annotations.
 - ◆ Parameterizing basic types (Enterprise version only): the code generated for attributes, associations and parameters is produced by the C++ generator which is associated to the basic types project: *ObjecteeringTypes* and *STLTypes* (Standard Template Library) and *MFC Types* (Microsoft Foundation Classes).
 - ◆ Parameterization of basic type default selection according to the modelit is possible to edit the basic types project, in order to modify the default selection, in order to implement attributes and associations according to their multiplicity.
 - ◆ Parameterizing by creating a type project: any user can use the basic types library of his choice in the generated code. This feature is obtained by creating and editing a specific types project. Respecting certain rules, the type project is then used by the basic generator to produce C++ code. (For further information, please refer to chapter 12, "*Parameterizing basic types*", of this user guide).
 - ◆ Parameterizing using the *UML Profile Builder* tool: the generated code can be adapted by redefining the J methods in charge of producing zones of C++ code. Design patterns can be automated and redefined using this feature.
-

The properties editor for C++

The properties editor is essentially a window designed to aid the user in his modeling, by providing rapid access to various information and services he may need to use.

The properties editor contains a number of tabs, including a "C++" tab when the *Objectteering/C++* module has been selected for the current UML modeling project. This tab is used to enter or modify certain information relevant to C++ generation on the element selected in the explorer, such as notes, tagged values or stereotypes.

For further general information on the properties editor, please refer to the "*The Properties editor*" section in chapter 3 of the *Objectteering/UML Modeler* user guide.

For further information on the "C++" tab of the properties editor, please refer to the "*The properties editor and the Objectteering/C++ module*" section in chapter 2 of this user guide.

Chapter 2: Using the Objecteering/C++ module

Working with the Objecteering/C++ module

Installing the Objecteering/C++ module

The *Objecteering/C++* module is delivered to and installed on your Objecteering/UML site automatically during the Objecteering/UML installation procedure. For further information, please refer to chapter 2 of the *Objecteering/Introduction* user guide.

Note: The complete module delivery and installation procedure is fully explained in the "*Detailed view of the Configuration menu*" in chapter 3 of the *Objecteering/Administrating Objecteering Sites* user guide.

Selecting the Objecteering/C++ module

The only operation the user has to carry out in order to be able to use the *Objecteering/C++* module is the actual selection of the module itself for the current UML modeling project. This operation is described in the "*Selecting modules in the current project*" section in chapter 3 of the *Objecteering/Introduction* user guide.

Note: It should be noted that if the user wishes to compile, he is obliged to go into the module parameters, in order to enter the compiler path, libraries, and so on.

Please note that because installation of the *Objecteering/C++* module is carried out automatically during installation of the Objecteering/UML tool itself, module parameters are standard. To customize these module parameters, simply edit the "*Modify module parameter configuration*" window. For further information on modifying module parameters, please refer to the "*Configuration window*" section in chapter 16 of this user guide.

The properties editor for C++

It should be noted that where a version of the *Objecteering/C++* module which adds a tab to the properties editor is selected in place of an earlier version of the module which did not provide this service, you should quit and restart Objecteering/UML, in order for the properties editor to be correctly displayed.

Selecting the C++ model type

When creating a new UML modeling project, it is possible to select the "*DefaultCpp*" UML model type in the "*UML model type*" field. By doing this, the *Objecteering/C++* module is automatically selected for use in your new UML modeling project.

The properties editor and the Objecteering/C++ module

The "C++" tab of the properties editor on a package

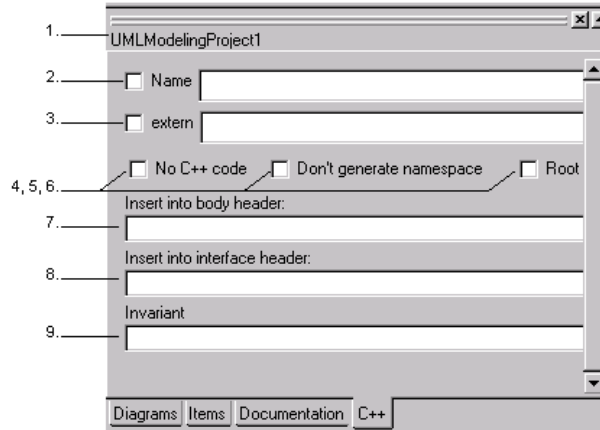


Figure 2-1. The "C++" tab of the properties editor on a package

Key:

- 1 - This provides the name of the package selected in the explorer.
- 2 - This field is used to add the {C++Name} tagged value.
- 3 - This field is used to add the {extern} tagged value.
- 4 - This field is used to add the {nocode} tagged value.
- 5 - This field is used to add the {C++NoNameSpace} tagged value.
- 6 - This field is used to add the {C++Root} tagged value.
- 7 - This field is used to add the "C++BodyHeader" note.
- 8 - This field is used to add the "C++InterfaceHeader" note.
- 9 - This field is used to add the "C++Invariant" constraint.

The "C++" tab of the properties editor on a class

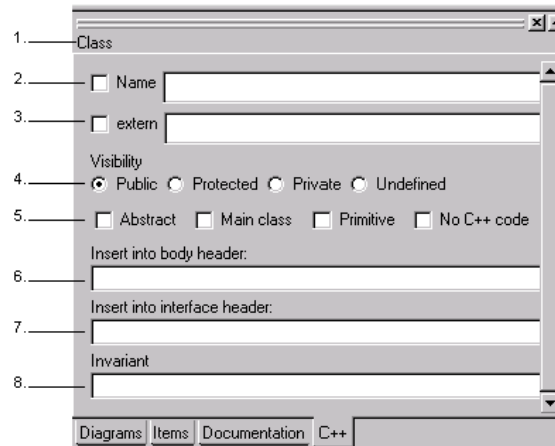


Figure 2-2. The "C++" tab of the properties editor on a class

Key:

- 1 - This provides the name of the class selected in the explorer.
- 2 - This field is used to add the {C++Name} tagged value.
- 3 - This field is used to add the {extern} tagged value.
- 4 - The "Visibility" radio buttons are used to select the visibility of the class.
- 5 - The tickboxes which appear here are used to indicate the nature of the class:
 - ◆ "Abstract": This means that the class is abstract.
 - ◆ "Main class": This defines the class as being main.
 - ◆ "Primitive": This means that the class is primitive.
 - ◆ The "No C++ code" tickbox is used to add the {nocode} tagged value.
- 6 - This field is used to add the "C++BodyHeader" note.
- 7 - This field is used to add the "C++InterfaceHeader" note.
- 8 - This field is used to add the "C++Invariant" constraint.

The "C++" tab of the properties editor on an operation

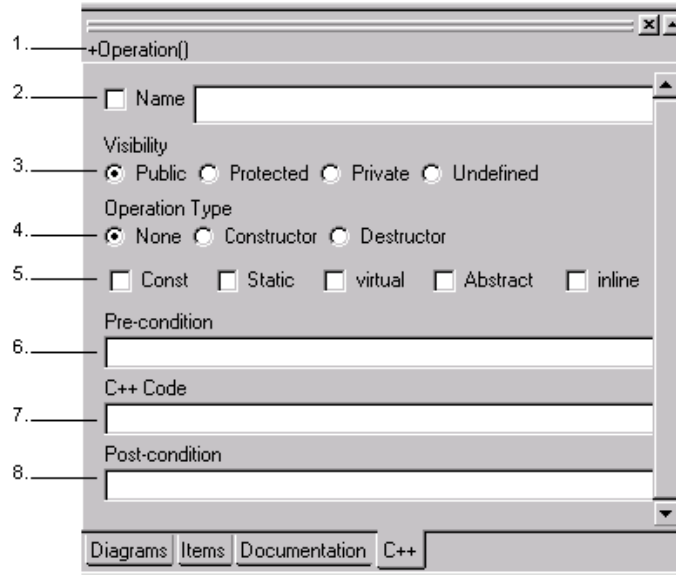


Figure 2-3. The "C++" tab of the properties editor on an operation

Chapter 2: Using the Objectteering/C++ module

Key:

- 1 - This provides the name of the operation selected in the explorer.
- 2 - This field is used to add the {C++Name} tagged value.
- 3 - The "*Visibility*" radio buttons are used to select the visibility of the class.
- 4 - The "*Operation type*" buttons which appear here are used to indicate the operation type:
 - ◆ "*None*": This means that the operation has no special type.
 - ◆ "*Constructor*": This adds the <<*create*>> stereotype to the operation, and means that it is a constructor of its class.
 - ◆ "*Destructor*": This adds the <<*destroy*>> stereotype to the operation.
- 5 - These tickboxes are used as follows:
 - ◆ "*Const*": This generates "const" operations.
 - ◆ "*Static*": This defines operations as being class.
 - ◆ "*virtua*": This adds the {virtual} tagged value to the operation.
 - ◆ "*Abstract*": This defines the operation as being abstract.
 - ◆ "*inline*": This adds the {inline} tagged value to the operation.
- 6 - This field is used to add the "*C++PreCondition*" note.
- 7 - This field is used to add the "*C++*" note. .
- 8 - This field is used to add the "*C++PostCondition*" note. .

Note: For operations with return parameters, an additional field is available, "*C++ return code*", which is used to add the "*C++Returned*" note.

The "C++" tab of the properties editor on an attribute

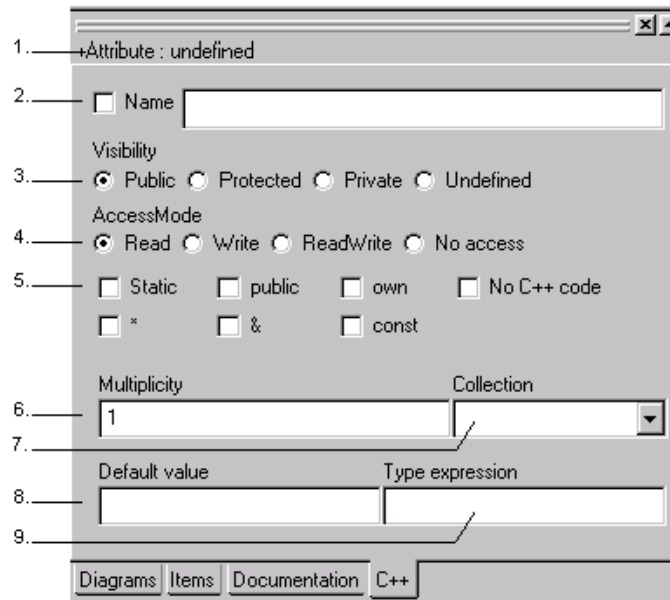


Figure 2-4. The "C++" tab of the properties editor on an attribute

Chapter 2: Using the Objectteering/C++ module

Key:

- 1 - This provides the name of the attribute selected in the explorer.
- 2 - This field is used to add the {C++Name} tagged value.
- 3 - The "*Visibility*" radio buttons are used to select the visibility of the attribute.
- 4 - The buttons which appear here are used indicate the attribute's access mode.
- 5 - These tickboxes are used as follows:
 - ◆ "*Static*": This defines the attribute as being class.
 - ◆ "*public*": This adds the {public} tagged value.
 - ◆ "*own*": This adds the {own} tagged value.
 - ◆ "*No C++ code*": This adds the {nocode} tagged value.
 - ◆ "***": This adds the {*} tagged value.
 - ◆ "*&*": This adds the {&} tagged value.
 - ◆ "*const*": This adds the {const} tagged value.
- 6 - The "*Multiplicity*" field is used to enter the attribute's multiplicity.
- 7 - This field is used to add the {type(...)} tagged value.
- 8 - This field is used to define the default value of the attribute.
- 9 - This field is used to add the "*C++TypeExpr*" note.

The "C++" tab of the properties editor on an association

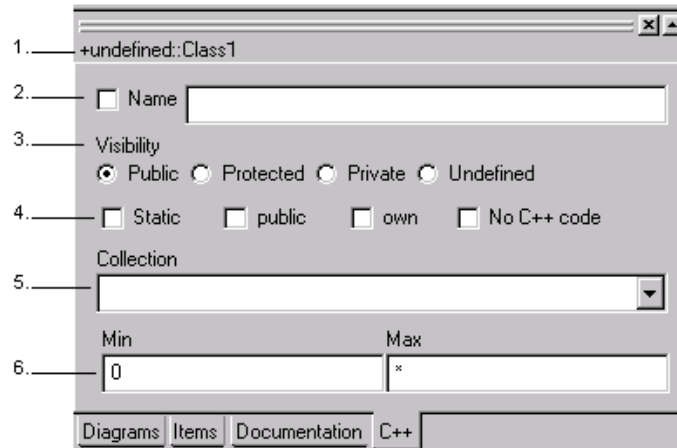


Figure 2-5. The "C++" tab of the properties editor on an association

Key:

- 1 - This provides the name of the association selected in the explorer.
- 2 - This field is used to add the {C++Name} tagged value.
- 3 - The "*Visibility*" radio buttons are used to select the visibility of the association.
- 4 - These tickbuttons are used as follows:
 - ◆ "*Static*": This defines the association as being class.
 - ◆ "*public*": This adds the {public} tagged value.
 - ◆ "*own*": This adds the {own} tagged value.
 - ◆ "*No C++ code*": This adds the {nocode} tagged value.
- 5 - This field is used to add the {type(...)} tagged value.
- 6 - These fields are used to define the minimum and maximum multiplicities of the association.

The "C++" tab of the properties editor on a parameter

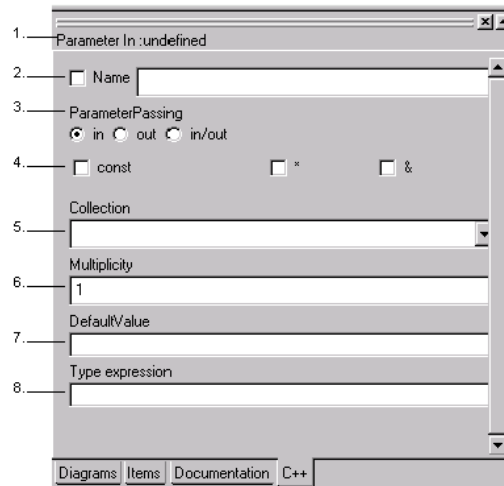


Figure 2-6. The "C++" tab of the properties editor on a parameter

Key:

- 1 - This provides the name of the parameter selected in the explorer.
- 2 - This field is used to add the {C++Name} tagged value.
- 3 - This field is used to set the parameter passing mode.
- 4 - These tickboxes are used to add the {const}, {*} and {&} tagged values.
- 5 - This field is used to add the {type(...)} tagged value.
- 6 - The "Multiplicity" field is used to define the multiplicity of the parameter.
- 7 - This field is used to set the default value of the parameter.
- 8 - This field is used to add the "C++TypeExpr" note.

Note: The same fields are displayed in the C++ tab of the properties editor on return parameters, except that the "Parameter passing" and "Default value" fields are not present.

Chapter 3: First Steps

Prerequisites

Getting started

In these first steps, we are going to be using the "*VendingMachine*" demonstration UML modeling project to generate C++ code and produce an executable. By following this example, you will discover, step by step, the different features of the *Objectteering/C++* module.


We recommend that before starting, every user carry out the general Objectteering/UML first steps in the *Objectteering/Introduction* user guide.

Note: <\$OBJING_PATH> designates the Objectteering/UML installation directory.

Initializing the First Steps UML modeling project

A First Steps UML modeling project is delivered along with the *Objectteering/First Steps* module. It is located in <\$OBJJING_PATH>/modules/CxxModule/FirstSteps.

Before starting, you must:

- 1 - Create a UML modeling project (for example, "*VendingMachine*") (for further details, please refer to the "*Creating or opening a UML modeling project*" section in chapter 3 of the *Objectteering/UML Modeler* user guide).
- 2 - Select the C++ module for your UML modeling project. This selection is made using the  "*UML modeling project modules*" icon.
- 3 - Import the contents of the "*FirstSteps*" UML modeling project (as shown in Figure 3-1), by following the steps explained below:

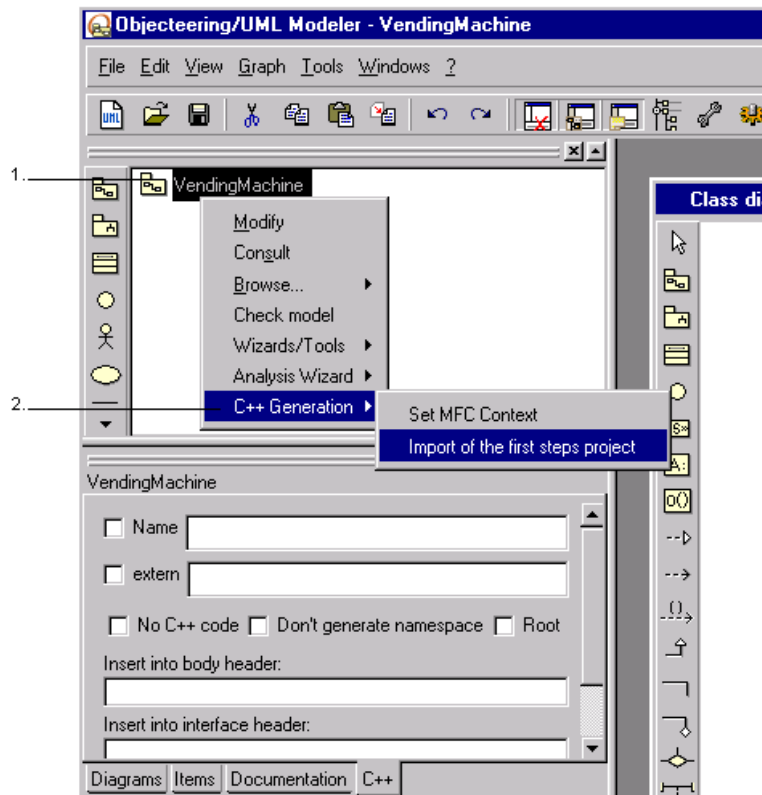


Figure 3-1. Importing the "FirstSteps" UML modeling project

Steps:

- 1 - Select the model root, and click on the right mouse button to display the associated context menu.
- 2 - Run the "C++ Generation/Import of the first steps project" options from the context menu.

Creating a generation work product

Procedure

Taking this example as a starting point, we will begin by creating a generation work product for a package (figure 3-2).

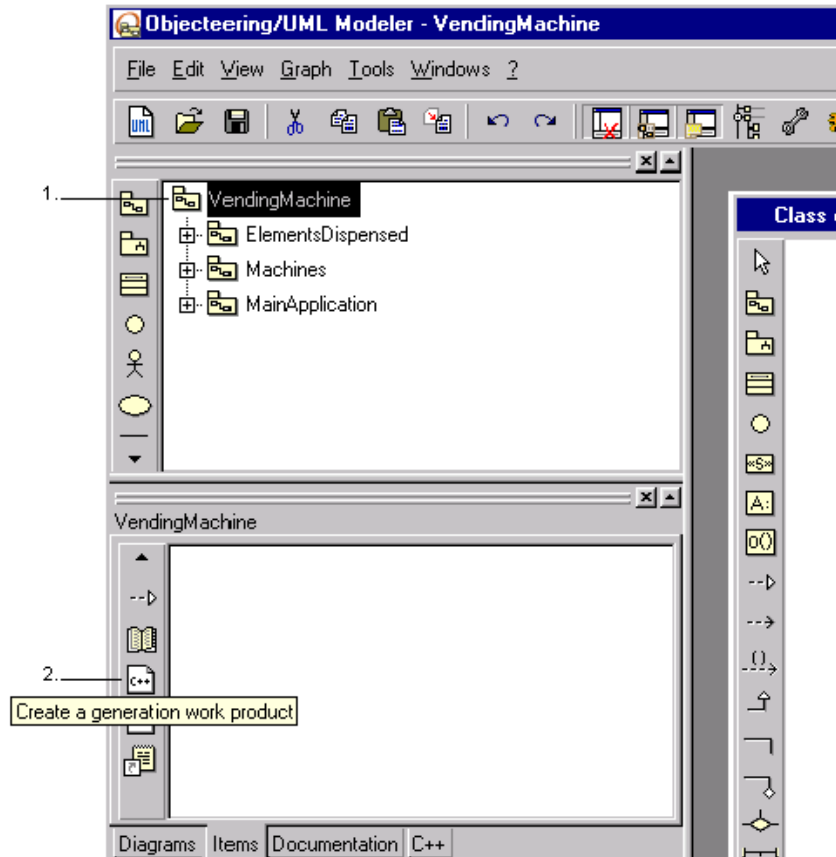



Figure 3-2. Creating a generation work product in the "VendingMachine" package

Steps:

1 - Select the "*VendingMachine*" package.

2 - Click on the  "Create a generation work product" button in the "*Items*" tab of the properties editor.

Note: The purpose of the generation work product is to define the generation options for the associated model element.

Creating a generation work product

The dialog box for a generation work product (figure 3-3) is used to specify the information linked to a generation. The dialog box's field values are described in chapter 3 of this user guide.

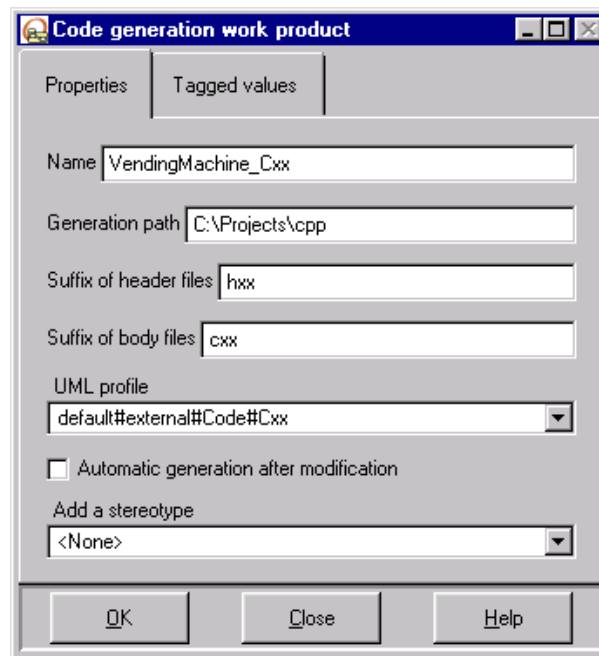


Figure 3-3. The "Generation work product" dialog box

After confirmation, generation is launched automatically if the "Automatic generation after modification" box is checked.

Note: The default values of this dialog box are parameterized at UML modeling project level (for further information, please refer to chapter 7, "Module parameters", of this user guide).

Generating code

We will now generate the code corresponding to the package in the explorer (figure 3-4).

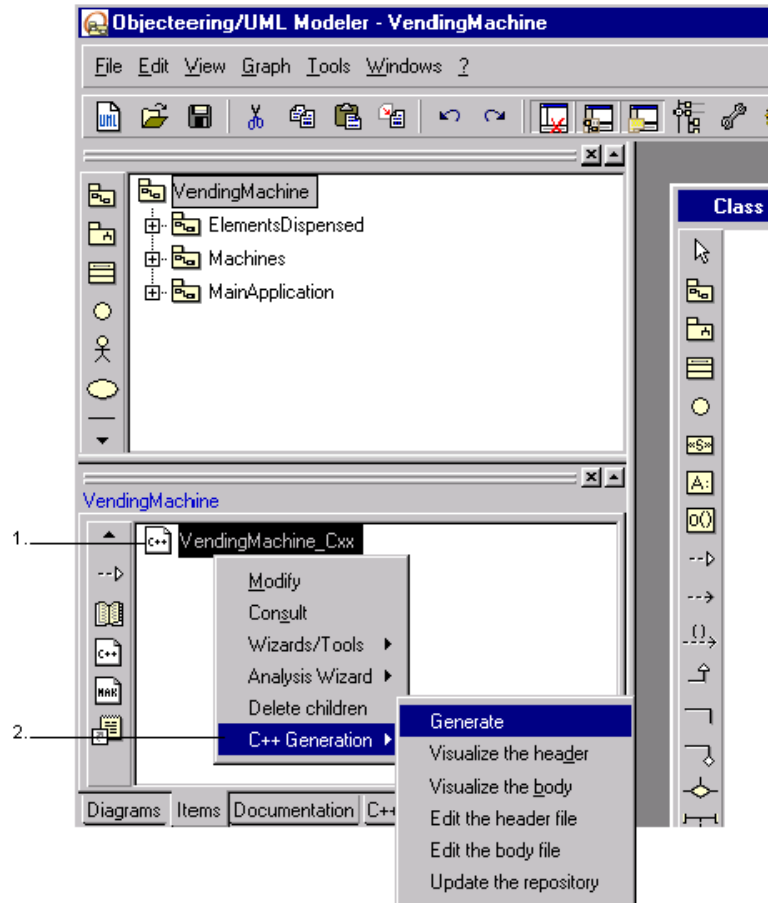


Figure 3-4. Generating code for the "VendingMachine" package

Chapter 3: First Steps

Steps:

- 1 - Click on the generation work product in the *"/items"* tab of the properties editor with the right mouse-button to display the context menu.
- 2 - Run the "C++ *Generation/Generate*" commands.

Note: Double-clicking on the generation work product executes the "*Generate*" command for this work product.

Visualizing the generated code

Visualizing the header of the generated code

Code has now been generated for the "VendingMachine" package and all its packages and classes. Visualizing the header of the generated code is possible from the explorer (figure 3-5).

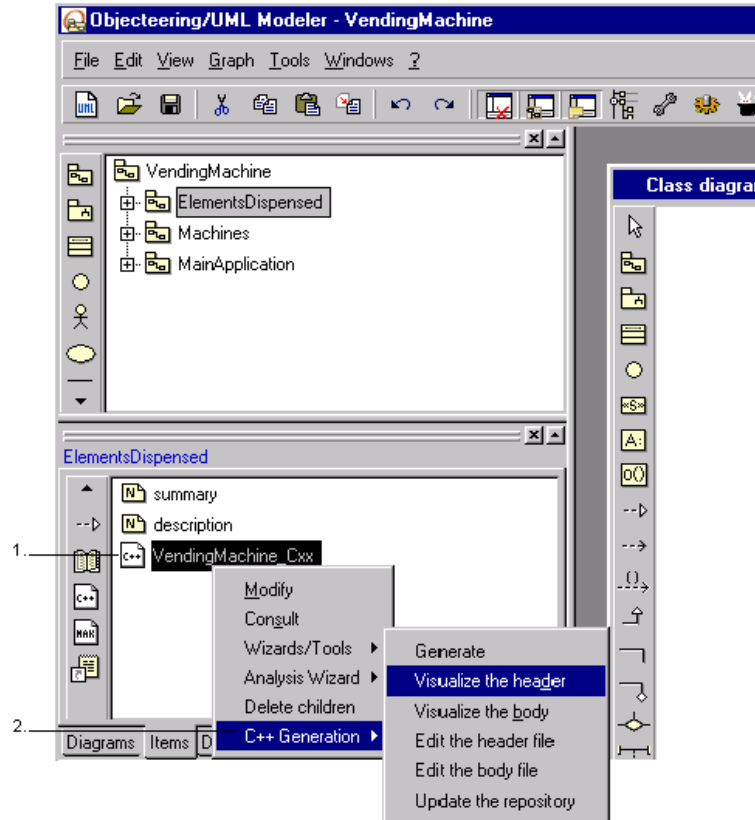


Figure 3-5. Visualizing the generated code *header* for the "ElementsDispensed" package

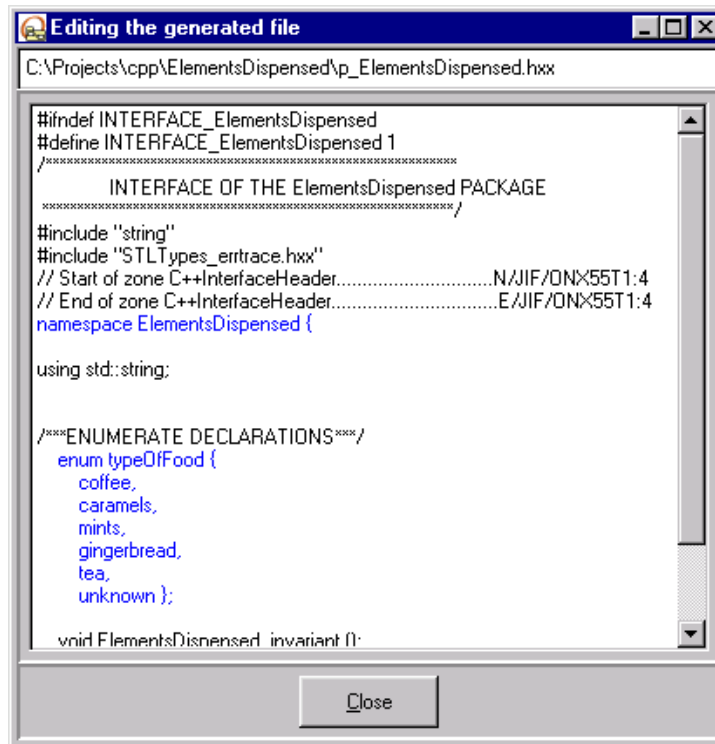
Chapter 3: First Steps

Steps:

- 1 - Select the "*ElementsDispensed*" package in the explorer and click on the associated generation work product in the "*Items*" tab of the properties editor with the right mouse-button to activate the context menu.
- 2 - Run the "C++ *Generation/Visualize the header*" commands.

Generated code header

The "Editing the generated file" window (figure 3-6) allows you to visualize the elements of the header's generated code.



```

C:\Projects\cpp\ElementsDispensed\p_ElementsDispensed.hxx

#ifndef INTERFACE_ElementsDispensed
#define INTERFACE_ElementsDispensed 1
/*****
INTERFACE OF THE ElementsDispensed PACKAGE
*****/
#include "string"
#include "STLTypes_ertrace.hxx"
// Start of zone C++InterfaceHeader.....N/JIF/DNX55T1:4
// End of zone C++InterfaceHeader.....E/JIF/DNX55T1:4
namespace ElementsDispensed {

using std::string;

/****ENUMERATE DECLARATIONS****/
enum typeOfFood {
    coffee,
    caramels,
    mints,
    gingerbread,
    tea,
    unknown };

void ElementsDispensed invariant ()

```

Figure 3-6. Generated code for the header for the "ElementsDispensed" package

Note: Double-clicking on the blue text opens a corresponding dialog box to modify the model. This corresponds to the Objectteering/UML mechanism of dynamic administration of the code/model traceability.

Visualizing the body of the generated code

The generated body file (figure 3-7) can be visualized from the explorer.

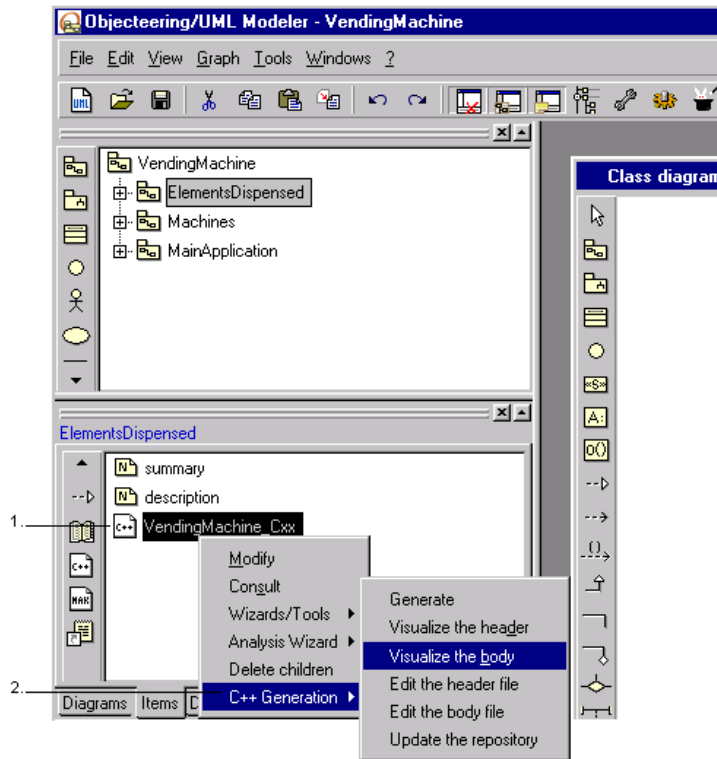


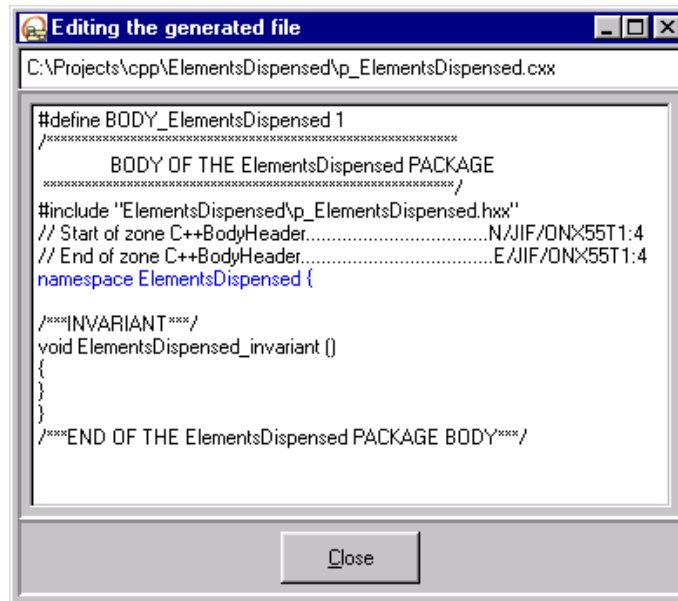
Figure 3-7. Visualizing the generated *body* code

Steps:

- 1 - Click on the generation work product in the "*Items*" tab of the properties editor with the right mouse-button to activate the context menu.
- 2 - Select the "C++ Generation/*Visualize the body*" options.

Body class

You can visualize the generated body code in the "Editing the generated file" window (figure 3-8).



```

C:\Projects\cpp\ElementsDispensed\p_ElementsDispensed.cxx

#define BODY_ElementsDispensed 1
/*****
      BODY OF THE ElementsDispensed PACKAGE
*****/
#include "ElementsDispensed\p_ElementsDispensed.hxx"
// Start of zone C++BodyHeader.....N/JIF/DN\x55T1:4
// End of zone C++BodyHeader.....E/JIF/DN\x55T1:4
namespace ElementsDispensed {

/****INVARIANT****/
void ElementsDispensed_invariant ()
{
}

/****END OF THE ElementsDispensed PACKAGE BODY****/

```

Figure 3-8. Generated body code for the "ElementsDispensed" package

Creating a compilation work product

Procedure

We will now create the compilation work product (figure 3-9) that contains all the information related to the compilation options for the previously generated code.

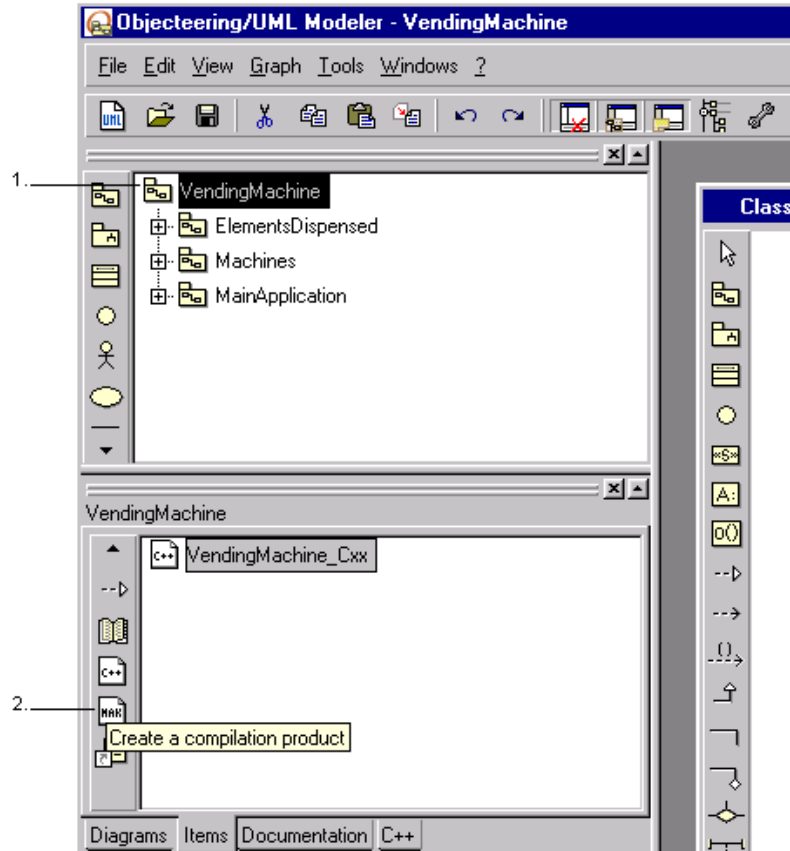



Figure 3-9. Generating a compilation work product for the "VendingMachine" package

Steps:

- 1 - Select the "VendingMachine" package in the explorer.
- 2 - Click on the  "Create a compilation work product" button in the "Items" tab of the properties editor.

Note 1: In order for compilation to be correctly carried out, certain C++ module parameters must already have been correctly defined, notably "Systems *include files*" (in the "Directories" group of parameters), "Libraries" (in the "Directories" group of parameters), and all the parameters in the "Attached tools" group, if the compiler used is not the default system compiler (MS Visual C++ for Windows, CC for Solaris and CC for HP). For further information on these module parameters, please refer to the "Detailed description of parameters" section in chapter 7 of this user guide.

Note 2: For Visual C++ for PC, include file and library path parameters are initialized by default, assuming that Visual C++ has been installed in C:\Program Files\Microsoft Visual Studio\Vc98.

Entering a compilation work product

The dialog box's field values are described in chapter 4 of this user guide.

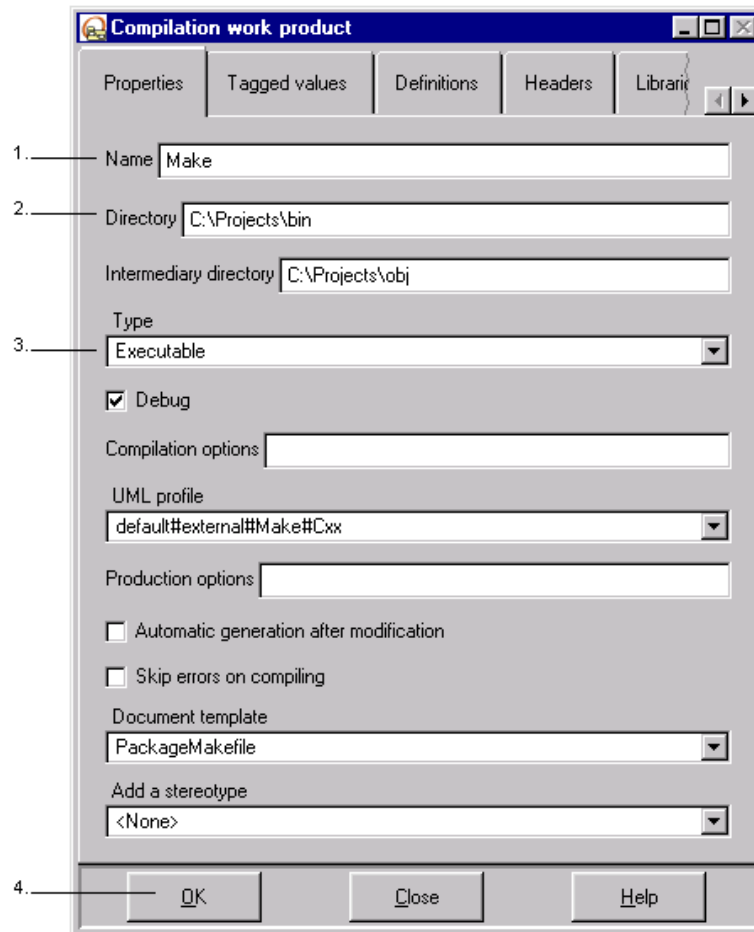


Figure 3-10. Entering a compilation work product

Steps:

- 1 - Enter the name of the compilation work product.
- 2 - Enter the directory where the makefile will be generated.
- 3 - Select the "*executable*" type.
- 4 - Confirm.

Note: After confirmation, the compilation work product is generated automatically.

Generating a makefile

Procedure

From the compilation work product, we are now going to generate the Makefile necessary to compile the package.

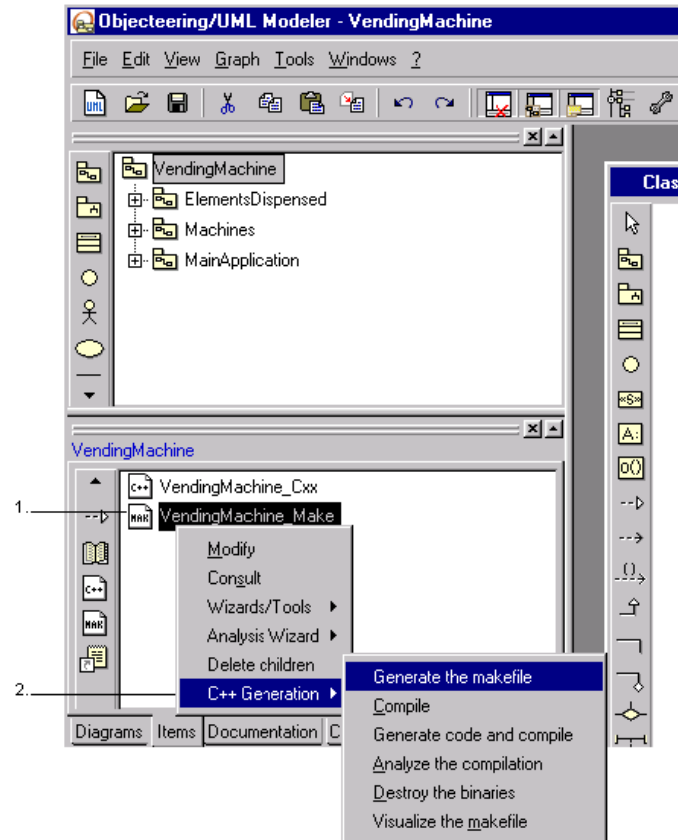


Figure 3-11. Generating a compilation work product for the "VendingMachine" package

Steps:

- 1 - Select the compilation work product in the "*Items*" tab of the properties editor with the right mouse-button to activate the context menu.
 - 2 - Run the "*C++ Generation/Generate the makefile*" commands.
-

Visualizing the makefile

Visualizing the Makefile

Visualizing the generated Makefile file header is possible from the explorer (figure 3-12).

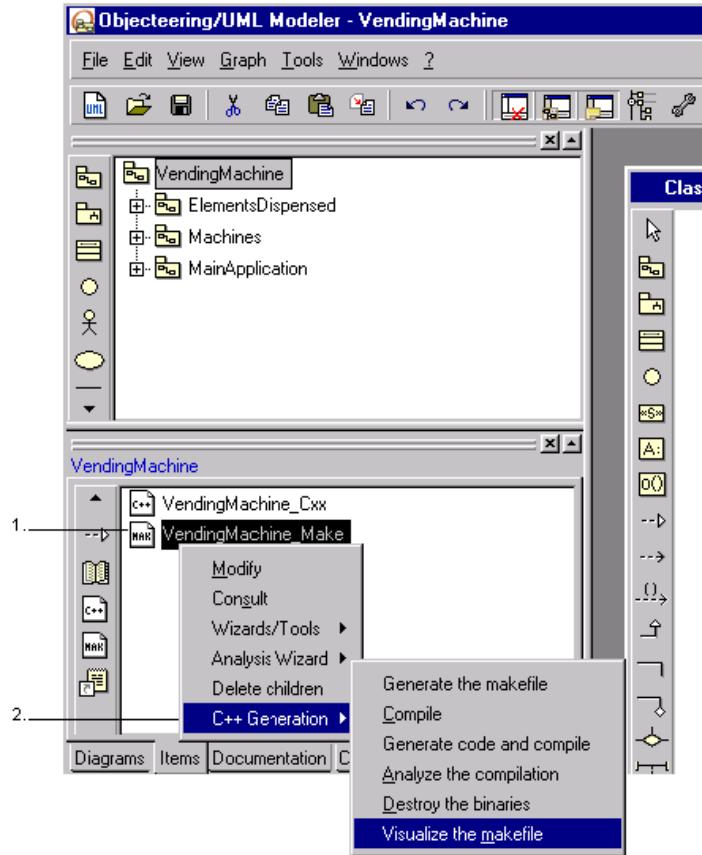


Figure 3-12. Visualizing the "VendingMachine_Make" makefile

Steps:

- 1 - Select the compilation work product in the "*Items*" tab of the properties editor with the right mouse-button to activate the context menu.
- 2 - Run the "*C++ Generation/Visualize the makefile*" commands.

The Makefile

The Makefile generated from the "VendingMachine" package is visible in the "Editing the generated file" window (figure 3-13).

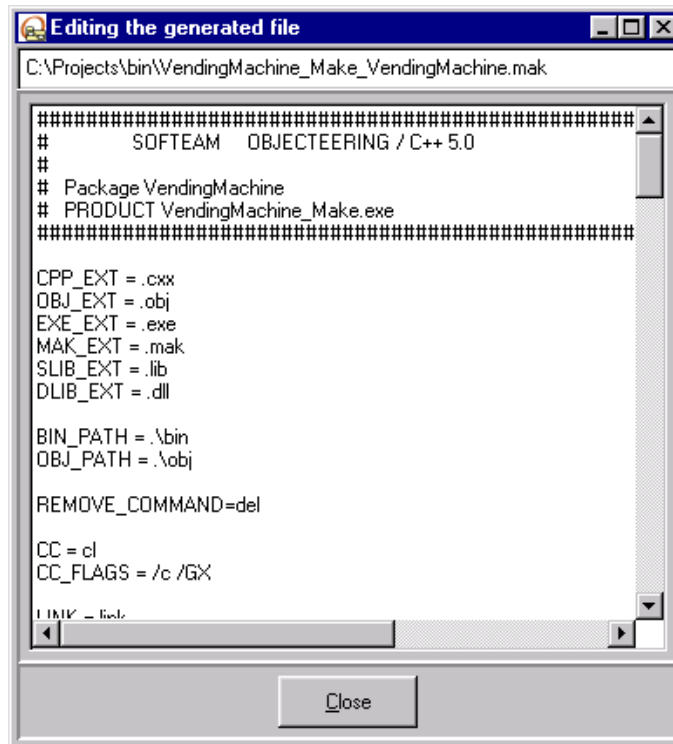


Figure 3-13. The Makefile of the "VendingMachine" package

Executing the makefile

Compiling the generated file

The compilation, or in other words, the command associated with the work product, will make it possible to produce the binary it specifies (i.e. the library, executable, etc.), and to analyze and correct the errors.

From the compilation work product, run the "*compile*" option. Objectteering/UML carries out the compilation automatically (execution of the Makefile), and opens a "*Compilation analysis*" window (Figure 3-14) containing 2 fields:

- ◆ a text field, in which you can visualize the text extracts containing possible errors
- ◆ a text field named "*List of errors*", that displays the errors and their gravity

Result

An executable binary is generated in the compilation work product directory, with the same name as the work product ("Application_mak" by default). Run the binary in this directory (DOS or UNIX window).

Compilation analysis

It is possible to analyze compilation errors directly from the tool. The "*Compilation analysis*" command opens a window containing the compilation results, as shown in Figure 3-14. In the case of an error during the compilation phase, the "*Compilation analysis*" window appears automatically.

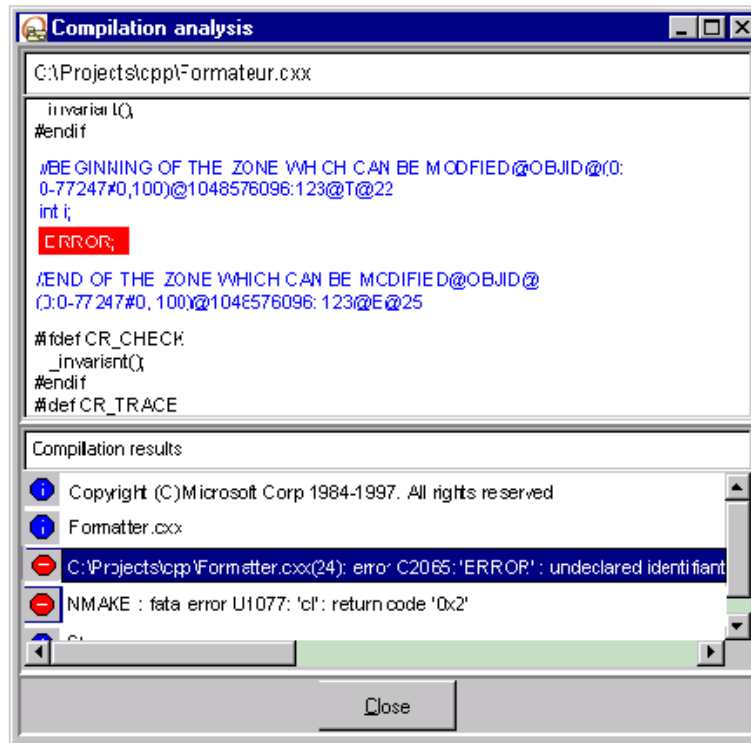




Figure 3-14. The "*Compilation analysis*" window displaying the error that must be corrected

This window is divided into two sections:

- 1 - The generated sources which present the errors. If you double-click on the blue text, a dialog box for the generation elements, used to generate this zone, will open.
- 2 - The lower section called "*Compilation results*" displays the errors prefixed by the  icon. The other lines are prefixed by the  icon. If you select a line of errors, the file containing the incorrect code will be displayed in the upper section of the window. If you double-click on this code, the model element which generated it will be displayed.

To correct the error, simply double-click on the error mentioned in red in the text field, and correct the inserted error in the dialog box.

Chapter 4: Compilation and generation
work products

The two work products in the Objecteering/C++ module

Overview

To obtain a binary from a model, you must create a generation work product for the highest level package, generate C++ code and then create a compilation work product. If you want to generate more than one executable from your model (for example, a client executable and a server executable), you have to organize your classes into two major packages. Work products should then be created for each package, with specific options for the client side and the server side.

The generation work product represents the C++ sources. The compilation work product represents the production process (*Makefile*) and the results of the compilation (libraries, error files, executables).

These two work products follow the UML modeling project's composition structure (UML modeling project/package/class logic). If you create a work product on a package during the first generation, a work product will be created on each component (sub-package or class), which can have such a work product. If you add components after the last generation, the following generation will create work products on the new components, which can have the work products. In this way, these work products will maintain consistency with the model.

Defining a generation work product



A pair of C++ sources (*interface* and *body*) are associated with each package or class.

A generation work product can:

- ◆ group together certain generation attributes (path and extensions of the C++ files for generation, generation UML profiles...)
- ◆ run C++ code generation
- ◆ maintain consistency between C++ files generated and the model
- ◆ offer the file services related to the model element

Defining a compilation work product



A file (makefile) is associated to each package.

A compilation work product can:

- ◆ group together compilation attributes, used to constitute the "*makefile*"
 - ◆ run the compilation and produce an executable and a static or a dynamic library
 - ◆ maintain consistency between the makefiles and the model
 - ◆ provide file services related to makefile and compilation error analysis
-

The attributes of a generation work product

Description

The dialog box for modifying a generation work product is shown in Figure 4-1.

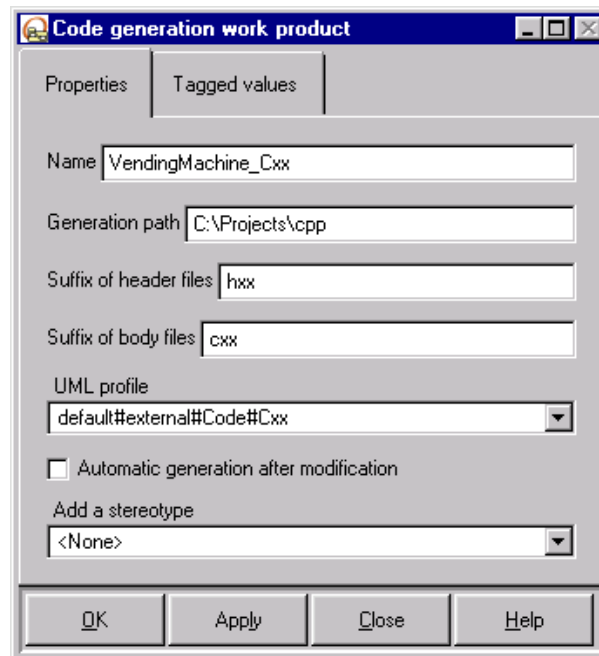


Figure 4-1. The "Generation work product" dialog box

The attributes are defined in the table below:

The ... field	is used to specify ...
Name	the name of the generation work product. By default, this name is "Cxx". The names of the generation work products of sub-packages and classes, created during propagation, are the same as the name of the original work product.
Generation path	the location of the files created by the generation. If you want to separate the generated header and body files into two different directories, you can specify paths for them here. The first is the path for header files, and the second is the path for body files. They are separated by ";".
Suffix of header files suffix	the header file extension.
Suffix of body files	the body file extension.
UML profile	the generation UML profile. It allows you to distinguish several possible generation parameterizations.
Automatic generation after modification	whether to run the generation immediately after clicking on the OK button.
Add a stereotype	the semantics of a given existing class. Stereotypes are defined at UML profiling project level.

Note: Only one C++ generation work product can be created per model element.

Here the "*Name*" field provides the name of the generation work product. This name is not the same as the name of those associated files. The name of a generated code file is always the same as that of the model element (for a class) followed by the extension, header file suffix and body file suffix in that order.

For a package, the name of the generated file is equal to the model element name prefix with the string "p_". This component is useful to avoid collision file names in the case of a package and one of its classes having the same name.

The "*Automatic generation after modification*" box is used to indicate whether to run generation automatically after the work product's parameters have been modified.

Note: *Generation path*: If you want to separate the generated header and body files into 2 different directories, you can indicate their paths here. The first one is the path for header files, the second for body files. They are separated by ";".

The attributes of a compilation work product

Overview

The attributes of a compilation work product are classified into six groups:

- ◆ Properties
- ◆ Tagged values
- ◆ Definition
- ◆ Headers
- ◆ Libraries to be linked
- ◆ Excluded classes

The "Properties" tab

This tab defines essential values and options.

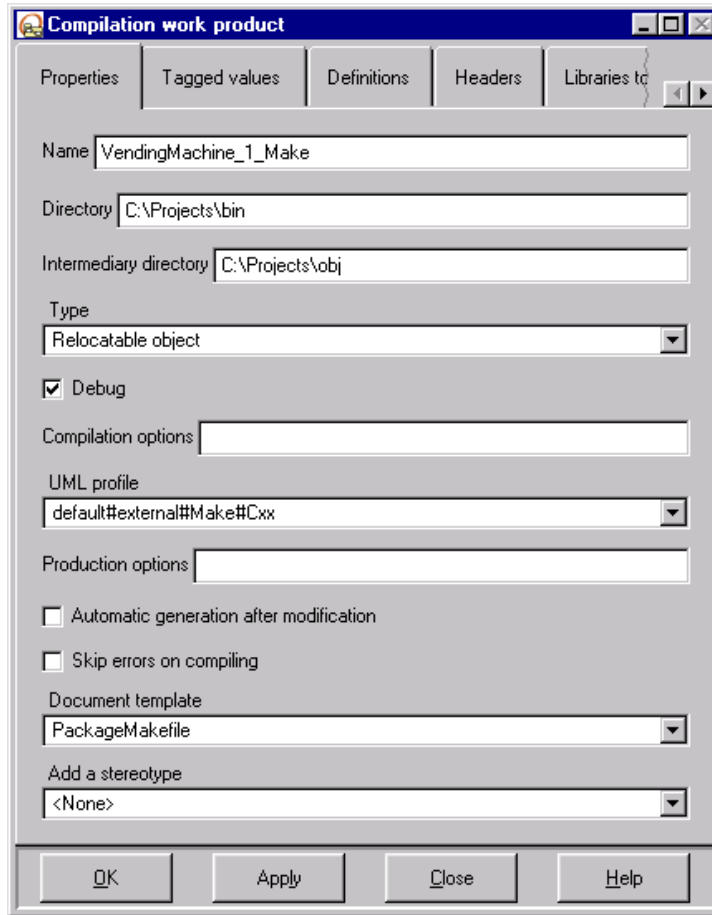


Figure 4-2. The "Properties" tab of the compilation work product dialog box

The ... field	is used to specify ...
Name	the name of the compilation work product. By default, this name is "Make". The compilation work products belonging to sub-packages and classes all have the same name as the original compilation work product.
Directory	the location of the targeted files and makefile.
Intermediary directory	the location of the intermediary files, such as the object files.
Type	the type of file targeted, i.e., object, and static or dynamic library.
Debug	whether or not to produce debug symbols.
Compilation options	compiling options.
UML Profile	the Makefile generation profile. This is used to select the Makefile generation parameterization.
Production options	the options for editing links.
Automatic generation after modification	whether or not to regenerate the Makefile automatically just after the work product has been modified.
Skip errors on compiling	whether or not to continue generating, in spite of errors.
Document template	the document template which will generate a Makefile for a package.
Add a stereotype	the semantics of a given existing class. Stereotypes are defined at UML profiling project level.

Note: A model element can have several C++ compilation work products (for example, one for debugging, one for releases, and so on. If this is the case, each compilation work product must have a different name.

The name of the Makefile generated takes the following form: WorkProductName_WorkProductPackageName.mak.

The "Definition" tab

This tab is used to provide definitions in the makefile's line of compilation. It corresponds to the /D option for some C++ compilers.

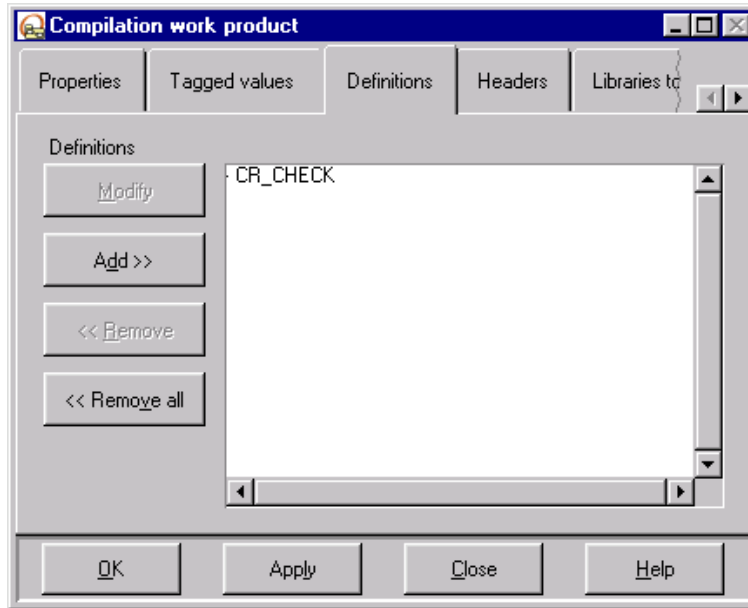


Figure 4-3. The "Definition" tab of the compilation work product dialog box

The ... button	is used to ...
Modify	modify the selected variable.
Add	add a variable.
Remove	remove the selected variable.
Remove all	remove all the variables.

The "Header" tab

This tab is used to define the paths used when searching for header files during compilation. They correspond to the `-I` option for some C++ compilers.

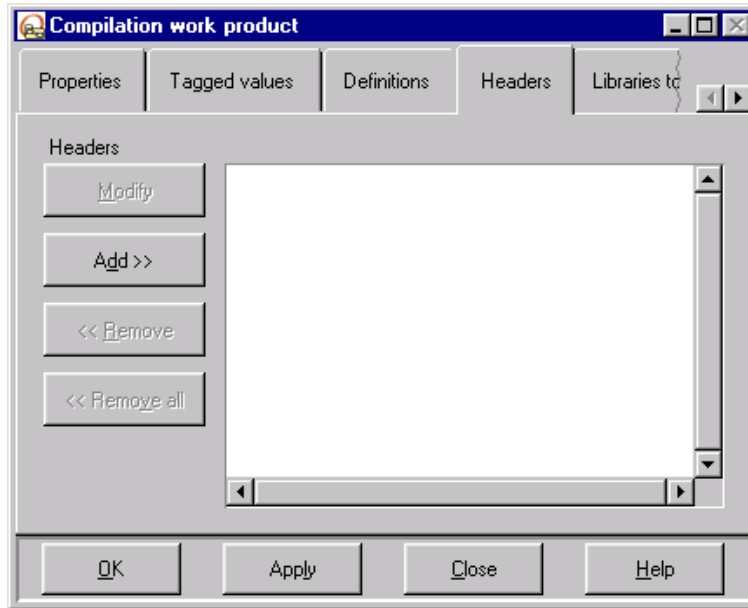


Figure 4-4. The "Headers" tab of the compilation work product dialog box

The ... button	is used to ...
Modify	modify the selected path.
Add	add a path used when searching for header files.
Remove	remove the selected path.
Remove all	remove all the paths for searching for the header files.

The "Libraries to be linked" tab

This tab is used to add the libraries to be linked, as well as the paths used to search for the libraries in the Makefile.

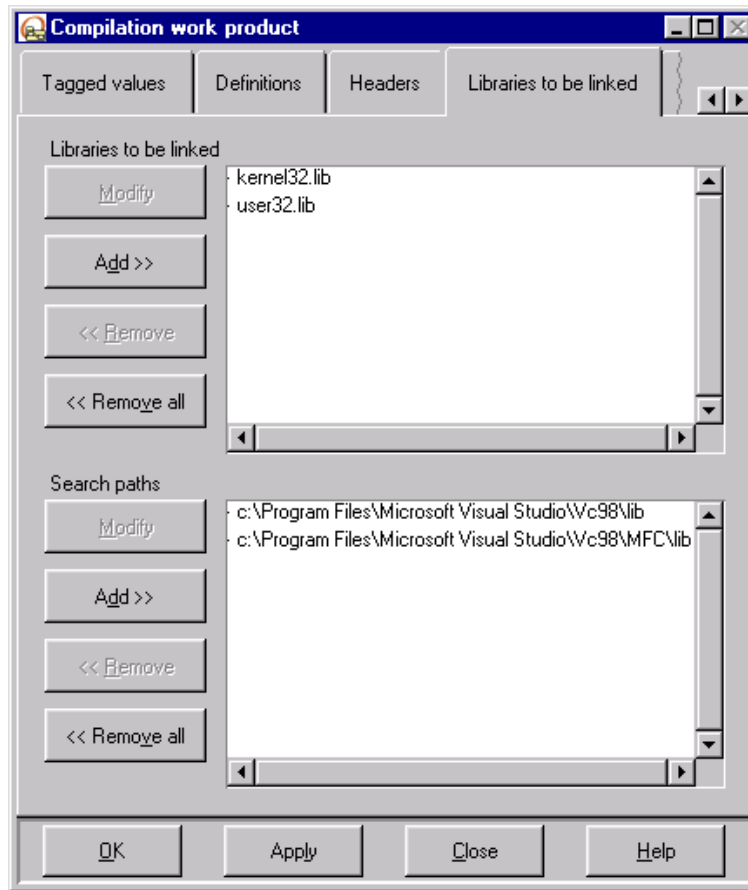


Figure 4-5. The "Libraries" tab of the compilation work product dialog box

Libraries that must be linked:

The ... button	is used to ...
Modify	modify the selected library.
Add	add a library.
Remove	remove the selected library.
Remove all	remove all the libraries.

Search paths:

The ... button	is used to ...
Modify	modify the selected path used to search for the libraries.
Add	add a path used to search for the libraries.
Remove	remove the selected path used to search for the libraries.
Remove all	remove all the paths used to search for the libraries.

Note: The format of the entered strings depends on the system on which the compilation is run.

The two lists in the above window are initialized from module parameters when the compilation work product is created. If module parameters are modified after the creation of the work product, these modifications will not be taken into account by the work product.

Chapter 4: Compilation and generation work products

In UNIX, the library format is as follows:

The ... string with the format	produces the inclusion of the ... library
my_lib	libmy_lib.a (static) or libmy_lib.so (dynamic)
-lmy_lib	libmy_lib.a (static) or libmy_lib.so (dynamic)
my_lib.a	my_lib.a
my_lib.so	my_lib.so

In Windows, the file format is as follows:

The ... string with the format	produces the inclusion of the ... library
my_lib	my_lib.lib
my_lib.lib	my_lib.lib
my_lib.dll	my_lib.dll

The "Excluded classes" tab

This tab allows you to exclude classes from the Makefile, i.e. the classes which are not taken into account during compilation.

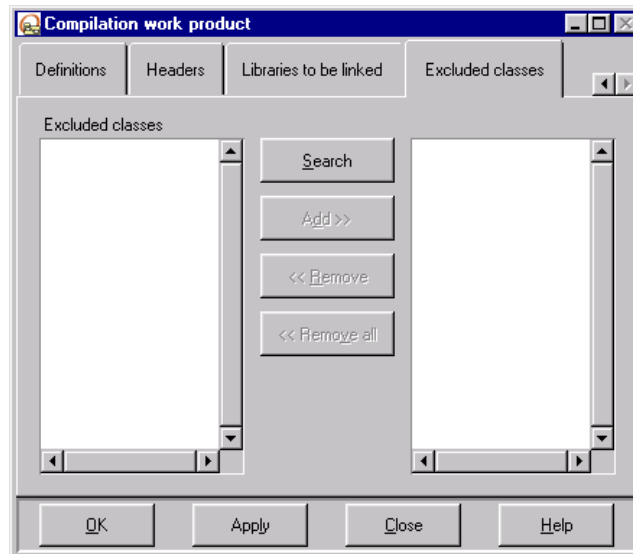


Figure 4-6. The "Excluded classes" tab of the compilation work product dialog

The ... button	is used to ...
Search	search for the classes that can be excluded.
Add	add the selected class(es) to the right-hand side (classes to be excluded).
Remove	remove the selected classes from the right-hand side.
Remove all	remove all the classes from the right-hand list.

The context menus of generation and compilation work products

The context menu of a generation work product

Selecting a work product (generation or compilation), by clicking on the right button, opens a pop-up menu called a context menu (Figure 4-7).

The context menu of a generation work product

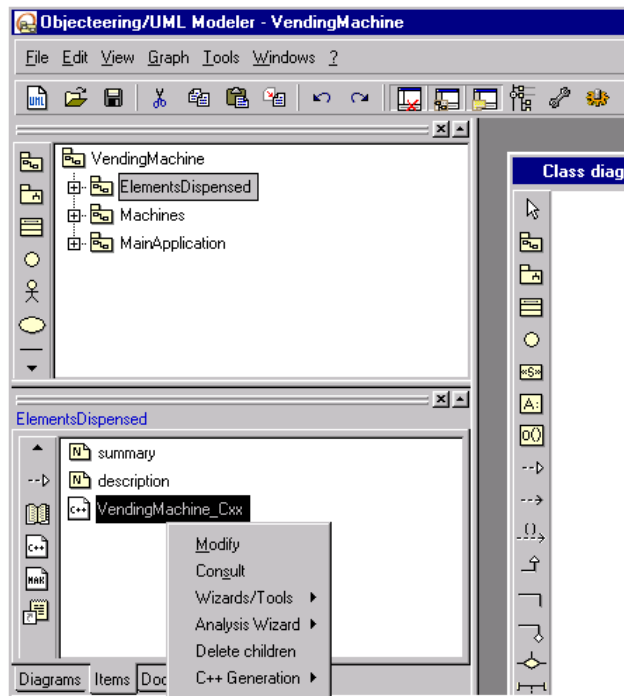


Figure 4-7. Context menu of a generation work product

The ... command	is used to ...
Modify	modify the generation work product attributes.
Consult	consult the generation work product attributes.
Delete children	suppress the child work products.
C++ generation	display the generation sub-menu.

The context menu of a compilation work product

Selecting a compilation work product, by clicking on the right mouse-button, opens a pop-up menu called a context menu (Figure 4-8).

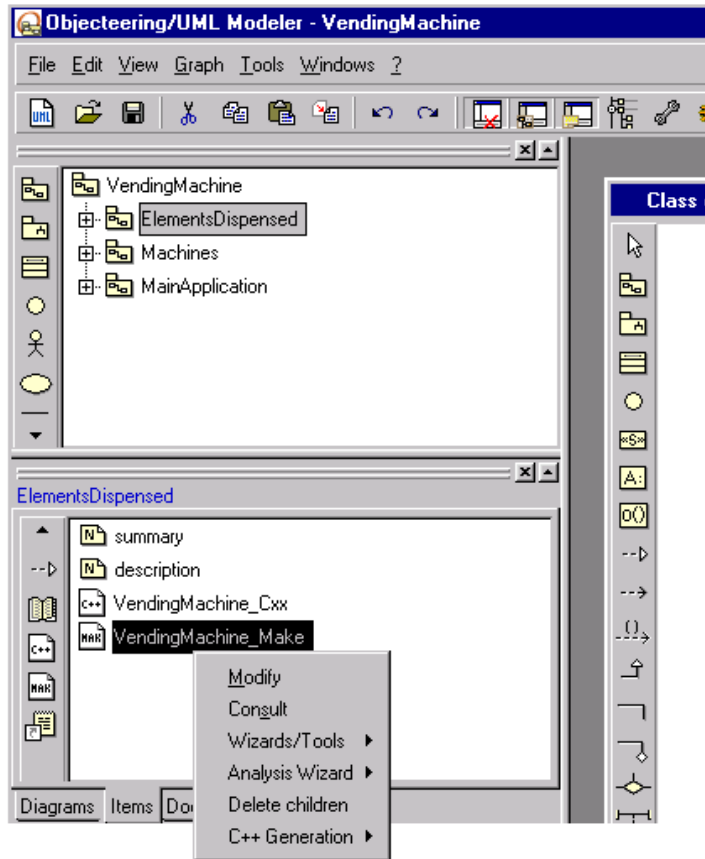


Figure 4-8. Context menu of a compilation work product

The ... command	is used to ...
Modify	modify the compilation work product attributes.
Consult	consult the compilation work product attributes.
Delete children	suppress the child work products.
C++ generation	display the generation sub-menu.

The available functions in the context menu of a work product can be classified into three groups: attribute management, propagation action and file related services.

Managing a work product's attributes

Each work product has a set of attributes related to its function. Both context menus of the products contain a pair of menu items, "*Modify*" and "*Consult*".

The attributes of each work product are described in the following sections of this chapter, and can be modified or consulted by the user.

When creating the generation work product and compilation work product, the related dialog boxes provide certain default values. These default values can be modified by changing the parameters of the module. For a detailed description, please refer to chapter 7, "*Module parameters*", of this user guide.

The action of propagation

A UML modeling project is organized in a structure of packages and classes. For certain product functions, if processing is run from an element, the action is also carried out on all its component elements. This function is called *propagation*.

For a generation work product, propagation actions are "*Generate*" and "*Update the repository*".

For a compilation work product, propagation actions are "*Compile*", "*Generate the makefile*", "*Generate the code and compile*" and "*Delete the binaries*".

The "*Delete child work products*" command, available on both work products, successively deletes those work products created by a propagation action.

Features related to associated files

Work products provide features related to the files associated with a work product.

The ...	provides the ... services
generation work product	visualize (body or header) edit (body or header)
compilation work product	visualize the " <i>makefile</i> "

Generating C++

Code generation

You can run C++ code generation by selecting the generation work product context menu, or simply by double clicking on the generation work product.

Code generation is a propagation action. If the generation work product of the sub-elements does not exist, it is created first.

Note: By properly organizing your model into packages, code generation on all parts of the model can be carried out in one go.

Code generation and model consistency checks

Code may be generated from the UML model regardless of whether consistency checks are active or inactive. However, when generation is launched, a message informs the user that he is in the process of generating code on a model which may potentially not conform to the UML modeling rules checked by Objectteering/UML (as shown in Figure 4-9).

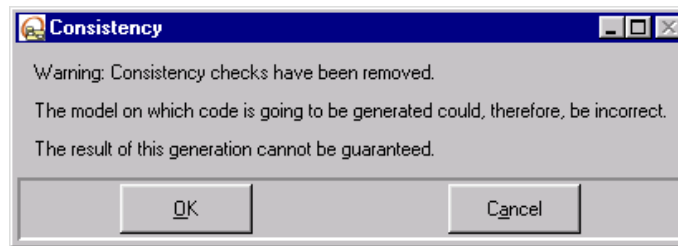


Figure 4-9. Message informing the user that consistency checks have been removed

Note: It should be noted that code generation in command line mode (please see objingcl) is assured, whatever the state of the consistency checks at the time of code generation.

Generating the makefile

A makefile is only generated for a package. This is a propagation action.

Editing the code file

External editor

Selecting the item *edit (the header or body)* from the context menu of the generation work product opens an editor called external editor.

In the external editor, code can be freely entered, but this code must always be between the following two markers:

```
// BEGINNING OF ZONE WHICH CAN BE MODIFIED @OBJID@...  
// END OF ZONE WHICH CAN BE MODIFIED @OBJID@...
```

Any modification made outside these zones will not be retrieved by the *Objectteering/C++* module.

Chapter 4: Compilation and generation work products

The zones you can modify correspond to the C++ notes presented in the table below.

The ... note	on the ... file	concerning the ... class	is retrieved ...
C++BodyHeader	body	Class	at the beginning of the class body.
C++InterfaceHeader	header	Class	at the beginning of the class header.
C++PrivateMember	header	Class	as free declarations with private visibility.
C++ProtectedMember	header	Class	as free declarations with protected visibility.
C++PublicMember	header	Class	as free declarations with public visibility.
C++	body or header	Operation	as method body.
C++ConstructorTransmission	body or header	Operation (constructor)	as instructions for passing information between constructors.
C++Returned	body or header	Operation	as a method's return value.
C++BodyHeader	body	Package	at the beginning of a package body.
C++InterfaceHeader	header	Package	at the beginning of a package header.

Retrieving edited code

Modifications made to a generated file are retrieved when the external editor closes. The file is scanned to find concerned zones within a pair of markers. For each zone that is not empty, Objecteering/UML checks the model. It automatically creates the C++ notes corresponding to the said zones, if they do not already exist. The code in the zones is then copied into the content of the notes.

Note: Retrieving does not control or compare C++ syntax. The code of the edited file is, therefore, simply retrieved and placed in the corresponding C++ note.

In this way it is possible, while editing the generated files, to create the C++ notes automatically in the model, by entering code into empty notes (between markers).

Updating the repository

When should updating be launched?

Objectteering/UML can retrieve the modifications made in the generated files, without going through the external editor in Objectteering/UML. In this case, it is necessary to update the repository.

The generation work product keeps track of your work. Each time you generate code for an element, the generation work product checks the registered time of the code file. If the file is more recent than the last generation, the update is run before generating the new code.

Updating the repository is an action of propagation.

What happens during updating?

During the retrieval, each file is scanned to collect the code zones between the pair of markers as explained in the previous section. The generation work product identifies the corresponding C++ note in the model.

- ◆ if the zone is not empty, but no C++ note in the model corresponds to this zone, a C++ note is created containing the code in the zone
- ◆ if the code in the zone differs from the content of the corresponding C++ note in the model, generation history is reviewed

If...	then...
generation occurred after this model element was created or modified	the content of the C++ note is updated with the zone's code.
the C++ note is created or modified after the last generation	a dialog box comparing the C++ note and the code of the zone opens , and the user decides which text version should be kept.

The comparison window

When both the file and the text in the repository, associated with a modified zone in the file, are more recent than the last code generation, a comparison window will open.

The upper section presents the text as it is in the model. The lower section contains the text zone corresponding to the file.

The user then chooses one of the following actions:

- ◆ Retrieve: the code presented in the lower section is placed into the repository and replaces the code presented in the upper section.
- ◆ Keep: the code presented in the upper section is kept.

In both cases the window will then close. If another similar case arises during the rest of the retrieval, the window will open again.

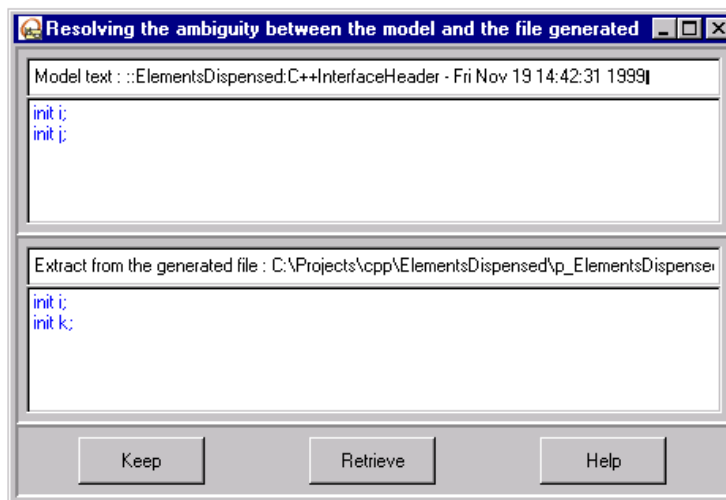


Figure 4-10. The window used to resolve differences between the model and the generated file

Analyzing the compilation

Overview

Selecting the "Compilation Analysis" item from the compilation work product's context menu will open the window used to analyze compilation. The window is divided into two parts. The upper section contains the code file, the lower section contains the result of the compilation (Figure 4-11).

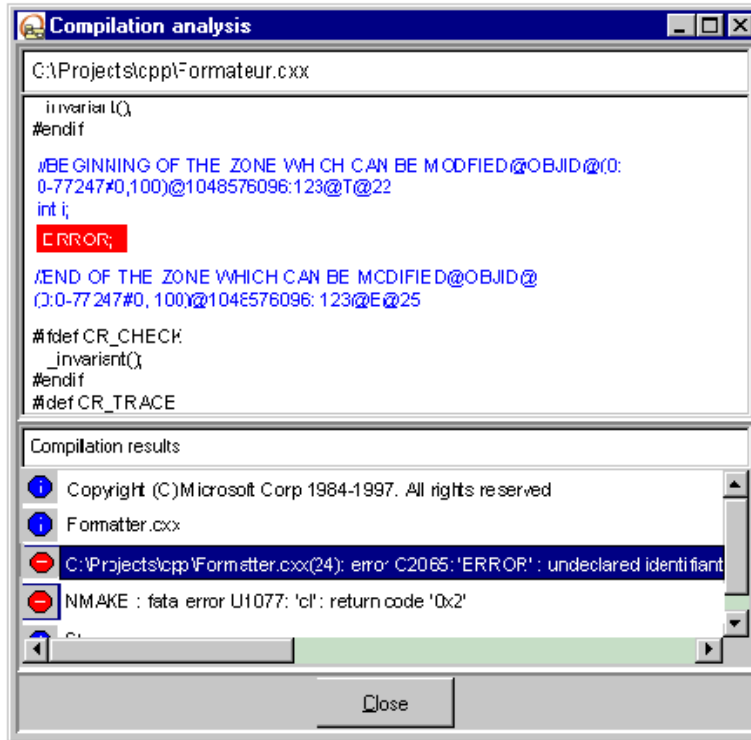





Figure 4-11. The "Compilation analysis" window

Chapter 4: Compilation and generation work products

The result of the compilation contains three different message categories: *error*, *warning* and *information*.

They are represented respectively by the ,  and  icons.

The code file is displayed in blue or black. The blue zone corresponds to a C++ note or a model element. A line of code may, however, be in red. This means that this line of code may be the one containing the error selected in the lower part of the window.

Correcting errors

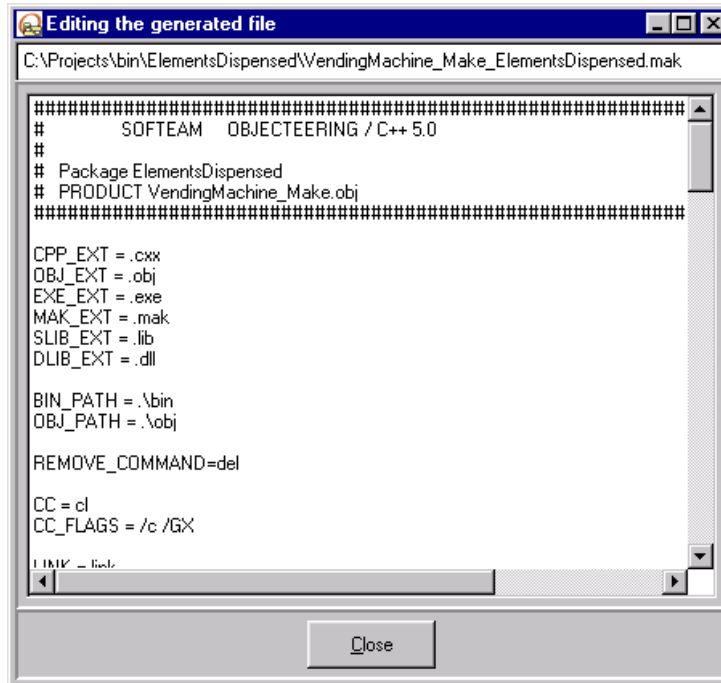
This window makes it possible to correct the compilation error by modifying the model element directly. Double-clicking on the blue zone in the upper section of the window opens a C++ note edition window or a model element definition dialog box, in which you can carry out the required modifications. You may then run the "*Generate and compile*" command from the compilation work product context menu. This development cycle eventually produces the final target file that will be consistent with the model.

Visualizing the file

Overview

When the following commands are run, the generated C++ or makefile files are displayed in read only mode:

- ◆ "Visualize the makefile" for a compilation work product
- ◆ "Visualize the header" for a generation work product
- ◆ "Visualize the body" for a generation work product



The screenshot shows a window titled "Editing the generated file" with a standard Windows-style title bar (minimize, maximize, close buttons). The window's content area displays a Makefile for a project named "WendingMachine_Make_ElementsDispensed". The file path is shown as "C:\Projects\bin\ElementsDispensed\WendingMachine_Make_ElementsDispensed.mak". The Makefile content includes a header section with "SDFTEAM OBJECTEERING / C++ 5.0", package and product names, and various compiler and linker settings. A "Close" button is located at the bottom center of the window.

```
#####  
#      SDFTEAM  OBJECTEERING / C++ 5.0  
#  
# Package ElementsDispensed  
# PRODUCT WendingMachine_Make.obj  
#####  
  
CPP_EXT = .cxx  
OBJ_EXT = .obj  
EXE_EXT = .exe  
MAK_EXT = .mak  
SLIB_EXT = .lib  
DLIB_EXT = .dll  
  
BIN_PATH = .\bin  
OBJ_PATH = .\obj  
  
REMOVE_COMMAND=del  
  
CC = cl  
CC_FLAGS = /c /GX  
  
LINK = link
```

Figure 4-12. Window for visualizing the "Makefile" file

Chapter 5: Tagged values and notes
specific to C++

Overview of tagged values and notes specific to C++

Objectteering/UML is a multi-target workshop used to model a large quantity of application elements, whatever the computing language used.

However, during the technical designing and programming phases, implementation details expressed in target language have to be specified and added to the model, such as the body methods or the exact type of method parameters.

This information is entered either with:

- ◆ "*Tagged values*" (also called "*directives*"), which provide implementation rules for the generator
- ◆ "*Notes*", which correspond to the zones inserted directly in the generated code
- ◆ "*Stereotypes*", which provide implementation rules for the generator, different to those of a non "*stereotyped*" element
- ◆ "*Constraints*", which provide the code inserted in pre-condition and post-condition zones


Tagged values and notes specific to C++ can be created for a model element only if the *Objectteering/C++ module* has been selected.

Tagged values

Creating a tagged value

Tagged values are used to express very specific C++ properties (for example the *"inline"* method, *"virtual"* method, *"const"* attribute, etc.).

They can be added to model elements:

- ◆ in the element's dialog box, by selecting the *"Tagged values"* tab (Figure 4-1).
- ◆ for elements appearing in the explorer, by clicking on the  *"Associate a tagged value"* icon in the *"Items"* tab of the properties editor.

In figure 5-1 for example, the `{virtual}` tagged value is added to an operation, to express that it is a C++ virtual method.

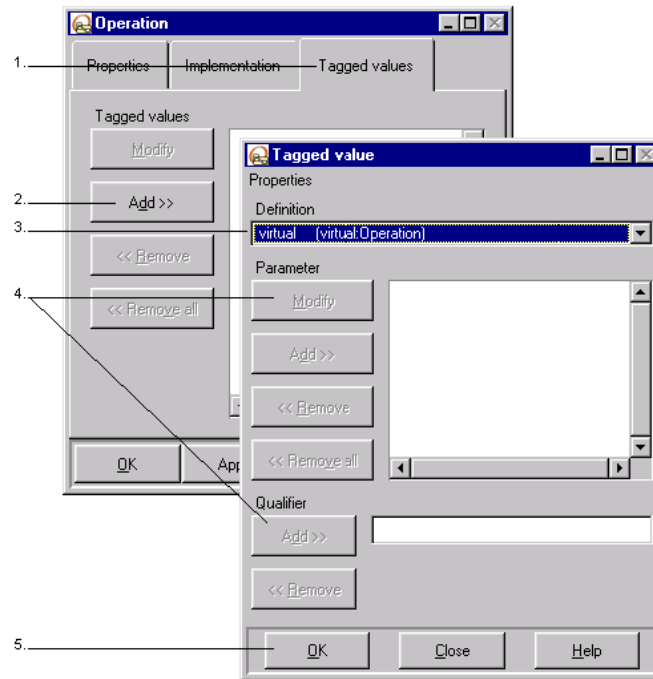


Figure 5-1. Entering a `{virtual}` tagged value on an operation

Steps:

- 1 - After having opened the "Operation" dialog box, select the "Tagged values" tab.
- 2 - Click on "Add".
- 3 - Select the `{virtual}` tagged value.
- 4 - Add parameters and qualifiers where necessary.
- 5 - Confirm.

Entering a tagged value

A tagged value is characterized by:

- ◆ its type (or specification), chosen from the list in the dialog box's "*Properties*" section
- ◆ possible parameters, entered in the dialog box's "*Parameters*" section

Note: The "*Qualifier*" section of this dialog box serves no purpose for a C++ tagged value.


Information on the full set of tagged values provided by the *Objectteering/C++* module can be found in the chapters on code generation, contained further on in this user guide.

Notes

Usual case

C++ code added by hand is entered via specific C++ notes. These notes can be entered in C++ external editors, directly in the "Items" tab of the properties editor or in the model element's entry dialog box.

C++ notes can be added to:

- ◆ all elements which can have notes, via the element's dialog box, by selecting the "Notes" tab
- ◆ those elements which appear in the explorer, by clicking on the  "Add a note" icon in the "Items" tab of the properties editor and thus creating a new note

Note: To add a C++ note to an operation, select the "Implementation" tab in the element's dialog box.

Chapter 5: Tagged values and notes specific to C++

For example, specific *"includes"*, headers of a C++ class body, will be entered in the *"C++BodyHeader"* text zone (Figure 5-2) on a class.

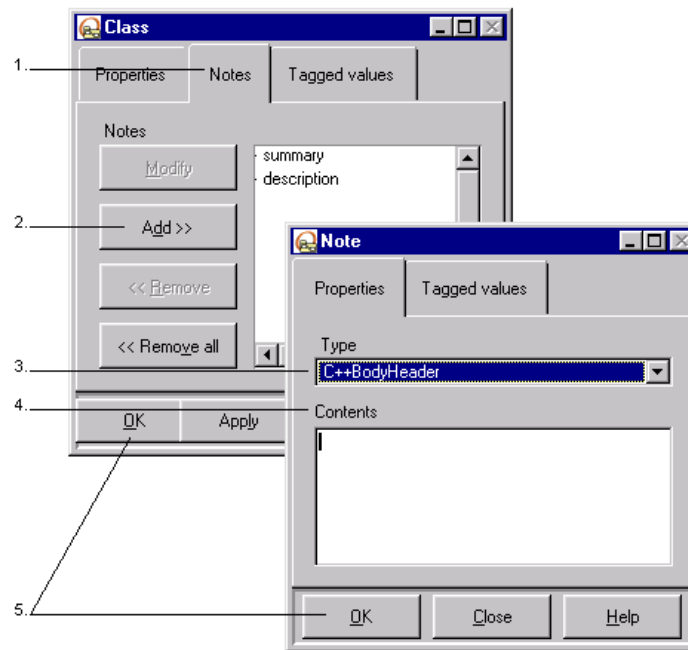


Figure 5-2. Entering a C++ note for a class

Steps:

- 1 - Select the "Notes" tab.
- 2 - Click on "Add".
- 3 - Select the "C++BodyHeader" type.
- 4 - Enter the code.
- 5 - Confirm.

Entering a C++ note

A note is characterized by:

- ◆ its type, which determines the meaning of the text for the C++ language. This is chosen only from the suggested list.
- ◆ its content, which consists of information that will be inserted into the code, at the location determined in the generator, depending on the type of note. This information must be entered in the corresponding text field.

Operation implementation

The implementation of an operation is defined with a C++ note. This note is created by selecting the "*Implementation*" tab of the "*Operation*" dialog box, or by directly creating a note in the explorer.

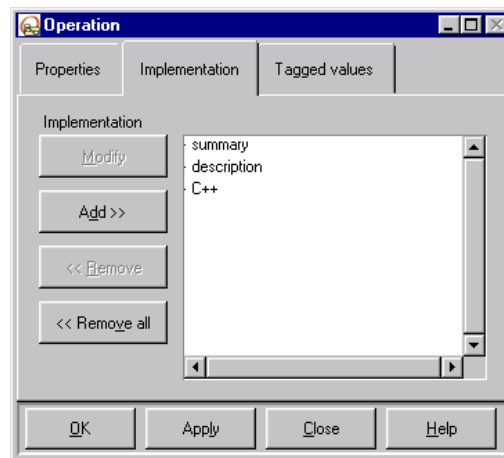



Figure 5-3. The "*Implementation*" tab of the "*Operation*" dialog box

The note called "C++" for an operation contains its implementation. The note named "*C++Returned*" contains the expression of a returned value. The note entitled "*C++ConstructorTransmission*" provides construction parameter values for the parent class.

Operation pre-conditions and post-conditions can be entered in the explorer by creating a constraint  stereotyped "*C++ precondition*" or "*C++ postcondition*". The same process is used to create "*invariant*".

Information on the full set of notes provided by the *Objectteering/C++* module can be found in the chapters on code generation, contained further on in this user guide.

Chapter 6: Calling module commands
in batch mode

Calling the module's commands in batch mode - Overview

Presentation

This chapter only applies to the *Objectteering/Enterprise Edition* version.

Module commands which do not necessitate a graphic interface can be run on line command using the *objincl* executable delivered with Objectteering/UML.

This "*command on line*" mode is very useful for, for example, regenerating a whole application as a background task, or for carrying out other administrative actions.

Calling commands

Syntax

An online command is called with the following type of instruction:

```
objingcl -db <database>  
-prj <project_name>  
-mdl <module_name>  
-cmd <command_name>  
<metaclass>:<object_name>
```

The name of the module for the C++ code generation and the constitution of the production process is *CxxModule*.

Invocable commands

Command ...	Metaclass ..	Action ...
generate	MpcCodeCxx	C++ code generation
generateMakefile	MpcMakeCxx	generates the Makefile files
compile	MpcMakeCxx	compiles the generated sources
deleteBinaries	MpcMakeCxx	destroys the binaries

Example

We want to generate C++ code for the "P1" package in the "my_project" UML modeling project, contained in the "my_database" database.

This code to be generated corresponds to the C++ generation work product.

The command line is as follows:

```
objingcl -db my_database -prj my_project -mdl CxxModule
-cmd generate MpcCodeCxx::my_project::P1::Sources
```

If the name of the project is the same as the name of the database, the following command line is used:

```
objingcl -db my_database -mdl CxxModule -cmd generate
MpcCodeCxx::my_database::P1::Sources
```


For a class, C1, belonging to the P1 package, the following command line is used:

```
objingcl -db my_database -mdl CxxModule -cmd generate
MpcCodeCxx::my_database::P1::C1::Sources
```

Chapter 7: Module parameters

Definition of module parameters

Introduction

You can parameterize the generated C++ code and makefile, by changing the values in the "Edit configuration". By clicking on the  "Modify module parameter configuration" icon in Objectteering/UML's toolbar, the "Edit configuration" window, as shown in Figure 7-1, is displayed.

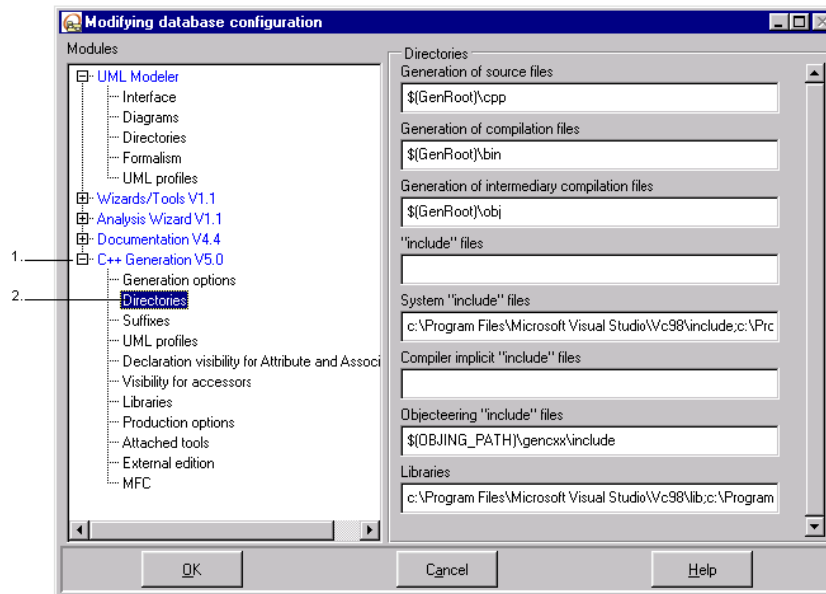


Figure 7-1. Editing the configuration of the *Objectteering/C++* module

Steps:

- 1 - Click on the "C++ Generation" sub-section.
- 2 - Choose the "Directories" option, to display all the required parameters.

Parameterizing

In the "C++ *generation*" sub-section, there are more than 30 parameters used to parameterize the generation and compilation work products. They are divided into eleven groups:

- ◆ external edition
- ◆ code generation options
- ◆ directories for saving the generated files, and for searching for inclusions and libraries
- ◆ suffixes for the generated files
- ◆ UML profiles
- ◆ Visibility declarations for attributes and associations
- ◆ Visibility for accessors
- ◆ libraries to be linked
- ◆ production options
- ◆ the tools used to compile and edit links
- ◆ MFC specific parameters

Each group contains parameters for work products. In the following section, we shall present a detailed description of each parameter and its work product.

Detailed description of parameters

External editor

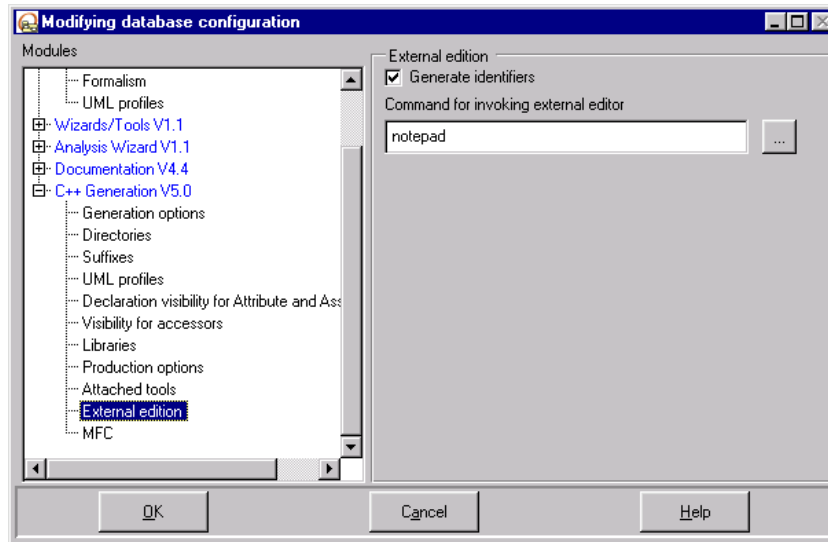


Figure 7-2. The "External edition" sub-section of the "Edit configuration" dialog box

The ... field	is used to ...
Generate identifiers	insert the markers in the generated code files. We recommend that it remain selected, in order to take advantage of the code/model consistency management system.
Command for invoking external editor	specify the name of editor used to edit the generated files from Objecteering/UML.

Both parameters concern generation work products, and the second also concerns compilation work products.

Generation options

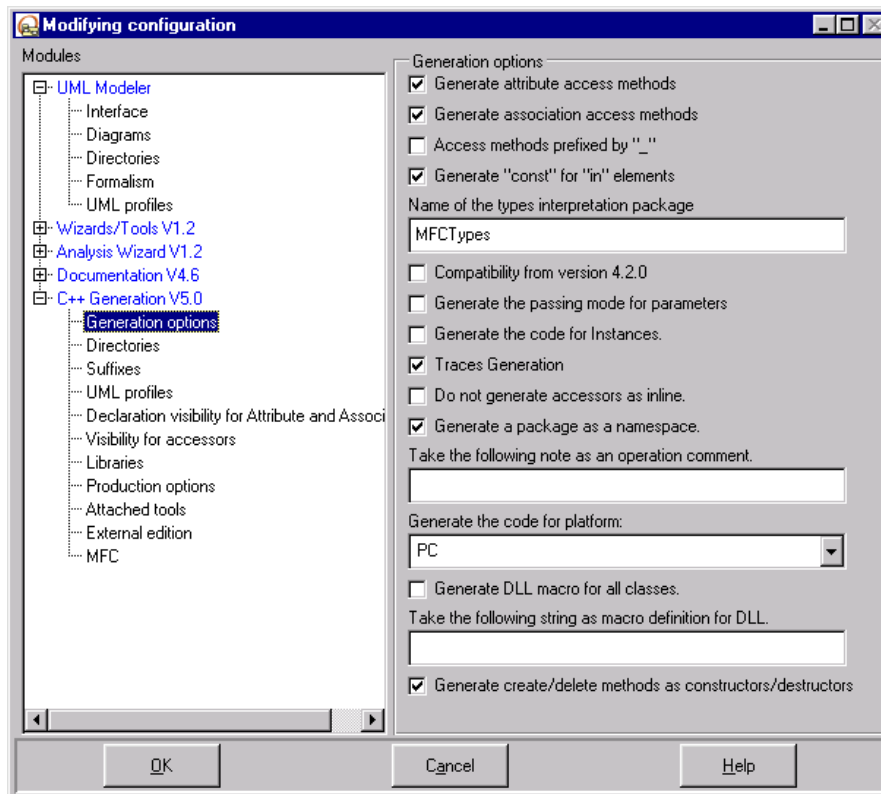


Figure 7-3. The "Generation options" sub-section of the "Edit configuration" dialog box

The ... field	is used to ...
Generate attribute access methods	generate modification methods for the attributes: set_Attribute()
Generate association access methods	generate modification methods for the association: append_role() erase_role()
Access methods prefixed by "_ "	generate the name of the access method prefixed by "_ ", instead of "get_" by default.
Generate "const" for "in" elements	generate the "const" keyword for the "entry" elements.
Name of the types interpretation project	indicate the project to be used within the generator to translate basic types.
Compatibility from version 4.2.0	keep the C++ module compatible with the version previous to 4.2.0.
Generate the passing mode for parameters	generate string (for example, In Out InOut) to indicate the parameter passing mode.
Generate the code for instances	generate C++ code on instances.
Traces generation	invalidate the generation of the CR_TRACE macro.
Do not generate accessors as inline	generate accessors as being non-inline.
Generate a package as a namespace	generate a namespace for a package.
Take the following note as an operation comment	define the type of the note whose contents should be taken into account as the operation comment.
Generate the code for platform	generate the code necessary to the platform specified.
Generate DLL macro for all classes	generate all classes as if they were tagged {C++DLL} (PC/Visual C++).
Take the following string as macro definition for DLL	define the string which will be generated in the macro, in place of "MSVC_DLL" for all the classes not tagged {C++DLL} or tagged {C++DLL} but with no parameters.
Generate create/delete methods as constructors/destructors	generate create and delete functions as accessors and destructors.

Chapter 7: Module parameters

These parameters concern the generation work product.

Note: "*access methods*" have prefixes, such as "set_", which can be parameterized.

Directories

For Objectteering/UML on a UNIX type operating system, it is possible to use the UNIX system environment variables when specifying directories.

Where several directories can be specified, you must specify the paths using the ";" character.

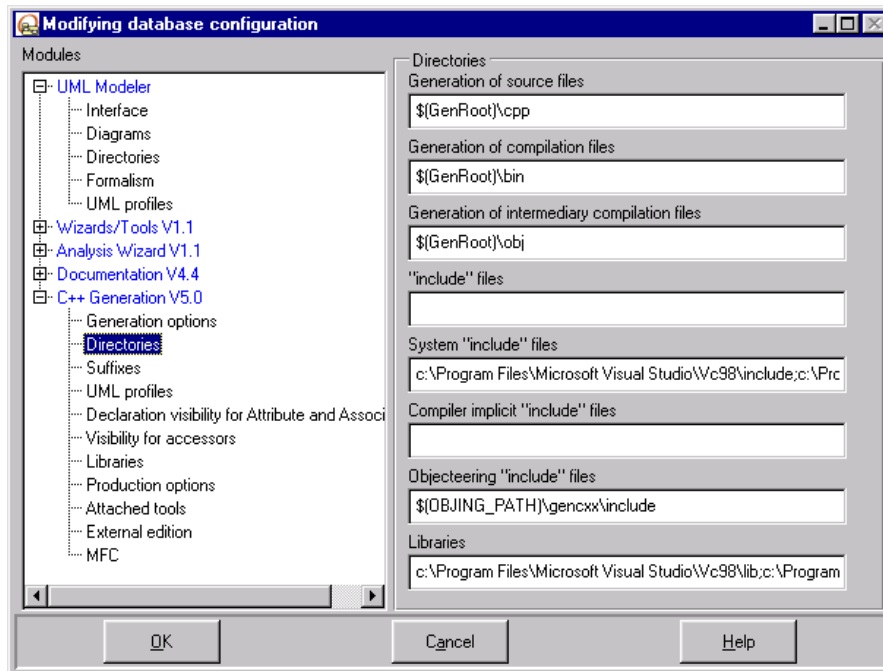


Figure 7-4. The "Directories" sub-section of the "Edit configuration" dialog box

The ... field	concerning the ... work product	specifies ...
Generation of source files	generation	the directory containing the generated code files
Generation of compilation files	compilation	the directory containing the Makefiles, the final target files and the compilation error files
Generation of intermediary compilation files	compilation	the directory containing the compiled object files
"include" files	compilation	the search path for the "include" files
System "include" files	compilation	the search path to find the system's header files
Compiler implicit "include" files	compilation	the search path to find the header files
Objectteering "include" files	compilation	the search path to find the library supplied by Objectteering/UML
Libraries	compilation	the search path to find the libraries.

Suffixes

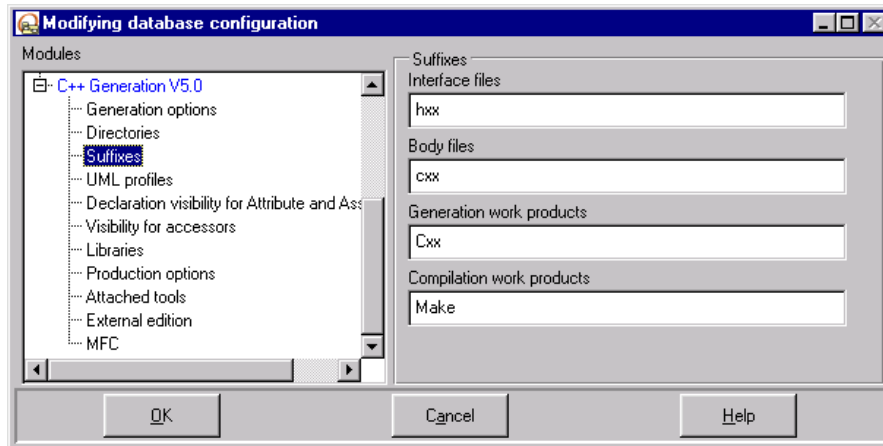


Figure 7-5. The "Suffixes" sub-section of the "Edit configuration" dialog box

The ... field	concerning the ... work product	specifies ...
Interface files	generation	the header file suffix
Body files	generation	the body file suffix
Generation work products	generation	the complete default name of the generation work product visible in the explorer
Compilation work products	compilation	the complete default name of the compilation work product visible in the explorer

UML profiles

You must specify UML profiles (that can be defined by the *Objecteering/C++* module) by supplying their absolute path (default#...).

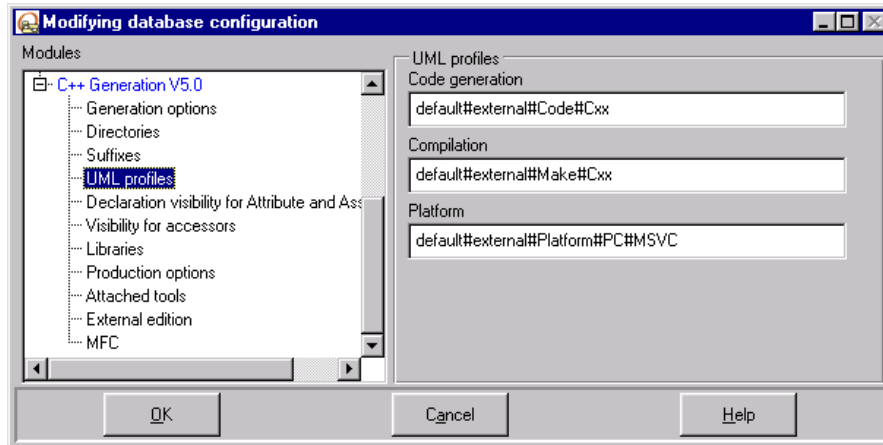


Figure 7-6. The "UML profiles" sub-section of the "Edit configuration" dialog box

The ... field	concerning the ... work product	specifies ...
Code generation	generation	the UML profile used for producing J code
Compilation	compilation	the UML profile used for producing the "Makefile" files
Platform	compilation	the UML profile defining the platform's characteristics (description of the target)

"Libraries"

Libraries are specified by supplying their access path on the current system. When several libraries are specified, you must separate them using the ";" character.

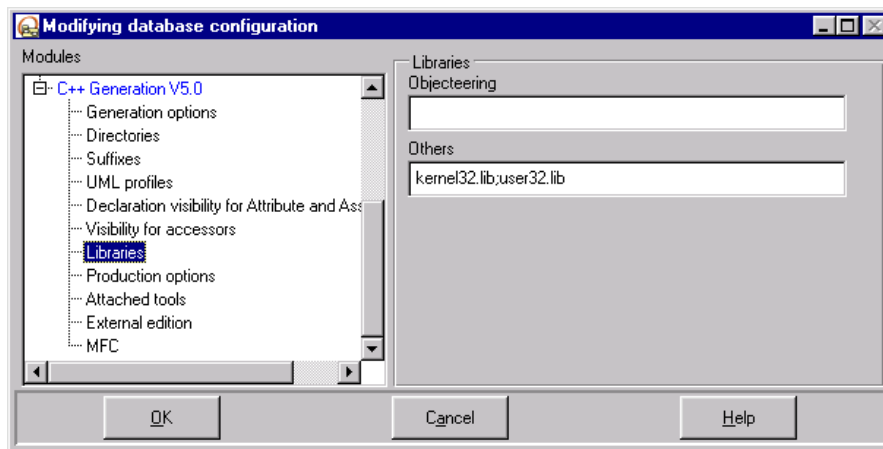


Figure 7-7. The "Libraries" sub-section of the "Edit configuration" dialog box

The ... field	displays ...
Objectteering	the Objectteering/UML base types library
Others	the other libraries used for developing

Both parameters concern only the compilation work product.

Note: If the "ObjectteeringTypes" types package is used, the name of the Objectteering/UML library has to be entered (for example, "libO.lib" for Windows).

Production options

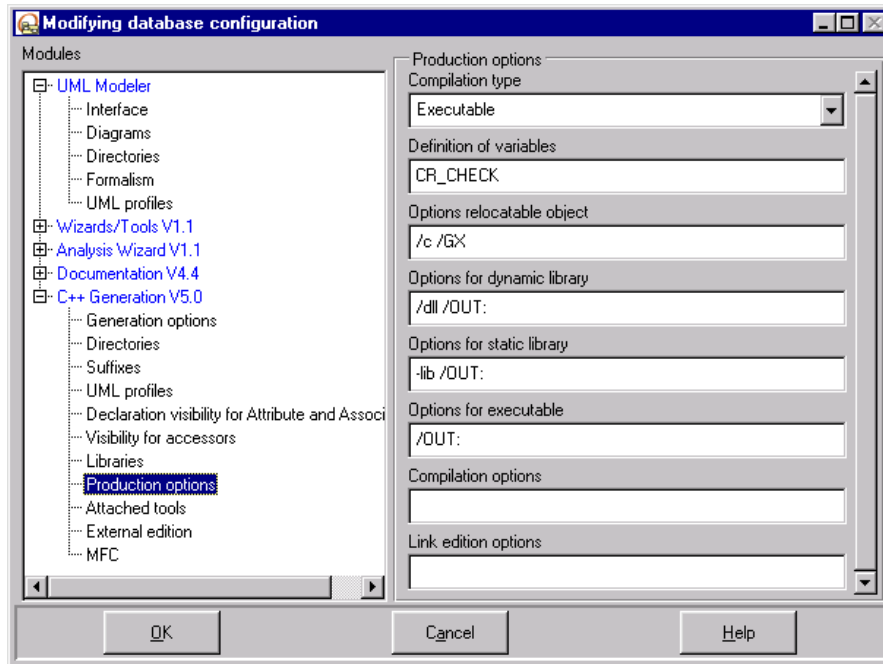


Figure 7-8. The "Production options" sub-section of the "Edit configuration" dialog box

The ... field	displays ...
Compilation type	the definition of the type of result expected from compilation (executable, relocatable, ...)
Definition of variables	the variables transmitted by default to the Makefile for pre-compilation
Options relocatable object	the compilation options for constituting a relocatable.
Options for dynamic library	the compilation options for constituting a dynamic library
Options for static library	the compilation options for constituting a static library
Options for executable	the compilation options for constituting an executable.
Compilation options	the default compilation options
Link edition options	the default options when editing links

Attached tools

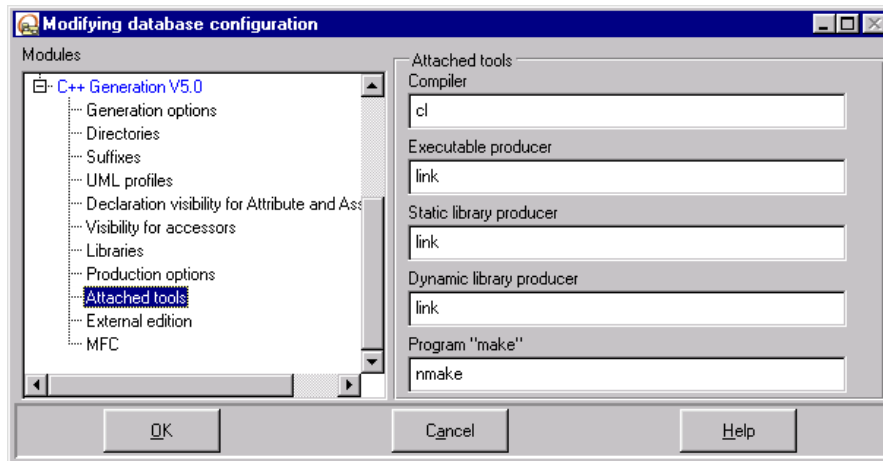


Figure 7-9. The "Attached tools" sub-section of the "Edit configuration" dialog box

The ... field	displays ...
Compiler	the compilation command
Executable producer	the command for generating executables
Static library producer	the command for producing static libraries
Dynamic library producer	the command for producing dynamic libraries
Program "make"	tool used to interpret the "Makefile" file

MFC

This applies only if you are using Microsoft Foundation Classes in a Windows environment.

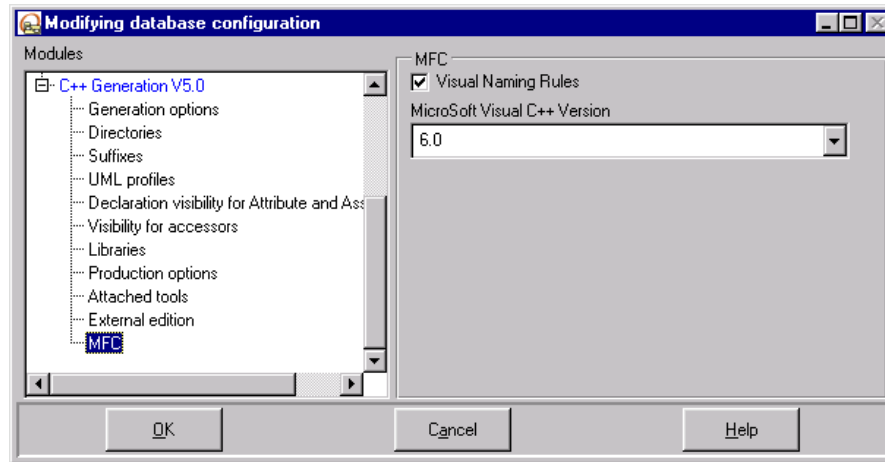


Figure 7-10. The "MFC" sub-section of the "Edit configuration" dialog box

The ... field	is used to ...
Visual Naming Rules	specify the use of MFC naming rules for attribute and relationship accessors (for example, GetCount, AddTail)
Microsoft Visual C++ Version	display the current version of Microsoft Visual C++ used in conjunction with Objectteering/UML

Visibility for attributes and associations

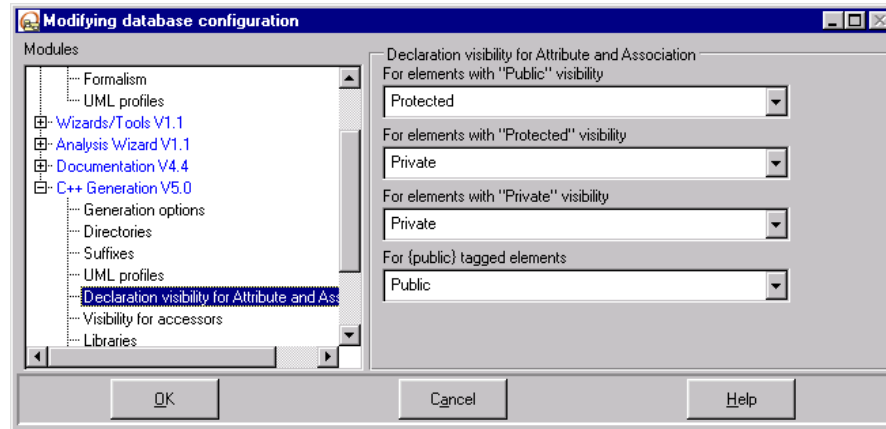


Figure 7-11. The "Declaration visibility for Attribute and Association" sub-section of the "Edit configuration" dialog box

The ... field	is used to ...
For elements with "Public" visibility	declare the visibility of the C++ instance variable generated for public attributes.
For elements with "Protected" visibility	declare the visibility of the C++ instance variable generated for protected attributes.
For elements with "Private" visibility	declare the visibility of the C++ instance variable generated for private attributes.
For {public} tagged elements	declare the visibility of the C++ instance variable generated for attributes annotated <i>{public}</i> .

Visibility for accessors

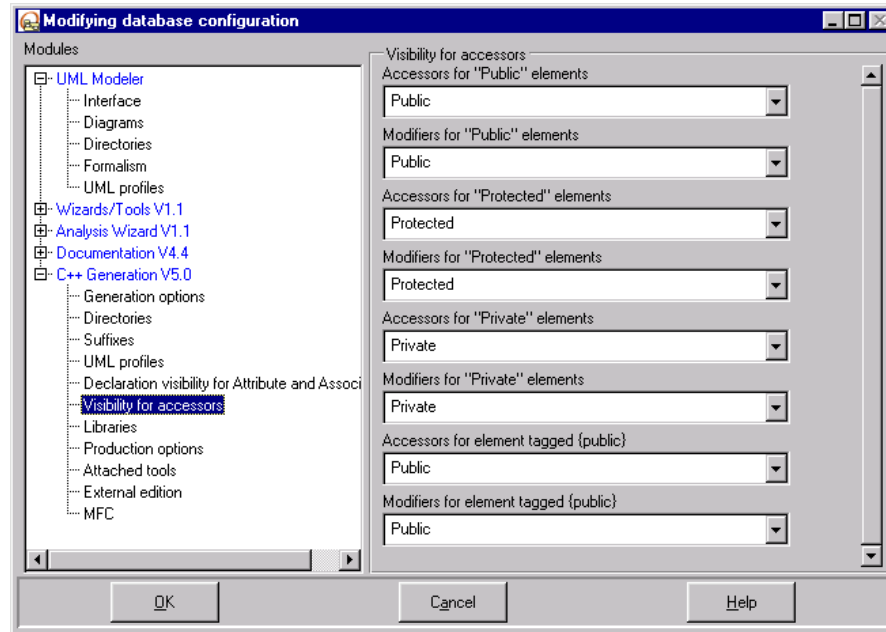


Figure 7-12. The "Visibility for accessors" sub-section of the "Edit configuration" dialog box

The ... field	is used to ...
Accessors for "Public" elements	declare the visibility of "get" type accessors for all public attributes.
Modifiers for "Public" elements	declare the visibility of "set" type accessors for all public attributes.
Accessors for "Protected" elements	declare the visibility of "get" type accessors for all protected attributes.
Modifiers for "Protected" elements	declare the visibility of "set" type accessors for all protected attributes.
Accessors for "Private" elements	declare the visibility of "get" type accessors for all private attributes.
Modifiers for "Private" elements	declare the visibility of "set" type accessors for all private attributes.
Accessors for elements tagged {public}	declare the visibility of "get" type accessors for all attributes annotated {public}.
Modifiers for elements tagged {public}	declare the visibility of "set" type accessors for all attributes annotated {public}.

Chapter 8: Generating code for a package

Overview of code generation on a package

The purpose of a package

A package is used to structure classes. The equivalent notion in C++ is a namespace. A package is used for:

- ◆ documentation
- ◆ production (development unit, library, test unit, etc.)
- ◆ coding: it is possible to define the information (new data types, constants) shared by all the classes of a package. The invariant of the package provides the general rules for the package which are followed by all its classes.

Correspondence

A C++ namespace with the same name as the package is generated from a package. All the package's classes are defined in the context of the associated namespace, except where the package is annotated `{C++NoNameSpace}` or `{C++Root}`.

Package invariant

A package invariant is checked each time the invariant of a contained class is valid. A package invariant accesses all the members of the package's contained classes.

A constraint stereotype C++ "invariant" on the package defines an invariant on the package.

For example, if the S package contains the classes C1 and C2, an invariant rule can be defined as follows : "C1::card_instance()> C2::card_instance()".

Initializing instances



: Button for creating an object

Objects created as instances in a package are generated as variables in the namespace of the package. These objects are implemented in their order of declaration.

Generalization and use

When a package S1 inherits from a package S2, we generate using a namespace of S2 in S1. The invariant of a derived package proves true the invariant of its base package.

Tagged values

Specific C++ tagged values available for a package are:

The ... tagged value	Role
{noinclude}	No generation of the "include".
{extern}	External package. A file name is supplied as a parameter.
{nocode}	No code generation for the package.
{MFCInclude}	Generation MFC specifically concerning include.
{C++NoNameSpace}	No generation of namespaces on packages and no generation of a sub-directory in the file structure.
{C++Root}}	Namespace and directory structure starts from this package, as if it were the root package.
{C++Name}	Name to take in account to generate nameSpace and file.

Notes

Specific C++ notes available for a package are:

The ... note	Role
C++ InterfaceHeader	text inserted at the beginning of the header file (.hxx)
C++ BodyHeader	text inserted at the beginning of the body file (.cxx).)

Constraint stereotypes

The C++ stereotypes applicable on a constraint and available for a package are:

The ... constraint stereotype	Role
C++Invariant	code used to generate the package invariant

Tagged values for a package

The {extern} tagged value

The {extern} tagged value on a package indicates that the content of this package is not modeled in Objectteering/UML, but is part of an external library.

The parameter of this tagged value is the header file in which the package is declared.

If the {extern} tagged value is placed on a package, it must equally be placed on all the classes belonging to this package.

Example:

The {extern} tagged value on...	with the ... parameter	is translated by ...
an "S" package	<code>fics.h</code>	a header file on the S package containing only: <code>#include "fics.h"</code> and an empty body file

The {noinclude} tagged value

The {noinclude} tagged value on a package indicates that the inclusion of files usually generated by Objectteering/UML should be ignored. This obliges the developer to declare them manually.

The {nocode} tagged value

The {nocode} tagged value on a package prevents this package from being generated. No C++ file is generated for the package.

If the {nocode} tagged value is added to a package, it must also be added to all the classes belonging to this package.

The {noCodeForAll} tagged value

The {noCodeForAll} tagged value is used to not generate files for the package, or for its component elements. No use links or associations are generated to or from the package or the classes it contains.

The {MFCInclude} tagged value

The {MFCInclude} tagged value on a package generates includes specific to MFC. These includes are related to MFC components and collections.

The {C++Name} tagged value

The parameter of the {C++Name} tagged value takes precedence over the modeling name during the generation phase.

The {C++NoNameSpace} tagged value

This tagged value generates no namespace or sub-directories for the package it annotates. However, namespaces for sub-packages are generated.

The {C++Root} tagged value

The structure of namespaces and directories starts from the package annotated with the {C++Root} tagged value.

Chapter 9: Generating code for a class

Overview of code generation on a class

Translating a model's class

A "C" class in a model corresponds to a C++ class with the same name, generated as:

- ◆ an interface file (e.g. C.hxx)
- ◆ a body file (e.g. C.cxx)

This chapter will only deal with the generation of a class generalization, a class invariant, a main class and a template class. For the other component elements of a class, such as attributes, associations and methods, please refer to their corresponding chapters.

Tagged values

The specific C++ tagged values available for a class are:

The ... tagged values	Parameters ...	Is used to ...
{C++Name}	name to generate	indicate the name to be taken into account to generate the class name and the file associated.
{extern}	external file name	indicate a C++ class outside the Objecteering/UML repository.
{MFCDynamicMacro}		specify MFC macro generation.
{nocode}		indicate no code generation for the class.
{noinclude}		indicate no generation of the "include".
{structure}		indicate generation of a C structure
{C++DLL}	the name of the DLL macro ("MSVC_DLL" if blank)	generate the definition of the export/import macro needed to compile ".dll" libraries.

Notes

Specific C++ notes available for a package are:

The ... note type	Is used to ...
C++BodyHeader	provide text to be inserted at the beginning of the body file (.cxx).
C++Inheritance	provide text of the declaration of a generalization
C++InterfaceHeader	provide text to be inserted at the beginning of the header file (.hxx)
C++PrivateMember	provide text to be inserted at the beginning of the private declaration zone.
C++ProtectedMember	provide text to be inserted at the beginning of the protected declaration zone
C++PublicMember	provide text to be inserted at the beginning of the public declaration zone
MFCMessageMacro	provide the text of the declaration of the callbacks function

Note : It is possible to have several notes of the same type for the same class; these are then generated one after the other.

Class generalization

Generalization



The "Specialize" icon

A generalization between classes can be modeled using the icon in the "Items" tab of the properties editor. The appropriate inclusion is then generated.

You can add generalization text using a C++ note, but the "#include" instruction is not generated by the *Objectteering/C++* module. Developers have to add the necessary inclusion with *C++InterfaceHeader* or *C++BodyHeader*.

Virtual generalization

"Virtual" C++ generalization is not systematically deduced by Objectteering/UML. It is necessary in the cases below.

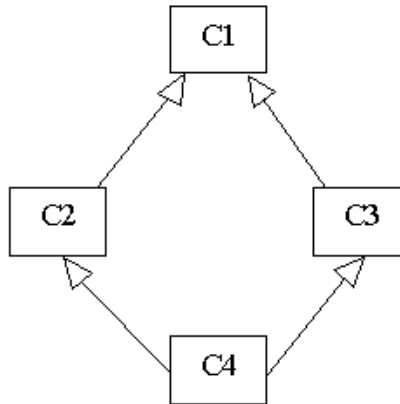


Figure 9-1. Virtual C++ generalization necessary in the case of repeated generalization from C1 to C4

To generate a virtual generalization, you must apply the *{virtual}* tagged value on the object representing the generalization in the model.

Private and protected generalization

"*Private*" or "*protected*" C++ generalization is generated if the tagged value *{private}* or *{protected}* is presented on the generalization.

For a complete description of the tagged values available for generalizations, please refer to chapter 13, "*Generating code for a generalization, an enumerate and basic types*", of this user guide.

Class invariant

Clauses

Objectteering/UML allows you to express clauses, Boolean conditions which must, in all cases, be verified.

Example:

For a "*Policeman*" class, two invariant clauses can be defined either as:

- ◆ age >= 18
 - ◆ height >= 1.75
- or as:
- ◆ (age >= 18)&&(height >= 1.75)

Controlling an invariant clause

Invariant clauses are controlled during the running of a program, if the program has been compiled with the "*CR_CHECK*" compilation option.

A class invariant:

- ◆ always expresses true properties for each of the instances
- ◆ must be verified systematically from the end of the running of the constructor until the start of the running of the destructor (if the case arises)
- ◆ is controlled in a dynamic way at the start and at the end of each actived method

Recommendation

It is recommended that you always place the "*-DCR_CHECK*" option during:

- ◆ the test and debugging phases of a program
- ◆ the final confirmation


This option introduces an overload during the running of a program and must be deactivated when the program is valid. The application must be tested again after the suppression of the invariant controls.

Faulty clauses

When a clause is not verified (for example, if a policeman is aged 16):

- ◆ an error message is generated with the associated line number indicated
- ◆ for applications in UNIX, a "core dump" is provoked. The program thus comes to a stop upon detection of a bug and it is possible for you to analyze a "core" file with a debugger
- ◆ for applications in PC, you can stop the application by calling the abort function, and a window then displays a message indicating the error

Construction

To create an invariant clause attached to a class, you have to create a constraint, using the  "Create a constraint" icon in the "Items" tab of the properties editor. In the constraint dialog box, add a stereotype, by selecting "C++Invariant" in the "Add a stereotype" field at the bottom of this dialog box.

Main class

Introduction

This section is only relevant if you are using the *ObjectteeringTypes* UML modeling project.

Definition

In a model aiming to generate a binary executable, there must be one and only one class defined as the "main" class (for this, check the "main" check box in the class's dialog box). The class must contain an operation called "start".

This class is declared in C++ as specializing the "application" class. It therefore makes it possible to own operations which allow access to the "argv" and "argc" parameters of the main function.

The "main" class will have one unique instance. The "start" operation is called to start running the application.

Handling

Any class specializing the "main" class is also a main class. The terminal class of this generalization graph is the class representing the application.

An instance of this class is automatically declared by Objectteering/UML. This instance is "static" from the C++ viewpoint. The programmer can declare no other instance.

The compulsory method, "void start()" in this case, is activated automatically after the construction of all the static instances of the application's classes.

It is possible to define the information (notes, constants) shared by all the package's classes.

Example


If a class called "*my_application*" is a "*main*" class:

- ◆ a unique instance of "*my_application*" is declared
- ◆ the application starts running when the "*start()*" method coded in "*my_application*" is called

The instance access is carried out in C++ using the "*my_application::get_instance()*" instruction.

Generic class: Template

Implementation

The generic class involves the declaration and instantiation of a template class. A template class is modeled using the  "Create a template parameter" icon, through a relationship with Template Parameter class.

The declaration of a template class is modeled by adding template parameters to a class. Instantiation takes place at *Attribute*, *Association*, *Parameter*, *Generalization* or *Instance* level. The parameters of the {bind} tagged value give different information to instantiate a template class.

Declaring a template class

A template class is declared by an association with the Template Parameter class. The class must be primitive.

Example:

On a C class, modeling with TemplateParameter, its Template parameter value is:

T, int s2

The generated code is as follows :

```
template<class T, int S2>
class C{
...
};
```


Instantiating a template class

A class template can be instantiated wherever a class can be mentioned:

- ◆ attributes
- ◆ associations
- ◆ parameters
- ◆ generalization
- ◆ instances

by the {bind} tagged value with the parameters to instantiate the template. The type of model element is equal to a Template class.

Example:

If the C class in the above example is to be an association Role in an A class, at the association you must annotate with the tagged value as follows:

```
{bind ( "myClass", "20" )}
```

The generated code is as follows:

```
class A :  
..  
    inline int card_cRole() const;  
    inline class C<maClass, 20>* _cRole (int) const;  
    inline void append_cRole( const class C<maClasse, 20>*  
);  
    inline void erase_cRole( const class C<maClasse,20> * );  
..  
};
```

where "myClass" can be a general class, that is a class, a type or an enumerate.

Tagged values on a class

The {extern} tagged value

The {extern} tagged value on a class indicates that the content of the class is not modeled in Objectteering/UML, but is part of an external library.

The parameter of this tagged value is the header file in which the class is declared.

Example:

The {extern} tagged value on...	with the ... parameter	is translated by ...
a "C" class	<code>fi.cc.h</code>	a header file on the C class containing only: <code>#include "fi.cc.h"</code> and an empty body file

The {noinclude} tagged value

The {noinclude} tagged value on a class indicates that the inclusions of files usually generated by Objectteering/UML (during the use of a class for example) must be ignored. The user must then use the "C++*InterfaceHeader*" zone to declare his own inclusions.

The {structure} tagged value

The {structure} tagged value on a class indicates that the class model element is to generate a C structure.

The {MFCDynamicMacro} tagged value

The {MFCDynamicMacro} tagged value on a class indicates that MFC specific macros are to be generated. These macros are as follows:

- ◆ DECLARE_DYNCREATE (className)
- ◆ IMPLEMENT_DYNCREATE (className, parentClassName)

The {nocode} tagged value

The {nocode} tagged value on a class prevents this class from being generated. No C++ file is generated for this class.

The {noCodeForAll} tagged value

The {noCodeForAll} tagged value is used to not generate any code either for the class and the associations for which the class is the destination, or for use links towards the class.

The {C++Name} tagged value

The parameter of the {C++Name} tagged value takes precedence over the modeling name during the generation phase.

The {C++DLL} tagged value

This tagged value only applies to Visual C++/PC.

For a class A annotated {C++DLL()} without parameters, the generated code is as follows:

```
#ifdef ObjingDLL
  #ifdef MSVC_DLL_EXPORT
    #define MSVC_DLL_declspec(dllexport)
  #else
    #define MSVC_DLL_declspec(dllimport)
  #endif
#else
#define MSVC_DLL
#endif

class MSVC_DLL A {
  ...
};
```

If the {C++DLL} tagged value has a parameter (for example, {C++DLL(myDLLMacro)}), the code generated is the same as the code above, except that "MSVC_DLL" is replaced throughout by "myDLLMacro".

Notes on a class

The C++*BodyHeader* note

The "*C++BodyHeader*" note on a class is used to express a section of code to be inserted at the beginning of a body file.

Example:

A " <i>C++BodyHeader</i> " note containing ...	is translated ...
<code>extern void fonc();</code>	at the beginning of a body file, after the inclusions generated by Objectteering/UML, by: <code>extern void fonc();</code>

The C++*InterfaceHeader* note

The "*C++InterfaceHeader*" note on a class is used to express a section of code to be inserted in the header file, before the class declaration.

Example:

A " <i>C++InterfaceHeader</i> " note containing...	is translated ...
<code>#include "myincl.h"</code>	at the beginning of the header file, just before the class declaration , by: <code>#include "myincl.h"</code>

The C++Inheritance note

The "*C++Inheritance*" note on a class allows the expression of a generalization in a free manner, without modeling it in Objecteering/UML.

The use of this C++ note should be limited to complex cases of generalization.

Example:

A " <i>C++Inheritance</i> " note containing...	is translated ...
public T	in the header file, to declare the class , by: class C : public T

Note: You must add the necessary inclusion concerning the related generalization.

The C++PrivateMember note

The "*C++PrivateMember*" note on a class allows the addition of free C++ code in the private declaration section.

Example:

A " <i>C++PrivateMember</i> " note containing...	is translated ...
friend class C;	in the declaration of the class, by: private: friend class C;

The C++ProtectedMember note

The "*C++ProtectedMember*" note on a class allows the free addition of C++ code in the protected declaration section.

Example:

A " <i>C++ProtectedMember</i> " note containing...	is translated ...
<code>mutable int count;</code>	in the class declaration, by: protected: <code>mutable int count;</code>

The C++PublicMember note

The "*C++PublicMember*" note on a class allows the free addition of C++ code in the public declaration section.

Example:

A " <i>C++PublicMember</i> " note containing...	is translated ...
<code>void f() throw;</code>	in the class declaration, by: public: <code>void f() throw;</code>

The MFCMessageMacro note

The "*MFCMessageMacro*" note on a class allows MFC messages, which are used to map instructions in the protected declaration section, to be added.

Example:

An " <i>MFCMessageMacro</i> " note containing...	is translated ...
ON_BN_CLICKED (IDOX, ONOK)	in the class declaration, by: protected: DECLARE_MESSAGE_MAP ()
ON_BN_CLICKED (IDOX, ONOK)	in the class implementation, by: BEGIN_MESSAGE_MAP (className, parentClassName { ON_BN_CLICKED (IDOX, ONOK) } END_MESSAGE_MAP () ()

Chapter 10: Generating code for an
attribute

Overview of code generation on an attribute

Definition

An attribute is a member of a class. Its type can be:

- ◆ a base type (int, real, char, string, etc)
- ◆ an enumerate
- ◆ a primitive class (the "*Primitive*" check box is checked in the class dialog box)

An attribute can also have a type that the model does not know, but defined in C++ form by the programmer. In this case, you must use a specific C++ note.

An attribute can be a single object or a set of objects of the same type.

Generation

The generation of an attribute involves declaration, initialization, access methods and inclusion.

Access methods are divided into three groups: read methods, modification methods and full access methods.

Tagged values

Tagged values specific to C++ available for an attribute are:

The ... tagged value	Purpose
{&}	Generation of a reference
{*}	Generation of a pointer
{access}	Generation of the attribute's modification methods
{array}	Generating a table
{bind}	Instantiation of a template class
{const}	Generation of a constant
{create}	Initialization of the attribute in the constructor initializer-list
{fullaccess}	Generation of the full access methods if it consists in a set
{long}	Long integer or double real
{nocode}	No code generation for the attribute
{noconst}	No const string for access methods
{public}	Generation of the modification methods as being public
{short}	Short integer
{type}	Generation of some special implementation
{unsigned}	Unsigned integer
{own}	No automatic generation of accessors.
{mutable}	Generation of the " <i>mutable</i> " attribute. " <i>mutable</i> " is a key word in C++.
{noInline}	Non-inline attribute access methods.

Notes

C++ specific notes available for an attribute are:

The ... note type	Purpose
C++TypeExpr	C++ expression of the attribute declaration. It replaces the usual generation.
C++Value	Initializing an attribute. This expression replaces the default value.
C++AccessDecl	C++ code for declaring the attribute, only if the attribute is annotated {own}.
C++AccessDef	C++ code for defining the attribute, only if the attribute is annotated {own}.

Declaration and initialization

The "Attribute" dialog box

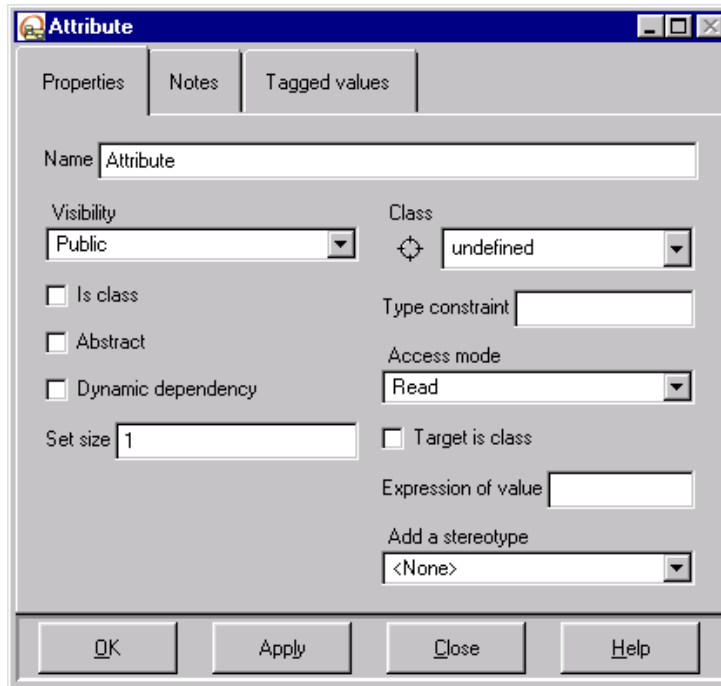


Figure 10-1. The "Attribute" dialog box

Visibility

In the "Attribute" dialog box, there appears a field used to define the visibility of the attribute. The {public} tagged value on an attribute also concerns visibility. Visibility generated by default is as follows:

Visibility field in the dialog box or the {public} tagged value on the ... attribute	defines the visibility of the attribute declaration as being...
public	protected
protected	protected
private	private
{public}	public

Note: Visibility can be modified through module parameters (for further information, please refer to the "Detailed description of parameters" section in chapter 7 of this user guide).

Class attribute

The "Target is class" checkbox in the dialog box is used to define whether or not the attribute is a class attribute. A class attribute is translated in C++ as a "static" attribute. Initialization of a static attribute is carried out in the body files of its own class.

Initialization

The "*Expression of value*" field in the dialog box and the *C++Value* note on the attribute have the same function, allowing you to define the initial value for an attribute or to enter the C++ code for a dependent attribute (see the following section).

An attribute is initialized in the constructor(s). By default, the initialization is in the body of the constructor(s). If you annotate the attribute with the {create} tagged value, the initialization is inserted in the constructor initializer-list.

If there is no constructor defined in the class, a default constructor is created to initialize attributes.

Attributes are initialized in their order of declaration in the class.

Dynamic dependent attribute

In the "*Attribute*" dialog box, there is a check box named "*dynamic dependency*". An attribute with this tickbox checked means that the attribute is decided by the state of the class object. Only one access method (`get_Attribute_x()`) is generated. The method allows you to get the value of the attribute.

You need to define the code, calculating the attribute, in the "*Expression of value*" field or through the *C++Value* note. This code is inserted into the body of the only access method.

Access methods

Encapsulation

To protect direct access to an attribute, several access methods are generated, except for attributes annotated {own}. They are divided into three groups:

The ... group	for example ...
Read methods	get_Attribute_x() card_Attribute_x()
Modification methods	set_Attribute_x() append_Attribute_x() erase_Attribute_x()
Full access methods	get_all_Attribute_x() set_all_Attribute_x()

Note : The two full access methods correspond, respectively, to the read and modification categories.

In the MFC context, the naming of access methods conforms to the MFC naming rules (for example, AddTail, GetCount).

Visibility of access methods

The visibility of the access methods generated by default is determined by the visibility field in the dialog box or the tagged value on the attribute (referring to the above section on the visibility of the attribute declaration). The visibility is as follows:

Visibility field in the dialog box or the {public} tagged value on the ... attribute	The visibility of the ... read methods	The visibility of the ... modification methods
public	public	protected
protected	protected	protected
private	private	private
{public}	public	public

Tagged values on an attribute

The {&} tagged value

The {&} tagged value on an attribute is used to generate the attribute in a reference form.

Example:

The {&} tagged value on ...	is translated by ...
an attribute "Att" of integer type	<code>int & Att;</code>

The {*} tagged value

The {*} tagged value on an attribute is used to generate the attribute in the form of a pointer.

Example:

The {*} tagged value on ...	is translated by ...
an attribute "Att" of integer type	<code>int * Att;</code>

The {access} tagged value

The {access} tagged value on an attribute generates modification method(s) for the attribute.

As far as parameterization of the *Objecteering/C++* module is concerned, the "*Generate the access methods to the attributes*" option determines whether to globally generate the modification methods for all the attributes.

Example:

The {access} tagged value on ...	is translated by ...
an attribute "Att" of integer type	adding methods: <code>void set_Attr(int);</code>

The {array} tagged value

The {array} tagged value on an attribute with the "set" type allows the generation of the member data representing the attribute in the form of an array.

Example:

The {array} tagged value on ...	is translated by ...
a attribute "Att" of type : set of integers with 10 elements	<code>int Att[10];</code>
an attribute "Att" of type : set of integers type with an undefined number of elements	<code>cr_array <int> Att;</code>

The {const} tagged value

The {const} tagged value on an attribute allows the generation of the attribute as constant member data.

Example:

The {const} tagged value on ...	is translated by ...
an attribute "Att" of integer type	<code>const int Att;</code>

The {create} tagged value

The {create} tagged value on an attribute generates the attribute initialization in the constructor's initializer-list (instead of in the constructor's body).

Example:

The {create} tagged value on ...	is translated...
an attribute "Att" of integer type with "2" for default value	in the initialization list of the class constructor by: <code>Att(2)</code>

The {fullaccess} tagged value

The {fullaccess} tagged value on an attribute of set type generates access methods to the set itself.

Example:

The {fullaccess} tagged value on ...	is translated by ...
an attribute "Att" with the integer set type	adding methods: <pre>void set_all_Att(const list_int&); const list_int& get_all_Att() const;</pre>

The {long} tagged value

The {long} tagged value on an attribute with the integer type is used to indicate that it has the "long int" type.

Example:

The {long} tagged value on ...	is translated by ...
an attribute "Att" of integer type	long Att;

The {short} tagged value

The {short} tagged value on an attribute of integer type allows you to indicate that it is of "short int" type.

Example:

The {short} tagged value on ...	is translated by ...
an attribute "Att" of integer type	short Att;

The {unsigned} tagged value

The {unsigned} tagged value on an attribute of integer type allows you to indicate that the attribute is of "unsigned int" type.

Example:

The {unsigned} tagged value on ...	is translated by ...
an attribute "Att" of integer type	unsigned Att;

The {public} tagged value

The {public} tagged value on an attribute generates the attribute's declaration and its access methods with the public visibility. By default, the methods that modify the attribute and the declaration of the attribute itself are generated in the protected part of the class's definition.

Example:

The {public} tagged value on ...	is translated by ...
an attribute "Att" of integer type with public visibility	<pre>public: int get_Attr() const; void set_Attr(int); int Att; instead of: public: int get_Attr() const; protected: void set_Attr(int); int Att;</pre>

The visibility generated can be modified through module parameters (for further information, please refer to the "*Detailed description of parameters*" section in chapter 7 of this user guide).

The {bind} tagged value

The {bind} tagged value is used to define an attribute in the form of a class template.

The parameters of this tagged value are the data used to instantiate the "template".

Example:

The {bind} tagged value on ...	with the parameters ...	is translated by ...
an attribute of a type template class "C"	anyClass 20	the type of the attribute is an instantiation of the template class C with the parameters: class C <anyClass, 20> Att ;

Please refer to the template generic class section in chapter 8 of this user guide for a description of template class generation.

The {noconst} tagged value

By default, the *Objectteering/C++* module generates "const" for certain parameters belonging to the attribute's access methods. It also generates "const" for read access methods.

This tagged value prevents the generation of "const".

The {type} tagged value

The {type} tagged value is used to enter the type of collection used if it exists in the predefined types.

For the advanced *Objectteering/C++* module user, you can modify or create a new type interpreting project and add new basic types implementations. This tagged value is used to generate the attribute with the new implementation.

Please refer to chapter 15 of this user guide for a detailed description of adaptation and parameterization.

The {nocode} tagged value

The {nocode} tagged value on an attribute prevents this attribute from being generated.

The {C++Name} tagged value

The parameter of the {C++Name} tagged value takes precedence over the modeling name during the generation phase.

The {own} tagged value

The {own} tagged value generates the declaration of the attribute, but does not generate accessors. The user can declare and define his own accessors using the "*C++AccessDec*" and "*C++AccessDef*" notes (for further information on these notes, please refer to the "*Notes on an attribute*" section in the current chapter of this user guide).

The {noInline} tagged value

The {noInline} tagged value is used not to generate accessors as being "*inline*". Their body is generated in the body (.cxx) instead of the header (.hxx).

The {mutable} tagged value

The {mutable} tagged value is used to generate the attribute as a "*mutable*" member data.

Notes on an attribute

C++TypeExpr

The "*C++TypeExpr*" note on an attribute allows the declaration of the attribute type.

If the attribute owns a "*C++TypeExpr*" note, it is used during the C++ generation. This allows the specification of a kind of C++ implementation.

The use of "*C++TypeExpr*" must be limited to the complex cases of declarations, which are impossible to express by any other means.

Example:

A " <i>C++TypeExpr</i> " note containing ...	is translated ...
void ** on an attribute named "Att"	to declare the attribute, by: void ** Att;

C++Value

The "*C++Value*" note on an attribute allows the declaration of an initialization value for the attribute.

If the attribute owns both an initialization value in the zone provided for it and a "*C++Value*" note, this note is used during C++ generation.

Example:

A " <i>C++Value</i> " note containing...	is translated ...
X:get_c() on an attribute named "Att"	in the class constructor by: set_Att(X::get());

C++AccessDecl

The "*C++AccessDecl*" note is used to declare {own} attribute accessors. To define the visibility of the accessor, a line "public:"/"protected:"/... must be inserted into the body of the note.

C++AccessDef

The "*C++AccessDef*" note is used to define accessors, to enter the C++ code in the body of the access methods declared in the "*C++AccessDecl*" note.

Chapter 11: Generating code for associations

Overview of code generation on an association

Definition

An association describes a relation between two classes. These classes cannot be primitive classes.

Modeling an association involves defining the name, deciding the multiplicity and orientation. Please refer to the "*Binary association dialog box*" section of chapter 3 of the *Objectteering/Model Dialog Boxes* user guide for further information.

The C++ generation of an association is described in this section.

Tagged values

The C++ specific tagged values available for an association are as follows:

The ... tagged value	Role ...
{access}	Generating modification methods
{array}	Generating a table
{C++Name}	Name to be taken into account to generate attribute name.
{create}	Initializing the association in the initializer-list of the constructor.
{fullaccess}	Generating the "set" methods.
{generic}	Instantiation of a template class.
{instanceHandling}	Generation of an instance handling association.
{nocode}	Generates no code for the association
{noconst}	No <i>const</i> string for the access methods
{noinit}	Prevents generation of the initialization of the associations
{own}	Implementation of accessors by the user
{noInline}	Non-inline accessors
{public}	Generating the public modification methods
{type}	Generating a special implementation
{mutable}	Generation of the " <i>mutable</i> " association. " <i>mutable</i> " is a key word in C++.
{virtual}	Creation of virtual access methods

Notes

C++ specific notes available for an association are:

The ... note type	Purpose
C++AccessDecl	C++ code for declaring the association, only if the association is annotated {own}.
C++AccessDef	C++ code for defining the association, only if the association is annotated {own}.

Generation

Overview

Generation for an association involves:

- ◆ generating a variable member, which references the objects associated with the association; by default, the name of the variable is the same as that of the association's class role.
- ◆ generating access methods
- ◆ generating the inclusion

A class association is translated in C++ as a *"static"* member variable. In this case, all the access methods for the association are static member functions.

Multiplicity

The generation of an association depends on the multiplicity, which is defined as follows:

If the multiplicity is equal to...	then...
"0-1" or "1-1"	the variable is a pointer its initial value is "NULL".
"0-n" (n1) ou "0-*"	the variable is a set of pointers its initial value is an empty set.

Access methods

To protect against direct accessing of the member variable which corresponds to an association, several access methods are generated, except for associations annotated {own}. They are divided into three groups:

The ... group	for example ...
Read methods	get_Role_x() card_Role_x()
Modification methods	append_Role_x() erase_Role_x()
Complete access methods	get_all_Role_x() set_all_Role_x()

Note: The two complete access methods correspond, respectively, to the read and modification categories.

The "*append_Role_x()*" and "*erase_Role_x()*" methods apply the following rule, which must be respected: any non null address passed as a parameter corresponds to currently linked objects.

In the MFC context, the naming of access methods conforms to MFC naming rules (for example, Add Fail).

Visibility

The visibility of the variable and the access methods, generated by default, is decided by the visibility field in the dialog box, or the tagged value on the association. The visibility is as follows:

Dialog box visibility field or the {public} tagged value on the association ...	Visibility of variable	Visibility of read methods	Visibility of modification methods
public	protected	public	protected
protected	protected	protected	protected
private	private	private	private
{public}	public	public	public

Note: Visibility can be modified through the module parameters (for further information, please refer to the "*Detailed description of parameters*" section in chapter 7 of this user guide).

Handling instances

Definition

Instance handling is a special association defined in the class, which implies:

- ◆ that the class handles all its instances
- ◆ that the maximum number of instances at any given moment may be limited

In terms of programming, this means that the class can at any time supply the list of its instances. It implies that the instance association must be an association of class. The manipulation of this set is governed by the visibility declared for it.

Modeling

Modeling an instance handling association requires:

- ◆ the creation of an association towards the class itself
- ◆ making sure that the association is "class"
- ◆ defining the multiplicity that represents the number of possible instances for the class
- ◆ making sure that the name of the association's object role is "*instance*", or that the association is annotated with the {instanceHandling} tagged value

Generation

Generating code for an instance handling association is carried out in the usual way. Generation requires:

- ◆ the generation of a static member variable corresponding to the association
- ◆ the generation of the static access methods according to the multiplicity
- ◆ the generation of the necessary inclusion

Moreover:

- ◆ in the constructor(s), the "*append_instance(this)*" method is called. This means that each time a new instance of the class is created, it is added to the set of instances.
- ◆ in the destructor(s), the "*erase_instance(this)*" method is called. This means that each time an instance disappears, it is removed from the set of instances.

Tagged values on an association

The {access} tagged value

The {access} tagged value allows the generation of modification methods for the association.

As far as parameterization of the *Objectteering/C++* module is concerned, the "Generate relationship access methods" option avoids having to place this tagged value.

Example:

The {access} tagged value on ...	is translated by ...
a "Role" association towards a "C" class	adding the methods: <pre>void append_Role(const C*); void erase_Role(const C*);</pre>

The {array} tagged value

The {array} tagged value on an association with a multiplicity that is higher than 1 generates the member data representing the association in the form of an array.

Example:

The {array} tagged value on ...	is translated by ...
an association "Role" towards a class "C", with the multiplicity "0-9"	<pre>C * Role[10];</pre>
an association "Role" towards a class "C", with the multiplicity "0-"	<pre>ext_array_C Role;</pre>

The {array} tagged value generates C++ arrays instead of ("set_of") sets.

Their implementation is less costly (memory usage, performance), but can be less flexible.

The {fullaccess} tagged value

The {fullaccess} tagged value on an association with a multiplicity higher than 1 generates access methods to the association itself.

Example:

The {fullaccess} tagged value on ...	is translated by ...
a "Role" association towards a "C" class, with the multiplicity "0-1"	adding the methods : <pre>void set_all_Role(const set_of_C&); const set_of_C& get_all_Role() const;</pre>

The {bind} tagged value

The {bind} tagged value indicates that it corresponds to a C++ template class. This tagged value has, as parameters, the data which allows the instantiation of the template class.

Example:

The {bind} tagged value on ...	with the parameter...	is translated by ...
an association "Role" with the multiplicity "0-1" towards a class "C" (generic itself)	anyClass 20	C<anyClass, 20> *Role;

Please refer to the "*Generic class: Template*" section in chapter 8 of this user guide for a description of template class generation.

The {noinit} tagged value

The {noinit} tagged value indicates that the association should not be initialized (NULL, or empty list).

The {public} tagged value

The {public} tagged value generates the declaration of the association and of its access methods with public visibility (by default, the modification methods and the declaration of the association are generated in the protected section).

Example:

The {public} tagged value on ...	is translated by ...
a "Role" association with the multiplicity "0-1" towards a class "C"	<pre>public: C* get_Role() const; int card_Role() const; void append_Role(C*); void erase_Role(C*); C * Role;</pre>

The {virtual} tagged value

The {virtual} tagged value specifies that the access methods for the association must be virtual.

Example:

The {virtual} tagged value on ...	is translated by ...
an association "Role" with the multiplicity "0-1" towards a class "C"	<pre>virtual C* get_Role() const; virtual int card_Role() const; virtual void append_Role(C*); virtual void erase_Role(C*); C * Role;</pre>

The {nocode} tagged value

The {nocode} tagged value prevents the generation of code for the association.

The {type} tagged value

The {type} tagged value is used to enter the type of collection used if it exists in the predefined types.

If you are familiar with the *Objectteering/C++* module, you can modify or create a new type interpreting project and add new set implementations. This tagged value is used to generate the association with the new implementation.

Please refer to chapter 14 of this user guide for a detailed description of adaptation and parameterization.

The {create} tagged value

The {create} tagged value generates the initialization of the member variable corresponding to the association in the constructor's initializer-list (instead of in the constructor's body).

The {instanceHandling} tagged value

The {instanceHandling} tagged value indicates that the association is an instance handling for the class. Please refer to the "*Handling instances*" section for how to model an instance handling association.

This tagged value allows the user to choose the name of an instance handling association, other than "instance", which is the default name.

The {noconst} tagged value

By default, the *Objectteering/C++* module generates "const" for some parameters of the association's access methods. It also generates "const" for read only access methods.

The {noconst} tagged value prevents the generation of "const".

The {C++Name} tagged value

The parameter of the {C++Name} tagged value takes precedence over the modeling name during the generation phase.

The {own} tagged value

The {own} tagged value does not generate access methods. The user can declare and define his own accessors using the "C++AccessDecl" and "C++AccessDef" notes.

The {noInline} tagged value

The {noInline} tagged value is used not to generate accessors as being "inline". Their body is generated in the body (.cxx) instead of the header (.hxx).

The {mutable} tagged value

The {mutable} tagged value is used to generate the association as a "mutable" member data.

Chapter 12: Generating code for an operation

Overview of code generation on an operation

Definition

Operations and parameters are the two main UML model elements used to generate C++ model functions.

An operation corresponds to a member function in a C++ class. The generation of an operation requires a C++ programmer to:

- ◆ declare the prototype of a member function
- ◆ define a member function
- ◆ generate the inclusion

This is done automatically by *Objectteering/C++*.

Different kinds of member functions

A member function in C++ may be a constructor, a destructor, a conversion operator or an operator for the owner class and a method. The name of the UML operation determines which kind of member function should be generated. They are reviewed in the following table:

The ... UML operation	generates a C++ member function
create	constructor
delete	destructor
operator associated with a type name (for example "operator double()")	conversion operator <code>operator double()</code>
operator and a C++ operator symbol for example the name "operator++()"	operator <code>int operator ++()</code>
myMethod	<code>void myMethod ()</code>

Note: The difference between a conversion operator and an operator is that a conversion operator does not have a return in the declaration, but must have a return in the member function's definition.

Operation

Information entered in the "Operation" dialog box will be interpreted, in order to generate all cases of function members in C++ (virtual, pure virtual, static, etc).

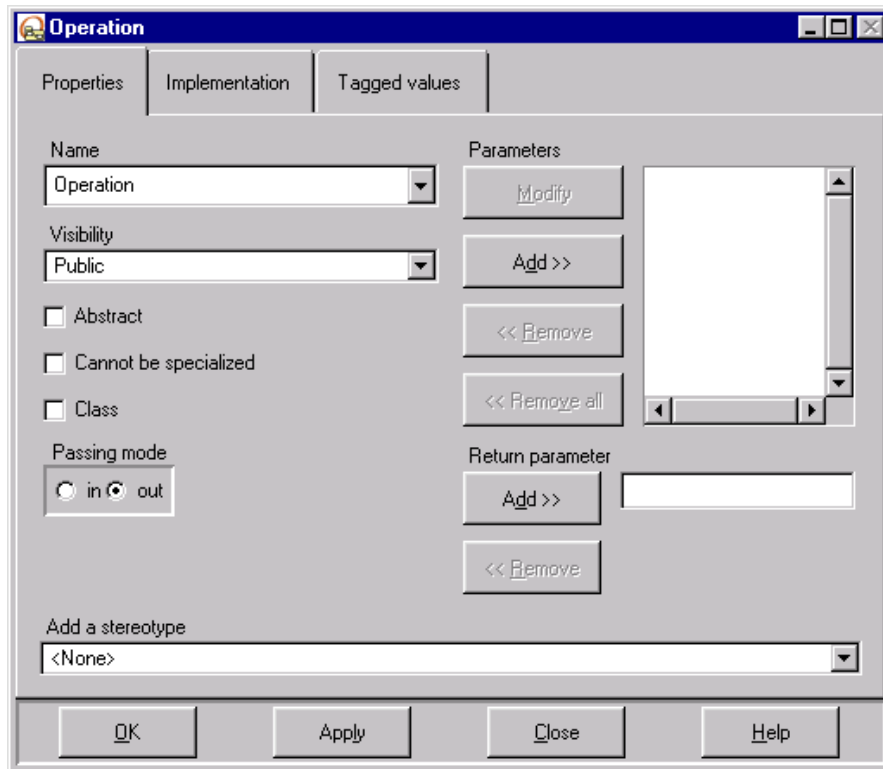


Figure 12-1. The "Operation" dialog box

Declaration (abstract, static, inline, in/out)

Three fields in the dialog box and two tagged values of an operation, {inline} and {virtual}, are taken into account for generating the declaration of a member function.

Checking the box or annotating with the ... tagged value	generates a ... function
{Abstract}	virtual pure
{Class}	static
{in}	const
{inline}	inline
{virtual}	virtual

Tagged values on an operation

The C++ specific tagged values available for an operation are:

The ... tagged value	Role ...
{C++Name}	Name to be taken into account to generate operation name.
{inline}	Generation of a C++ "inline" method.
{nocode}	No code generation for the method.
{noconst}	No "const " string for the method.
{noinvariant}}	Inhibiting the call for an invariant.
{virtual}	Generation of a "virtual" C++ method.

Notes on an operation

The C++ specific notes available for an operation are:

The ... note type	Role
C++	text to be inserted as the body of the C++ method
C++ConstructorTransmission	text to be inserted in the constructor's initializer-list
C++Returned	text to be inserted between "return " brackets.

Note: The "C++" note is the most widely used. The operation's processing is declared therein.

Use links on an operation

If the user models a dependency link from an operation to a class, and stereotypes this link <<throw>>, the following code is generated:

In the header file:

```
/*METHODS*/  
void f() throw (x1, x2, x3, x4);
```

In the body file:

```
void A::f() throw (x1, x2, x3, x4)  
{  
&  
}
```

Parameter

Definition

A parameter is used to represent both the parameter and the returned parameter of an operation.

The type of a parameter can be a *general class*, in other words:

- ◆ a basic type (int, real, char, string, etc)
- ◆ an enumerate
- ◆ a class

A parameter can also be a type that is unknown to the model, but defined in C++ by the programmer. In this case, you must use C++ specific notes ("*C++TypeExpr*" or "*C++ParamExpr*") for the parameter of an operation; and use "*C++TypeExpr*" for the returned parameter of an operation.

A parameter can be a single object or a set of objects of the same type.

Passing mode

In the model, the parameter is defined with a passing mode, in other words:

- ◆ IN: the content of the parameter will not be changed
- ◆ OUT: the content of the parameter can be changed
- ◆ INOUT: the content of the parameter can be changed

These passing modes are converted into C++ by Objectteering/UML, which takes into account implementation notions, such as pointers, references or passing values. The generation of passing mode strings can be invalidated by selecting the "*Generate the passing mode for parameters*" check box in the C++ module parameters (for further information on module parameters, please refer to chapter 7 of this user guide).

Default generation

For a parameter of the simple type (except for *string*, whose generation rule is the same as for class), its argument is transmitted by value in mode IN and by reference in mode INOUT or OUT.

For a parameter with a class or a set type, its argument is always transmitted through reference. Indeed, it is a C++ programming rule to avoid passing the argument by value when the type is a complex class. To protect the argument with the IN passing mode from being modified, the "const" string is generated.

Of course, you may change this default generation rule by annotating the parameter with one of the following tagged values

- ◆ {*} to generate a pointer argument
- ◆ {noconst} to inhibit the generation of the "const" string for the IN argument.

Tagged values on a parameter

C++ specific tagged values available for a parameter are:

The ... tagged value	Role ...
{*}	Passing by pointer
{&}	Passing by reference
{array}	Set of "array" type
{bind}	Definition of the instantiation parameters of a template
{C++Name}	Name to be taken into account to generate parameter name
{const}	Generation of the "const" keyword
{long}	Long integer or double real
{noconst}	Non generation of the "const" keyword
{short}	Short integer
{type}	Generating a special implementation
{unsigned}	Unsigned integer
{val}	Passing by value

Notes on a parameter

C++ specific notes available for a parameter are:

The ... note type	Role
C++DefaultValue	default value
C++ParamExpr	expression of the complete type (including the parameter name)
C++TypeExpr	expression of the type not including the parameter name

Detailed tagged values on an operation

The {C++Name} tagged value

The parameter of the {C++Name} tagged value takes precedence over the modeling name during the generation phase.

The {inline} tagged value

The {inline} tagged value generates the operation as an *inline* function member in the C++ sense.

The operation declaration is generated with the "inline" keyword. Its definition is inserted after the declaration of the class, in the header file.

The {nocode} tagged value

The {nocode} tagged value on an operation inhibits operation generation.

The {noconst} tagged value

This tagged value inhibits the generation of "const" string for the method, even if the mode has the IN mode.

The {noinvariant} tagged value

In the generated operation definition code, there are certain zones contained by the pair of `CR_CHECK` macros. By default, the invariant call and the pre-post conditions are inserted in the macro, to control the state of the class object and the conditions in a dynamic way.

```
#ifdef CR_CHECK
...
#endif
```

Pre or post conditions are put after the class and package invariant.

The {noinvariant} tagged value inhibits the call for the invariant for this operation. Pre and post-conditions, entered at the method level, are nonetheless verified.

The {virtual} tagged value

The {virtual} tagged value on a method specifies that the method must be virtual in the C++ sense.

Example:

The {virtual} tagged value on ...	is translated by ...
an "m" method without parameters	<code>virtual void m();</code>

Notes on an operation: Detailed description

Generation

Generation of the definition of a member function inserts some of the following C++ specific notes into the generated code, according to the category of the function.

- ◆ *C++ConstructorTransmission* in the initializer-list for the constructor
- ◆ *C++Returned*
- ◆ *C++*

The "C++ConstructorTransmission" note

The "*C++ConstructorTransmission*" note on a method is taken into account only if the method name is "create", meaning that it is a constructor.

The note allows you to carry out the call for the constructors of the basic class of the current class.

Example:

A " <i>C++ConstructorTransmission</i> " note containing...	is translated ...
: Cbase (2, 5)	in the body file, by: <pre> C::C(...) : Cbase (2,5) { ... } </pre>

The "C++Returned" note

The "*C++Returned*" note on an operation defines the return value of the operation and should enter the note only if the operation owns a return parameter.

For an operation which has a return parameter, "*return ()*" C++ code is always generated. The code in brackets is the content of the "*C++Returned*" note on the operation or the "*C++DefaultValue*" note on the return parameter. If none of these notes are present in the model, a marker allowing you to enter the code while editing the generated file is generated.

Example:

A " <i>C++Returned</i> " note containing ...	is translated ...
TRUE	at the end of the method definition, by: <pre>return (TRUE);</pre>

Note: The "return" instructions in the method's body are not advised. The "*C++Returned*" note should thus be the only exit point of the method. It is inserted after the invariant and the post-condition.

The "C++" note

This note contains the procedural instructions of the C++ program. The text is simply inserted between the { } of the generated member function.

Constraints on an operation: Detailed description


Constraints available on operations

The *Objectteering/C++* module provides two C++ specific constraints available for use with operations:

- ◆ *C++PreCondition*
- ◆ *C++PostCondition*

C++PreCondition

A constraint entered as a pre-condition stereotyped <<C++PreCondition>>, associated with an operation, represents the list of clauses which represent the pre-condition of the associated operation.

Entering a pre-condition constraint is done in the browser palette, using the  "Create a constraint" button. The stereotype is selected via the "Add a stereotype" field.


The syntax is the same as for class invariants.

Example:

Constraint stereotyped <<C++PreCondition>> containing ...	is translated ...
<pre>x_i > 0 y_i > 0</pre>	<p>in the body of the operation, before the implementation code section, by:</p> <pre>if (!(x_i > 0)) ::error(...) if (!(x_i > 0)) ::error(...)</pre>

C++PostCondition

A constraint entered as a pre-condition stereotyped <<C++PostCondition>>, associated with an operation, represents the list of the clauses, which represent the post-condition of the associated operation.

A pre-condition constraint is entered in the browser palette, using the  "Create a constraint" button. The stereotype is selected via the "Add a stereotype" field.

The syntax is the same as for class invariants.

Example:

Constraint stereotyped <<C++PostCondition>> containing ...	is translated ...
<pre>x_i == 0 y_i == 0</pre>	<p>in the body of the operation, after the implementation of the code section, by:</p> <pre>if (!(x_i > 0)) ::error(...) if (!(y_i == 0)) ::error(...)</pre>

Detailed tagged values on a parameter

The {&} tagged value

The {&} tagged value on a parameter is used to generate the parameter in the form of a reference.

Example:

The {&} tagged value on ...	is translated by ...
a parameter "p" with the integer type on a method "m"	<code>void m(int & p);</code>

The {*} tagged value

The {*} tagged value on a parameter allows you to generate it in the form of a pointer.

Example:

The {*} tagged value on ...	is translated by ...
a parameter "p" with the integer type on a method "m"	<code>void m(int * p);</code>

The {array} tagged value

The {array} tagged value on a parameter with the "set" type is used to generate this parameter in the form of an array.

Example:

The {array} tagged value on ...	is translated by ...
a parameter "p" with the integer set type on a method "m"	<code>const cr_array <int> & p</code>

The {C++Name} tagged value

The parameter of the {C++Name} tagged value takes precedence over the modeling name during the generation phase.

The {bind} tagged value

The {bind} tagged value indicates that the type of the parameter corresponds to a C++ template class. The parameters of this tagged value are the data allowing you to instantiate the "template" class.

Example:

The {bind} tagged value on ...	with the ... parameter	is translated by ...
a parameter "p" with a class "C" (generic itself) on a method "m"	anyClass 20	void m(C<anyClass,20>&p) ;

Please refer to the "*Generic class: Template*" section in chapter 8 of this user guide for the description of the template class generation.

The {const} tagged value

The {const} tagged value on a parameter allows you to generate the tagged value as a constant parameter.

If the corresponding option is chosen at Objectteering/UML parameterization level, the tagged value is useless, as all the "in" parameters are generated with the "const" keyword.

Example:

The {const} tagged value on ...	is translated by ...
a parameter "p" with the string type ("string") on a method "m"	void m (const CR_string & p) ;

The {long} tagged value

The {long} tagged value on a parameter with the integer type allows you to indicate that it has the "long int" type.

Example:

The {long} tagged value on ...	is translated by ...
a parameter "p" with the integer type on a method "m"	<code>void m (long p);</code>

The {noconst} tagged value

The {noconst} tagged value on a parameter inhibits the generation of "const" string for the parameter even if it has the passing mode IN.

Example:

The {noconst} tagged value on ...	is translated by ...
a parameter "p" of string type ("string") on a method "m"	<code>void m (CR_string & p);</code>

The {short} tagged value

The {short} tagged value on a parameter with the integer type is used to indicate that it has the "short int" type.

Example:

The {short} tagged value on ...	is translated by ...
a parameter "p" with the integer type on a method "m"	<code>void m (short p);</code>

The {type} tagged value

If you are familiar with the *Objectteering/C++* module, you can modify or create a new type interpreting project and add new set implementations. The tagged value allows you to generate the parameter with the new implementation.

Please refer to chapter 15 of this user guide for a detailed description of the adaptation and parameterization.

The {unsigned} tagged value

The {unsigned} tagged value on a parameter with the integer type is used to indicate that this parameter has the "unsigned int" type.

Example:

The {unsigned} tagged value on ...	is translated by ...
a parameter "p" with the integer type on a method "m"	<code>void m (unsigned p);</code>

The {val} tagged value

The {val} tagged value on a parameter indicates that it is passed by value and not by reference.

Notes on a parameter: Detailed description

C++DefaultValue

The "*C++DefaultValue*" note on a parameter presents the default value for this parameter. You can also define a default value for the parameter in the parameter's dialog box (not for the return parameter).

If the parameter owns both a default value in the field provided and a "*C++DefaultValue*" note, the note is used during C++ generation. In the case of a return parameter, the "*C++DefaultValue*" note is used if no return value has been defined on the associated method ("*C+Returned*" note on the method).

Example:

A " <i>C++DefaultValue</i> " note containing...	is translated ...
aValue on a method named "m" for a parameter "p" with the integer type	in the method declaration by: <code>void m(int p=aValue);</code>

C++TypeExpr

The "*C++TypeExpr*" note represents the declaration, in C++, of the parameter type.

It is useful for very tricky C++ parameter syntax cases, which cannot easily be included in UML.

If you define a "*C++TypeExpr*" note for a parameter, it is taken into account for generating the type of argument. The generator ignores anything entered in the parameter's dialog box concerning the type of the argument.

Example:

A " <i>C++TypeExpr</i> " note containing ...	is translated ...
<code>void **</code> on a method named "f", for a parameter named "by"	in the method declaration by: <code>void f(void ** by);</code>

Note: If a parameter owns both a "*C++TypeExpr*" and a "*C++ParamExpr*" note, the generation will stop, with an error message displayed in the console.

C++ParamExpr

The "*C++ParamExpr*" note allows you to enter the complete declaration, in C++, of the parameter type.

If you define a "*C++ParamExpr*" for a parameter, it is taken into account when generating the argument. The generator ignores anything entered in the parameter's dialog box concerning the type, the name and the default value of the argument.

Example:

A " <i>C++ParamExpr</i> " note containing ...	is translated ...
char ** p[20] on a method named "f", for a parameter named "by"	in the method declaration by: void f(char** by [20]);

Note 1: If a parameter owns both a "*C++TypeExpr*" and a "*C++ParamExpr*" note, the generation will stop, with an error message displayed on the console.

Note 2: You must not define the "*C++ParamExpr*" note on a return parameter.

Chapter 13: Generating code for a
generalization, an
enumerate and basic types

Introduction

Overview

This chapter describes C++ code generation for model elements which have not yet been presented in the previous chapters. These elements are:

- ◆ basic types
 - ◆ type creation
 - ◆ enumerates
 - ◆ generalization
 - ◆ object instance
-

Generating basic types

Definition

In the "*_predefinedTypes*" UML modeling project, there are several types: boolean, char, integer, real and string. These are basic types.

The type of an attribute, a parameter, or an instance (see the following section for the description of an instance) can be a basic type. The type can be selected in a text field present in the attribute, parameter or instance dialog boxes. The suggested list contains these basic types. If you add a new type in the "*_predefinedTypes*" UML modeling project, it will appear in the suggested list.

Default correspondence

By default, the following correspondence is defined. The types in italics come from the basic types UML modeling project, such as *MFCTypes*, *ObjecteeringTypes* or *STLTypes*.

Objecteering/UML type	ObjecteeringTypes	STLTypes	MFCTypes
integer	int	int	int
char	char	char	char
boolean	<i>CR_boolean</i>	bool	<i>Bool</i>
real	float	float	float
string	<i>CR_String</i>	string	<i>CString</i>

Tagged values


It is possible to adapt this default correspondence to the tagged values, as follows:

Type	The ... tagged value	Default correspondence ...
char	{unsigned}	unsigned char
integer	{long}	long
integer	{short}	short
integer	{unsigned}	unsigned
integer	{unsigned} {long}	unsigned long
integer	{unsigned} {short}	unsigned short
real	{long}	double

These tagged values are available on attributes and parameters.

Generating a new type

Overview

While modeling your application in the Explorer, you can create a new type (*Type*) on the model elements *Package* or *Class*, by selecting the  "Add a note" icon.

The new type defined on a package is visible for all the classes contained in the package. The new type defined on a class can be visualized by the class and its derived classes.

Here *visible* means that the new type appears in the list of types of the attribute's dialog box of an attribute, parameter or instance.

The "C++" note on a Type

The "C++" note on a type contains the complete code used to declare the type in C++ (declaration of the "typedef").


Example:

A "C++" note containing ...	is translated ...
typedef char ** PCHAR; for a type named "myType"	in the header file of the corresponding class or package, by: typedef char ** PCHAR;

Generating an enumerate

Overview

While modeling your application with the explorer, you may create an enumerate corresponding to the *enum* in C++, on the *Package* or *Class* model elements

using the icon .

The enumerate type defined on a package can be visualized by all the classes contained in the package. The enumerate type defined on a class can be visualized by the class and its derived classes.

In this context, *visible* means that the enumerate type appears in the list of types in the attribute, parameter or instance dialog boxes.

The enumeration type dialog box

Enter an enumerate as shown in figure 13-1:

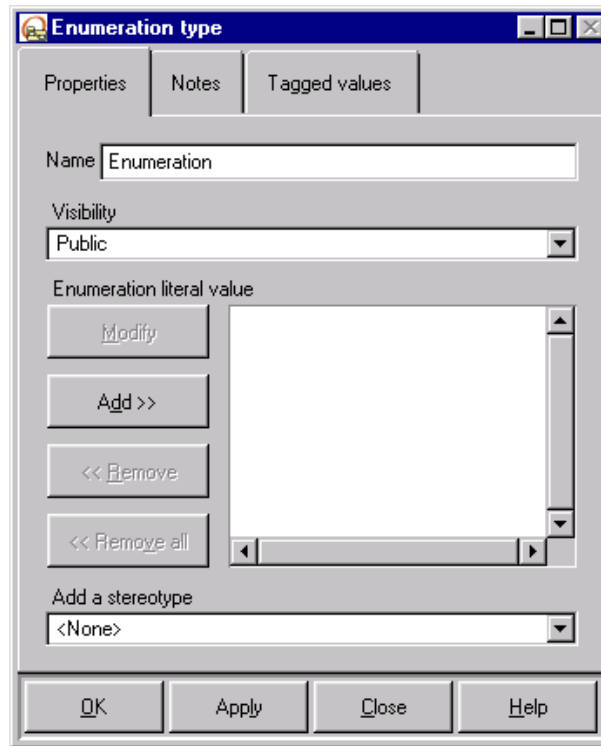


Figure 13-1. The "Enumeration type" dialog box

The generated code is as follows:

```
enum colors { red=10, green, blue } ;
```

Generating a generalization

Definition

One package can specialize another package; one class can specialize another class.

By default, generalization is generated as "public".

Example:

```
class A : public B { ... };
```

By annotating the generalization with the tagged values, you can change this default generation rule.

If the base class is a template class, use {bind} to instantiate the template class.

The {bind} tagged value

When a class specializes a template class, you must annotate the generalization using the {bind} tagged value. The parameters of this tagged value are the data used to instantiate the template class.

Example:

The { <i>bind</i> } tagged value on ...	with the parameter...	is translated by ...
a generalization of "A" towards "B" ("B" being itself a generic class)	anyClass, 20	the following declaration: class A : public B<anyClass,20>{ ... };

Refer to the "*Generic class: Template*" section in chapter 9 of this user guide for a description of template class generation.

You must not annotate a generalization between packages with this tagged value.

The {private} tagged value

The {private} tagged value on a generalization specifies that the generalization must be private in the C++ sense.

Example:

The { <i>private</i> } tagged value on ...	is translated by ...
a generalization of "A" towards "B"	class A : private B{ ... }

The {protected} tagged value

The {protected} tagged value on a generalization specifies that the generalization must be protected in the C++ sense.

Example:

The { <i>protected</i> } tagged value on ...	is translated by ...
a generalization of "A" towards "B"	class A : protected B{ ... }

The {virtual} tagged value

The {virtual} tagged value on a generalization specifies that the generalization must be virtual in the C++ sense.

Example:

The { <i>virtual</i> } tagged value on ...	is translated by ...
a generalization of "A" towards "B"	class A : public virtual B{ ... }

Creating an instance

Definition

While modeling in the explorer, you may create objects corresponding to global variables, in the C++ sense of the term. An object instantiation can be declared

using the  "Create an instance" icon.

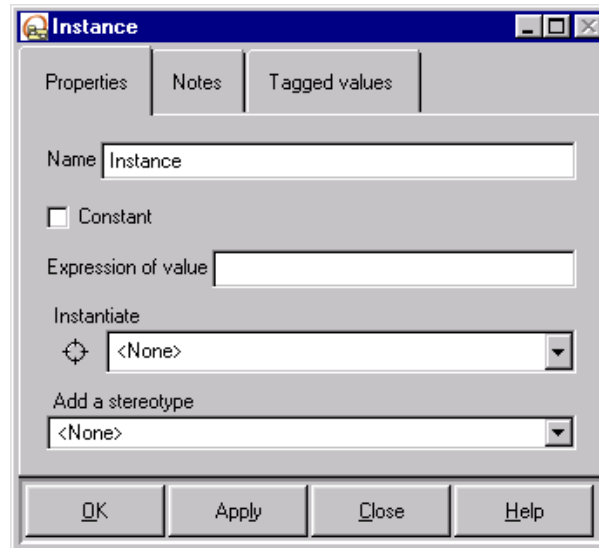


Figure 13-2. The "Instance" dialog box

Note: In order to carry out generation from instances, the "Generate the code for instances" tickbox in the "Generation options" parameter set must be checked at module parameter configuration level. For further details on this module parameter, please refer to the "Detailed description of parameters" section in chapter 7 of this user guide.

C++TypeExpr

The "*C++TypeExpr*" note on an instance is used to declare the instance's type.

If you define a "*C++TypeExpr*" note on an instance, it is taken into account during generation of the type of the instance. The generator ignores anything entered in the instance's dialog box concerning the type of the instance.

The use of "*C++TypeExpr*" must be limited to the complex cases of declaration, which are impossible to express in a different way.

Example:

A " <i>C++TypeExpr</i> " note containing...	is translated ...
<code>void **</code> on an instance named "Inst"	in the header file, to declare the instance, with: <code>void ** Inst;</code>

C++Value

The "*C++Value*" note on an instance is used to declare an initialization value for this instance.

If a "*C++Value*" note is defined on an instance, it is taken into account for the generation. The generator ignores anything entered in the instance's dialog box concerning the initialization value.

Example:

A " <i>C++Value</i> " note containing...	is translated ...
<code>sizeof(T)</code> on an instance named "Inst" of "integer" type	in the header file, to declare the instance, with: <code>int Inst = sizeof(T);</code>

The {bind} tagged value

The {bind} tagged value indicates that the type of the parameter corresponds to a C++ template class. The parameters of this tagged value are the data used to instantiate the "template" class.

Example:

The {bind} tagged value on ...	with the parameter...	is translated by ...
an instance "p" of a class "C" (generic itself)	anyClass 20	C<anyClass, 20> p;

Please refer to the "*Generic class: Template*" section in chapter 9 of this user guide for a description of the template class generation.

Chapter 14: Parameterizing basic types

Parameterizing basic types

Overview

The C++ code generator provides the possibility of using the basic types library of your choice in the generated code. These basic types can be defined outside the generator using a typed package. The generator can be broken down into two parts: one part as a basic generator in charge of producing the structure of the C++ code files and the other part in the form of a package (e.g. `ObjecteeringTypes`) in charge of generating C++ code according to the chosen basic types. This code is scanned by the basic generator for insertion in the generated files. This association is defined with the *"Name of the types interpretation project"* parameter.

All the basic types packages are defined in the *"predefinedTypes"* UML modeling project.

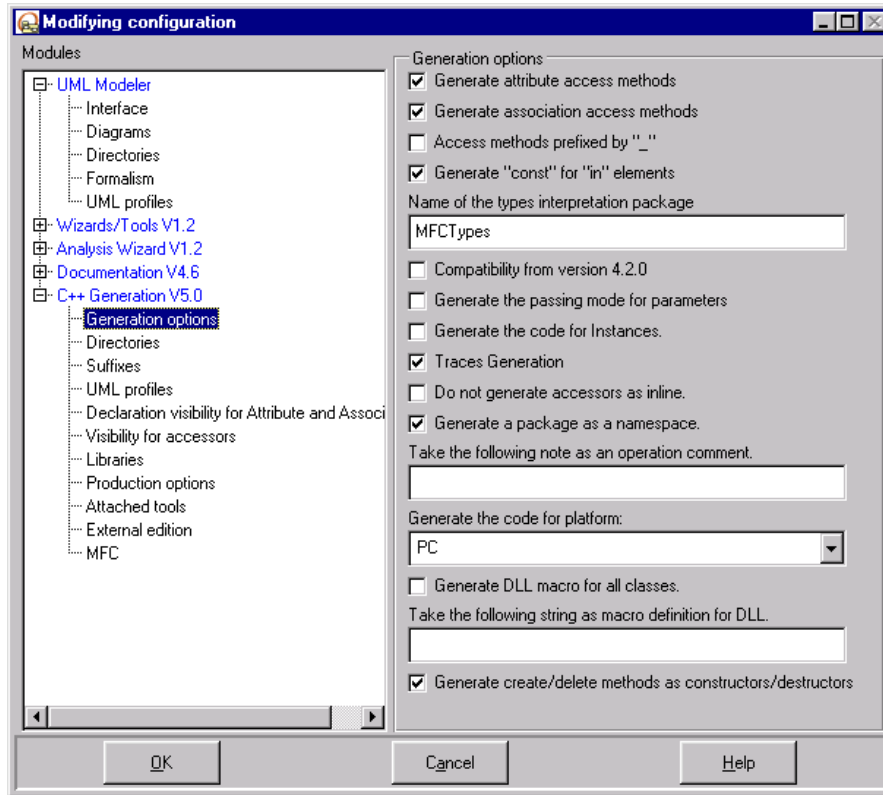


Figure 14-1. Dialog box for selecting a type package

Packages for translating types and generating accessors

The *Objectteering/C++* module delivery includes the "*ObjectteeringTypes*", "*STLTypes*" and "*MFCTypes*" packages defined in the "*predefinedTypes*" UML modeling project, which are type translation and accessor generation packages. These packages can be modified. Furthermore, it is possible to create another type translation and accessor generation package.

The "*ObjectteeringTypes*" package groups together types developed by Objectteering Software and which are absent from the C++ language. The sources of these basic types are delivered with *Objectteering/UML*, and are presented in chapter 18.

The "*STLTypes*" package groups one part of the types, defined in the C++ Standard *Template Library*, where the container types are stored. This library has the *ANSI/ISO* norm and is part of the C++ language definition. Only a recent version of C++ compiler supports STL.

The "*MFC Types*" package is available only the platform PC it uses the classes of MFC as type translation and accessor generation.

Content

A type package supports:

- ◆ the translation of model element base type
 - ◆ the generation of attribute(s) owned by a class; the code concerns the declaration and its accessors
 - ◆ the generation of associations, which exist between the classes, the declaration and their accessors
 - ◆ the generation of operation parameters
-

Package structure

To work on the type package, edit the "*_predefinedType*" UML modeling project. This UML modeling project must reference the *TypesEditor* module.

In the UML modeling project, you will find several type packages with a similar structure.

Each package contains:

- ◆ BaseTypes to give the translation of the basic type, such as int or string
- ◆ DefaultTranslations to guide default generation
- ◆ TranslationClasses to define every element used by the basic generator

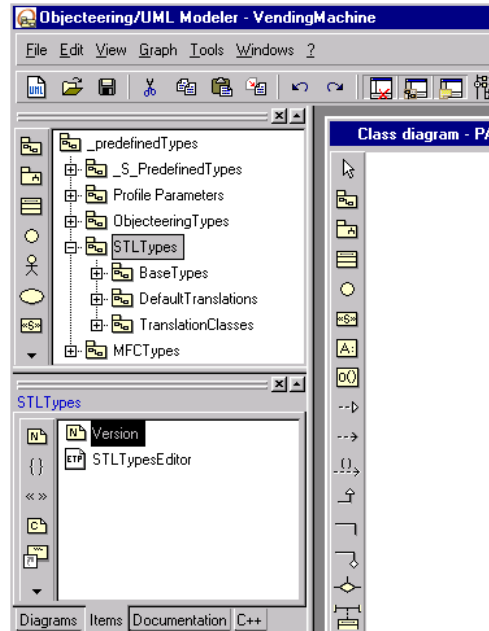


Figure 14-2. The "*STLTypes*" package in the "*_predefinedType*" UML modeling project

BaseTypes

The "BaseTypes" package is used for translating the basic types declared in the "_S_predefinedTypes", for example *integer* and *string*. As it is not possible to create a type with exactly the same name as a UML modeling project type, the aim is to look for a type in this package, whose name contains the name of the modeled type.

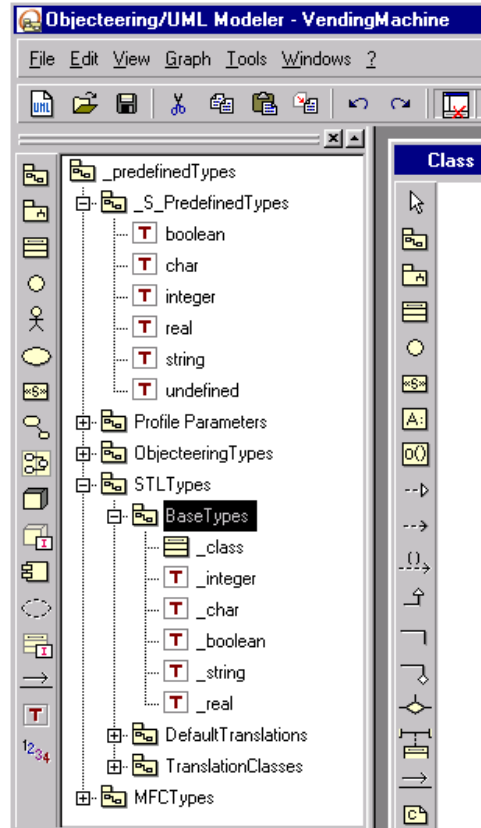


Figure 14-3. The "BaseTypes" package

On this associated type, the aim is to look for its *targetType* text. If this text does not carry the {Jeval} tagged value, then the C++ type is the content of the text itself. If it carries this tagged value, the type is the J evaluation of the text content. The text is then evaluated in the context of the modeled class.

A communication protocol exists in the generator. The C++ *targetType* must be placed in the *hContent of J type "string"* variable.

Example: Translation of the modeled *boolean* type for *ObjecteeringTypes*.

The translation of this type is given by the *boolean* type. Its *targetType* text does not carry the {Jeval} tagged value. The C++ type is, therefore, the content of the text itself, i.e. *CR_boolean*.

In this package, there are some notes associated with *_class* and the package itself, which indicate the elements to be inserted concerning the generation for every class and every package.

Package

The ... note	to generate ...
hxxInclude	include to be inserted into the header file of each package
namespace	to insert using namespace in the header file of the package

Class

The ... note	to generate the code for ...
hxxInclude	the include to be inserted into the header file
cxx Include	the include to be inserted into the body file
mainInstance	the main class instantiation (used only by <i>"ObjecteeringTypes"</i>)
mainInclude	the include to inserted into the main class (used only by <i>"ObjecteeringTypes"</i>)
mainInheritance	the inheritance for main class (used only by <i>"ObjecteeringTypes"</i>)

Translating set type associations, parameters and attributes

The classes in the package support the generation of declarations and accessors for associations and attributes, and the code concerning the parameters. When the generator has to process these modeling cases, it firstly determines the class that will be used to support the translation of the type, the declarations and its features. The zones of code for inserting into the generated C++ file are then obtained, according to the type of texts attached to this class.

By default, the link between a modeling case such as associations, attributes or parameters and the class defined in the "*TranslationClasses*" follows a naming rule to be presented below.

It is possible to indicate explicitly the type to be used for the translation using the {type} tagged value.

When this tagged value is present, the code generator takes the name of the class passed on as the parameter for "*TranslationClass*".

It is also possible to specify a translation class contained in a type package different from that chosen in the current parameter of the module, "*Name of the types interpreting package*".

Example for an association annotated with a {type(<STLTypes:vector>)} tagged value

In this case, whatever the value of the "*Types interpreting project name*" parameter, the generator will load the *STLTypes* UML modeling project to search for the *Vector* class (ie *vectorAssociation*).

Then, according to the existence of the tagged value {Jeval}, it will assess or copy the content of the texts attached to it to obtain the C++ code to be generated.

Translation class naming rules

The name of class must be the type name, which is to be used as the parameter of {type}, suffixed by *Attribute*, *Association*, *IOPParameter* or *ReturnParameter*.

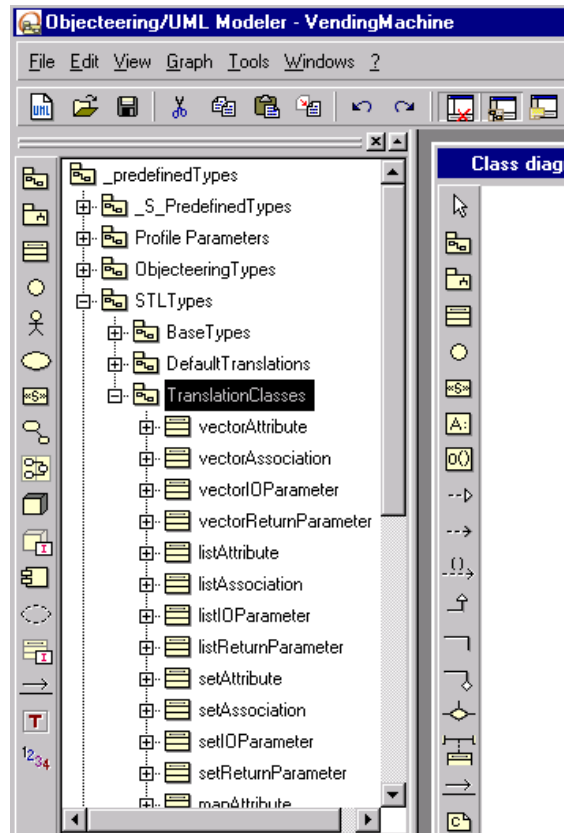


Figure 14-4. The "TranslationClasses" package

The translation class typeAttribute or typeAssociation

The content of a translation class for Attribute or Association must include:

- ◆ a method call "*declare*"
- ◆ accessor methods, which are grouped into access, modified or fullaccess methods indicated by {access}, {modify}, {fullaccess} or {fullmodify}
- ◆ notes associated with the class, containing the rule to generate the code needed, such as "*hxxInclude*", "*cxxInclude*", "*type*"

Notes associated with a translation class

The ... note	to generate the code for ...
type	the type of attribute or association
typedef	the typedef in public part of the class, to simplify the use of type.
hxxInclude	the include to be inserted into header file
cxxInclude	the include to be inserted into body file
forward	the forward declaration needed
setInstanciation	the set of instantiation (used only by " <i>ObjectteeringTypes</i> ")
cardMethod	the clause used for invariants concerning the multiplicity control
appendMethod	the code inserted into constructor, needed by class instance handling (only for Association)
eraseMethod	the code inserted into destructor, needed by class instance handling (only for Association)

Chapter 14: Parameterizing basic types

The J note on different elements of a method of the translation class allows you to define the code which is to be inserted. They are summed up in the following table.

On the ... element	to generate the code for ...
method "declare"	the declaration of the Attribute and Association
accessor method	the content of the method
parameter	the type of the parameter

Each accessor method has a Jname note to define its name.

The translation class type `IOParameter` or `typeReturnParameter`

The content of a translation class for an operation's parameter or its return parameter must include:

- ◆ a method called "*declare*"
- ◆ notes associated with the class, containing the rule to generate the code needed

Notes for a translation class

Note ...	to generate the code for ...
type	the type of parameter
hxxInclude	the include to be inserted into header file
cxxInclude	the include to be inserted into body file
forward	the forward declaration
setInstantiation	the set of instantiation (used only by " <i>ObjectteeringTypes</i> ").

DefaultTranslations

This package contains a group of packages, whose names conform to the naming rule used by the basic generator. Each package refers to a translation class that is used to generate the code.

If there are no tagged values on the modeling element, the basic generator finds the translation class through the "*DefaultTranslations*" package.

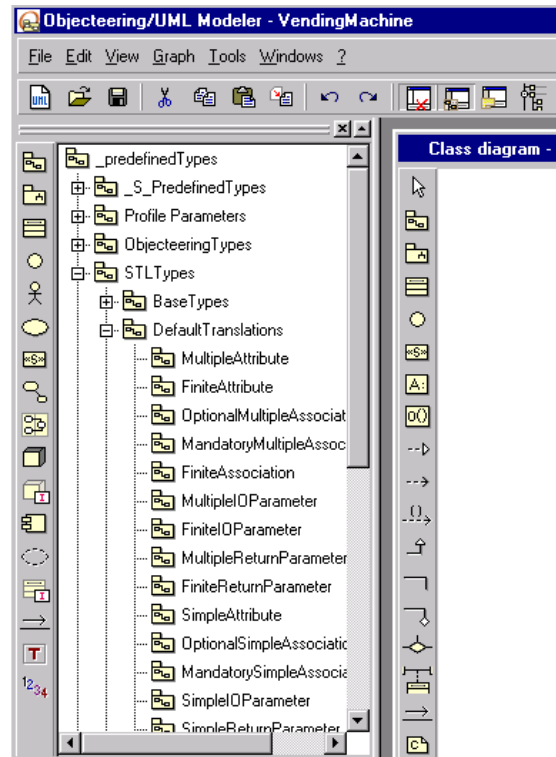


Figure 14-5. The "*DefaultTranslations*" package

Name ...	Default generation of ...
SimpleAttribute	declaration and accessors for a size 1 attribute
FiniteAttribute	declaration and accessors for a size n>1 attribute
MultipleAttribute	declaration and accessors for a size* attribute
MandatorySimpleAssociation	declaration and accessors for an association with the 1 multiplicity
OptionalSimpleAssociation	declaration and accessors for an association with the 0-1 multiplicity
FiniteAssociation	declaration and accessors for an association with the 0-n, n multiplicity value >1
MandatoryMultipleAssociation	declaration and accessors for an association with the multiplicity1-*
OptionalMultipleAssociation	declaration and accessors for an association with the multiplicity0-*
SimpleIOParameter	declaration for a parameter with size 1
FiniteIOParameter	declaration for a parameter with size n>1
MultipleIOParameter	declaration for a parameter with size *
SimpleReturnParameter	declaration for a return parameter with size 1
FiniteReturnParameter	declaration for a return parameter with size n>1
MultipleReturnParameter	declaration for a return parameter with size *

Note: Classes that represent the model cases can reference the same type class. In other words, the same type can be used for an association with 0-1 or 0-* multiplicity.

Chapter 15: Adapting C++ code
generation

Overview of C++ generation adaptation

Overview

In order to parameterize C++ code generation in a more highly defined way than is possible through the module parameters, you must have the *Objectteering/UML Profile Builder* module in the *Objectteering/Enterprise* edition. The J language allows generation parameterization. This parameterization is carried out:

- ◆ by creating a child "*default#external#Code#Cxx*" UML profile
- ◆ by redefining J methods accessible from the *Objectteering/C++* module

For each type of model element (metaclass), one or more methods can be adapted according to your needs. Parameterization can be carried out to a very fine degree. For example, it is possible to modify the generation of the instance management without changing the generation of the associations.

Possible adaptations

The Objectteering C++ code generation can be adapted by:

- ◆ defining specific new notes (code, pre-condition, etc.)
- ◆ adding new project parameters
- ◆ specialization of some generation behavior (J methods)
- ◆ adding new commands (entries, menu) for the generator
- ◆ defining the basic types project to be used in C++ code generation

The first three services are described in detail in the *Objectteering/UML Profile Builder* module.

The creation and modification of the type package are described in chapter 14 of this user guide ("*Parameterizing basic types*").

The "C++" UML profile

The "default#external#Code#Cxx" UML profile supplies all the J methods that are accessible to the user.

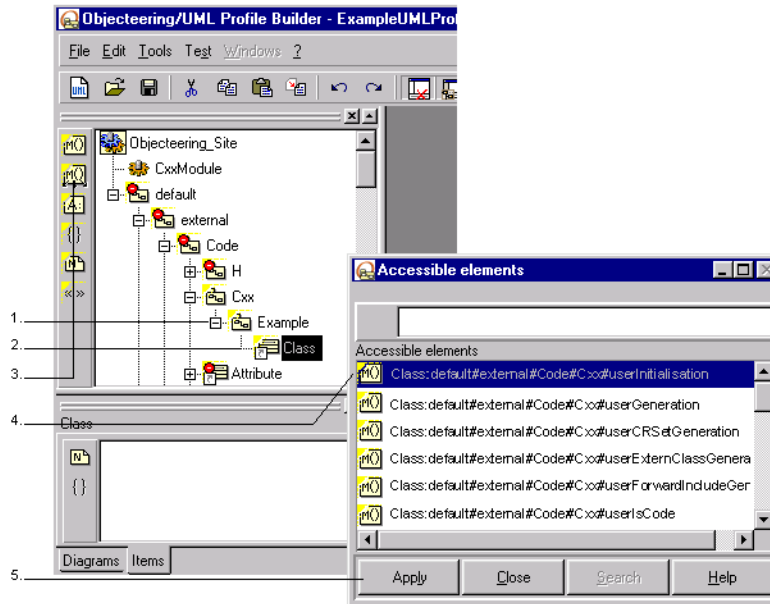


Figure 15-1. Access to the methods that can be parameterized

Steps:

- 1 - Create a child UML profile under the Cxx generation UML profile.
- 2 - Reference the metaclasses you wish to define.
- 3 - Redefine a parent J method.
- 4 - Indicate the method that is going to be redefined.
- 5 - Confirm.

Procedure for adapting the C++ code generation module

- 1 - Edit a UML profiling project.
- 2 - Import the standard *Objectteering/C++* module.
- 3 - Create a child UML profile in "*default##external##Code##Cxx*".
- 4 - Create in this child UML profile redefined J methods, tagged values, constraints and notes of your choice to be used when redefining the methods.
- 5 - Create a module.
- 6 - Make this module specialize the C++ module.
- 7 - Reference the new UML profile from this new module.

Procedure for using parameterization in a UML modeling project

- 1 - Install the new module.
 - 2 - Select the module. (For further information, please refer to the "*Selecting modules in the current UML modeling project*" section in chapter 3 of the *Objectteering/Introduction* user guide).
-

Generation constant

J methods

C++ code generation initializes, when it is run, a set of constants used later. At the end of this initialization phase, a user method is activated to allow the modification of these generation constants. The methods activated according to the class or package units are:

```
Package:userInitialization() return boolean;  
Class:userInitialization() return boolean;
```

Example:

```
boolean Package:default#Code#Cxx#User#userInitialization ()  
{  
  
    // redefining the name of the "invariant" methods  
    ClassInvariantCheckMethodName = "_MyInvariant";  
    SchemaInvariantCheckMethodName = "_MyInvariant";  
    return=true;  
}
```

All the constants are attributes defined on the "*object*" metaclass , with the "*string*" type.

Naming and filtering constants

Overview

Naming constants allow the modification of the names of the methods generated automatically by Objectteering/UML. Constants and their default values are presented in the following tables.

The name of "invariant" methods

The name of the methods for calculating invariants can be parameterized by redefining constants.

The ... constant	defines ...	by default
SchemaInvariantCheckMethodName	the name of the method that implements the class invariant	"_invariant"
ClassInvariantCheckMethodName	the name of the method that implements the package invariant	"_invariant"

Prefix of the accessors to attributes and associations

Code generation masks the implementation of relations and methods, through accessors. The names of the accessors are deduced from those of roles or attributes, by concatenating prefixes that can be parameterized.

The ... constant	defines ...	by default ...
AccessMethodPrefix	the access methods' prefix.	"_"
SetMethodPrefix	the allocation methods' prefix.	"set_"
AppendMethodPrefix	the prefix of the methods for adding in a set	"append_"
EraseMethodPrefix	the prefix of the methods for deleting in a set	"erase_"
CardinalMethodPrefix	the prefix of the methods that return the multiplicity	"card_"
FullAccessMethodPrefix	the prefix of the methods that return the sets	"get_all_"
FullSetMethodPrefix	the prefix of the methods that allocate the sets	"set_all_"
AttributePrefix	the prefix of the C++ attributes for the generated attributes and relations.	""

Basic types name

The ... constant	defines ...	by default ...
CxxString	the C++ type corresponding to the string type.	"CR_string"
CxxChar	the C++ type corresponding to the char type.	"char"
CxxBoolean	the C++ type corresponding to the boolean type.	"CR_boolean"
CxxInteger	the C++ type corresponding to the integer type.	"int"
CxxReal	the C++ type corresponding to the real type.	"float"

Name of the generated suffix files

The generated C++ files can be called according to the targets and preferences:

- ◆ for the body: ".cxx", ".cpp", ".c"
- ◆ for the interface: ".hxx", ".hpp", ".h"

The ... constant	defines ...	by default ...
CxxFileSuffix	the suffix of the C++ body file	".cxx"
HxxFileSuffix	the suffix of the produced interface file.	".hxx"

All the constants in this table have the "*string*" type.

Name of the comment zones

Different comments are inserted in the generated C++ code files. These can be modified by re-defining the following variables:

The ... constant	Defined on the ... metaclass	is used to redefine the comment ...	with the ... default value
InterfaceBeginningComment	Class	located before the declaration of the class in the header file.	/*INTERFACE OF THE xx CLASS*/
InterfaceEndComment	Class	located after the declaration of the class in the header file.	/*END INTERFACE OF THE xx CLASS*/
BodyBeginningComment	Class	located before the class definition in the generated code file.	/*BODY OF xx CLASS*/
BodyEndComment	Class	located before the class definition in the generated code file.	/*END OF xx CLASS BODY*/
EnumerateDeclarationsComment	Class	located before the declaration of an enumerate	/*ENUMERATE DECLARATIONS*/
ConstantDefinitionsComment	Class	located before the definition of the C++ constants.	/*CONSTANTS DEFINITIONS*/
FreeCodeComment	Class	located before the insertion of the "CxxInterfaceHeader" type notes	/*FREE C++ PROGRAMMER'S CODE*/
ForwardClassDeclarationsComment	Class	located before the "forward" declarations	/*FORWARD CLASS DECLARATION*/
SetInstantiationsComment	Class	located before the instantiation of the CR_SET macro	/*INSTANCE SET DECLARATION */
TypeDeclarationsComment	Class	located before the declaration of the types	/*TYPE DECLARATIONS*/

The ... constant	Defined on the ... metaclass	is used to redefine the comment ...	with the ... default value
MethodDeclarationsComment	Class	located before the declaration of the class methods.	<i>/* METHODS*/</i>
MethodComment	Class	located before the definition of each class method.	<i>/*METHOD xx OF yy CLASS*/</i>
RelationshipDeclarationsComment	Class	located before the declaration of the implementation of the class associations.	<i>/*RELATIONSHIP*/</i>
AttributeDeclarationsComment	Class	located before the declaration of the class attributes.	<i>/*ATTRIBUTE*/</i>
InvariantDeclarationsComment	Class	located before the declaration of an invariant	<i>/*INVARIANT*/</i>
InvariantDefinitionsComment	Class	located before the definition of an invariant	<i>/*INVARIANT OF THE xx CLASS*/</i>
InterfaceBeginningComment	Package	located before the declaration of the class associated to a package in the header file.	<i>/*INTERFACE OF THE xx PACKAGE</i>
InterfaceEndComment	Package	located after the declaration of the class associated to a package in the header file.	<i>/*END INTERFACE OF THE xx PACKAGE*/</i>
BodyBeginningComment	Package	located before the definition of the class associated to a package in the generated body file.	<i>/*BODY OF xx PACKAGE*/</i>
BodyEndComment	Package	located after the definition of the class associated to a package in the generated body file.	<i>/*END OF xx PACKAGE*/</i>

The ... constant	Defined on the ... metaclass	is used to redefine the comment ...	with the ... default value
InheritedSchemaIncludesComment	Package	located before the inclusion of the header files of the parent packages.	/*INCLUDES OF INHERITED PACKAGE*/
FreeCodeComment	Package	located before the insertion of the "CxxInterfaceHeader" type notes	/*FREE C++ PROGRAMMER'S CODE*/
ConstantDefinitionsComment	Package	located before the definition of C++ constants.	/*CONSTANTS DEFINITIONS*/
EnumerateDeclarationsComment	Package	located before the declaration of an enumerate	/*ENUMERATE DECLARATIONS*/
TypeDeclarationsComment	Package	located before the declaration of the types	/*TYPE DECLARATIONS*/
InvariantDefinitionComment	Package	located before the definition of an invariant	/*INVARIANT OF THE xx CLASS*/

Pre-compilation tagged values

Pre-compilation tagged values can be adapted by re-defining the following variables:

The ... constant	Defined on the ... metaclass	defines ...	by default ...
CxxObj_CR_Check	Object	the pre-compilation tagged value concerning the "Check" mode with the calling of the invariants.	#ifdef CR_CHECK
CxxObj_CR_Endif	Object	the end of a pre-compilation tagged value	#endif
CxxObj_CR_Trace	Object	the pre-compilation tagged value concerning the "Trace" mode in the adjustment phase.	#ifdef CR_TRACE

Size of the line indents in the generated C++ code

To make it more clear, the generated lines are indented, in order to put forward the scope of the variables. However, to respect a programming norm specific to the user, it is possible to modify the size of the indents by re-defining the following variables:

The ... constant	Defined on the ... metaclass	defines the indents ...
indentHeaderMember	Object	of the lines of code concerning the declaration of a class's members.
indentCode	Object	of the lines of code of a method's body.
indentClass	Object	of the lines of code of a class's declaration.
indentVisibility	Object	of the "private", "protected" and "public" keywords.
indentBodyMember	Object	of the lines of code of a method's declaration.
indentPreprocessor	Object	of the lines of code surrounded by a pre-compilation tagged value.
indentBeginEnd	Object	opening and closing curly brackets of a method's body.

Passing mode of operation parameters

The ...constant	Defined on the ... metaclass	defines ...	by default ...
CxxAtt_ConstReturnString	ModelElement	the const mode for a method's return parameter.	const
CxxAtt_RefReturnString	ModelElement	the passing mode for a method's return parameter.	&
CxxDCR_ConstMethodString	Attribute	the const mode for a read only access method	const
CxxDCR_IndexMethodString	Attribute	the string passed as the "set" type access methods' parameter	(cr_index)

Parameterizing the generation of different units

Mechanism

When a unit (class, operation, attribute, parameter, etc.) has finished being generated, one or more methods are then activated with the C++ translations of the characteristics specific to the unit. Each parameterization method linked to the object to be generated (the C++ class, operation, etc.), and all the information necessary for generation are immediately accessible. For example, the J parameterization method "*UserGeneration*" on a parameter will simply obtain the name of the parameter through the "*Name*" attribute of the current parameter.

Return value

The parameterization method returns a "true" value, if no processing error occurred. If a "false" value is returned, code generation is stopped. This allows possible errors to be diagnosed, depending on what has been added by generation.

Global parameterization

Certain methods of parameterization provide parameters, which allow the complete overloading of the C++ generation specific to the current unit. C++ code generation then checks that the parameter is not empty, and replaces its generation with the one produced by the parameterization.

Note: In the following sections, the J methods flagged with * are not called when the "*Ascending compatibility since version 4.2.0*" parameter is not selected.

Parameterization is made possible by the types package.

Package

Parameterization

At the end of the generation of a package, the parameterization method is run with the C++ translations of the different package characteristics:

- ◆ expression tagged value at the beginning of the interface file
- ◆ public and private members
- ◆ expression tagged value at the end of the interface file
- ◆ expression tagged value at the beginning of the body file
- ◆ package body

When the method is returned, the formatting of the "*interface*" and "*body*" files is carried out, taking into account the possible modifications of the C++ code.

Set of generated code zones

The signature of the parameterization method is as follows:

```
Package:userGeneration (
    interfaceHeader_io : inout string,
    publicMembers_io   : inout string,
    protectedMembers_io : inout string,
    privateMembers_io  : inout string,
    interfaceBottom_io : inout string,
    bodyHeader_io       : inout string,
    schemaBody_io       : inout string ) return boolean
```

The ... parameter	contains in C++ ...
interfaceHeader_io	the code at the beginning of the interface file.
publicMembers_io	the code of the public members.
protectedMembers_io	the code of the protected members.
privateMembers_io	the code of the private members.
interfaceBottom_io	the code at the end of the interface file.
bodyHeader_io	the code at the beginning of the body file.
packageBody_io	the code in the body file and deduced from the package's characteristics.

Initialization before generating code

The signature of the parameterization method is as follows:

```
Package:userInitialisation () return boolean
```

The user can use this method to update the variables documented in the previous section so they be taken into account in the generated code. Preliminary processing, such as consistency checks specific to the user, may be inserted into the body of this method.

Producing code

The signature of the parameterization method is as follows:

```
Package:userIsCode () return boolean
```

When the method returns FALSE, code is not generated for the current package.

Class

Parameterizing

After a class is generated, the parameterization method is launched, with the C++ translations of the different class characteristics:

- ◆ expression tagged values at the beginning of the interface file
- ◆ class generalization
- ◆ public, protected and private members
- ◆ expression tagged value at the end of the interface file
- ◆ expression tagged value at the beginning of the body file
- ◆ class body

When the method is returned, the formatting of the "*interface*" and "*body*" files is carried out, taking into account the modifications that may have been made to the C++ code.

Set of generated code zones

The signature of the parameterization method is as follows:

```
Class:userGeneration (
  interfaceInclude_io : inout string,
  interfaceHeader_io : inout string,
  inheritedClasses_io : inout string,
  publicMembers_io : inout string,
  protectedMembers_io : inout string,
  privateMembers_io : inout string,
  inlineMethods_io : inout string,
  interfaceBottom_io : inout string,
  bodyIncludes_io : inout string,
  bodyHeader_io : inout string,
  classBody_io : inout string ) return boolean
```

The ... parameter	contains in C++ the code ...
interfaceIncludes_io	inclusion of the classes in the interface file
interfaceHeader_io	at the beginning of the interface file
InheritedClasses_io	that declares the generalized classes
PublicMembers_io	public members
ProtectedMembers_io	protected members
PrivateMembers_io	private members
inlineMethods_io	corresponding to the non included "inline" methods
interfaceBottom_io	at the end of the interface file
bodyIncludes_io	class inclusion in the body file
bodyHeader_io	at the beginning of the body file
classBody_io	placed in the body file corresponding to the methods

Initialization before generating code

The signature of the parameterization method is as follows:

```
Class:userInitialisation() return boolean
```

The user can use this method to update the variables documented in the previous section so they be taken into account in the generated code. Preliminary processing, such as consistency checks specific to the user, may be inserted into the body of this method.

Redefining the CR_Set macro

The signature of the parameterization method is as follows:

```
Class:userCRSetGeneration (
  typeName_io : inout string ) return boolean
```

The ... parameter	contains in C++ the code ...
typeName_io_io	the name of the class to instantiate the CR_Set macro.

When the "*typeName_io*" parameter is equal to an empty character string, the macro is not called.

This method is not called when the parameter "*Ascending compatibility since version 4.2*" is not selected.

Processing specific to the classes annotated {extern}

The signature of the parameterization method is as follows:

```
Class:userExternClassGeneration () return boolean
```

By default, no code is generated for a class annotated with the {extern} tagged value. By re-defining this method, it is possible to launch specific processing associated to these classes.

"forward" inclusions and declarations

The signature of the parameterization method is as follows:

```
Class:userForwardIncludeGeneration (
  hxxInclude_io : inout string
  hxxIncludeSuffix_io : inout string
  cxxInclude_io : inout string
  cxxIncludeSuffix_io : inout string
  forward_io : inout string
  forwardPrefix_io : inout string ) return boolean
```

The ... parameter	contains in C++ the code ...
hxxInclud_io	the name of the files to be included for primitive class type attributes in the class's header file
hxxIncludeSuffix_io	the suffix of the inclusion files to insert them in the class's header file
cxxInclude_io	the name of the files to be included for primitive class type attributes in the class's body file
cxxIncludeSuffix_io	the suffix of the inclusion files to insert them in the class's body file
forward_io	the name of the classes to be added in the "forward" declarations zone
forwardPrefix_io	the prefix of the classes in "forward" declaration

Producing code

The signature of the parameterization method is as follows:

```
Class:userIsCode () return boolean
```

When the method returns FALSE, code is not generated for the current class.

Declaration

The signature of the parameterization method is as follows:

```
Class:userPrefixDeclarationClassGeneration (
  declarationClass_io : inout string ) return boolean
```

The ...parameter	contains in C++ the code ...
declarationClass_io	the class declaration

Encapsulation

The signature of the parameterization method is as follows:

```
Class:userFriendsGeneration (
  hxxFriends_io : inout string ) return boolean
```

The ... parameter	contains in C++ the code ...
hxxFriends_io	the string containing the declaration of the "friend" classes

Invariant

The signature of the parameterization method is as follows:

```
Class:userInvariantGeneration (
  methodsDeclarations_io : inout string
  methodsCxxDefinition_io : inout string ) return boolean
```

The ...parameter	contains in C++ the code ...
methodsDeclarations_io	the declaration of the class invariant.
methodsCxxDefinition_io	the definition of the class invariant.

Attributes

Parameterizing

After an attribute is generated, a parameterization method is launched, with the C++ translations of the different attribute characteristics:

- ◆ C++ name
- ◆ C++ type
- ◆ initial value
- ◆ attribute access

According to the kind of attribute (list, table or scalar), the appropriate parameterization method is activated.

When the method is returned, the final C++ formatting of the attribute is carried out, taking into account the modifications that may have been made to the C++ code.

In the following sections, the J methods flagged with * are not called when the "*Ascending compatibility since version 4.2.0*" parameter is not selected.

Set of generated code zones for a simple attribute

For a simple attribute, the signature of the parameterization method is as follows:

```
*Attribute:userAtomicGeneration (
  cxxName_io : inout string,
  cxxType_io : inout string,
  cxxExternal_tpe_io : inut string,,
  initialValue_io : inout string,
  cxxGetLine_io : inout string,
  cxxSetLine_io : inout string,
  cxxCardLine_io : inout string,
  cxxIndexLine_io : inout string,
  hxxIncludeType_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
cxxName_io	the name of the attribute
cxxType_io	the name of the attribute's type
cxxExternalType_io	the name of the type used by the access methods
initialValue_io	the attribute's initial value
cxxGetLine_io	the line for obtaining the attribute
cxxSetLine_io	the line for allocating the attribute
cxxCardLine_io	the line for obtaining the multiplicity of the list
cxxIndexLine_io	the line for obtaining the nth element of the list
hxxIncludeType_io	the type to be included

Set of generated code zones for a list type attribute

For an attribute translated in the form of a list, the signature of the parameterization method is as follows:

```
*Attribut:userListGeneration (
  cxxName : inout string,
  cxxType : inout string,
  cxxExternalType : inout string,
  cxxAtomicType : inout string,
  isReturnedByValue : inout boolean,
  initialValue : inout string,
  cxxGetLine : inout string,
  cxxSetLine : inout string,
  cxxCardLine : inout string,
  cxxIndexLine : inout string,
  cxxAppendLine : inout string,
  cxxEraseLine : inout string,
  hxxIncludeType : inout string ) return boolean
```

The ... parameter	contains in C++ ...
cxxName	the name of the attribute
cxxType	the name of the attribute's type
cxxExternalType	the name of the type used by the access methods
cxxAtomicType	the primitive type in the list
isReturnedByValue	an indicator specifying whether the return is by value or through reference
initialValue	the attribute's initial value
cxxGetLine	the line for obtaining the attribute
cxxSetLine	the line for allocating the attribute
cxxCardLine	the line for obtaining the multiplicity of the list
cxxIndexLine	the line for obtaining the nth element of the list
cxxAppendLine	the line for adding an element in the list
cxxEraseLine	the line for suppressing an element in the list
hxxIncludeType	the type to be included

Set of generated code zones for an attribute annotated {array}

For an attribute translated in the form of a table, the signature of the parameterization method is as follows:

```
*Attribute:userArrayGeneration (
  cxxName : inout string,
  cxxType : inout string,
  cxxExternalType : inout string,
  initialValue : inout string,
  cxxGetLine : inout string,
  cxxSetLine : inout string,
  cxxCardLine : inout string,
  cxxIndexLine : inout string,
  hxxIncludeType : inout string : return boolean
```

The ... parameter	contains in C++ ...
cxxName	the name of the attribute
cxxType	the name of the attribute's type
cxxExternalType	the name of the type used by access methods
initialValue	the attribute's initial value
cxxGetLine	the line for obtaining the attribute
cxxSetLine	the line for allocating the attribute
cxxCardLine	the line for obtaining the multiplicity of a set
cxxIndexLine	the line for obtaining the nth element of a set
hxxIncludeType	the type to be included

Redefining the CR_Set macro

The signature of the parameterization method is as follows:

```
*Attribute : userCRSetGeneration (
    typeName_io : inout string
```

The ... parameter	contains in C++ the code ...
typeName_io	the name of the class to instantiate the CR_Set macro.

When the "typeName_io" parameter is equal to an empty character string, the macro is not called.

"forward" inclusions and declarations

The signature of the parameterization method is as follows:

```
*Attribute:userForwardIncludeGeneration (
    hxxInclude_io : inout string
    hxxIncludeSuffix_io : inout string
    cxxInclude_io : inout string
    cxxIncludeSuffix_io : inout string
    forward_io : inout string
    forwardPrefix_io : inout string ) return boolean
```

The ... parameter	contains in C++ the code ...
hxxInclude_io	the name of the files to be included for primitive class type attributes in the class's header file
hxxIncludeSuffix_io	the suffix of the inclusion files to insert them in the class's header file
cxxInclude_io	the name of the files to be included for primitive class type attributes in the class's body file
cxxIncludeSuffix_io	the suffix of the inclusion files to insert them in the class's body file
forward_io	the name of the classes to be added in the "forward" declarations zone
forwardPrefix_io	the prefix of the classes in "forward" declaration

Producing code

The signature of the parameterization method is as follows:

```
Attribute:userIsCode () return boolean
```

When the method returns FALSE, code is not generated for the current package.

Name of the access methods

The signature of the parameterization method is as follows:

```
*Attribute:userAccessMethodName () return string
```

This method allows the definition of a rule used to name the "get_" type attribute accessors.

Name of the method used to obtain multiplicity

The signature of the parameterization method is as follows:

```
*Attribute:userCardinalMethodName () return string
```

This method allows the definition of a rule used to name the "card_" type methods.

Associations

Parameterizing

The *AssociationEnd* metaclass has parameterization methods. This metaclass represents one end of the association link.

After an association is generated, a parameterization method is launched, with C++ translations of the different association characteristics:

- ◆ C++ name
- ◆ C++ type
- ◆ initial value
- ◆ access to the association elements

When the method is returned, the final C++ formatting of the association is carried out, taking into account the possible modifications to the C++ code.

In the following sections, the J methods flagged with * are not called when the "Ascending compatibility since version 4.2.0" parameter is not selected.

Set of generated code zones for an association

The signature of the parameterization method is as follows:

```
*Association:userGeneration (
  cxxName_io : inout string,
  cxxType_io : inout string,
  cxxInit_io : inout string
  cxxGetLine_io : inout string,
  cxxCardLine_io : inout string,
  cxxAppendLine_io : inout string,
  cxxEraseLine_io : inout string,
  cxxFullGetLine_io : inout string,
  cxxFullSetLine_io : inout string ) return boolean
```

The ...parameter	contains in C++ ...
cxxName_io	the name of the attribute that implements the association
cxxType_io	the name of the type that implements the association
cxxInit_io	the deduced initialization code

cxxGetLine_io	the line for obtaining the nth element of the association
cxxCardLine_io	the line for obtaining the multiplicity of the association
cxxAppendLine_io	the line for adding an element to the association
cxxEraseLine_io	the line for deleting an element of the association
cxxFullGetLine_io	the line for obtaining the association set
cxxFullSetLine_io	the line for allocating the association set

Parameter type for operations used to access associations

The signature of the parameterization method is as follows:

```
*AssociationEnd:userGenerationRelationArgument -
  argType_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
argType_io	the name of the parameters used in the services for accessing associations (ex. : append_role())

Return parameter type for operations used to access associations

The signature of the parameterization method is as follows:

```
*AssociationEnd:userGenerationRelationReturn (
  retType_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
retType_io	the name of the return parameter used in the services for accessing the associations

Redefining the CR_Set macro

The signature of the parameterization method is as follows:

```
*AssociationEnd:userCRSetGeneration (
    typeName_io : inout string ) return boolean
```

The ...parameter	contains in C++ ...
typeName_io	the name of the class to instantiate the CR_Set macro.

When the "*typeName_io*" parameter is equal to an empty character string, the macro is not called.

"forward" inclusions and declarations

The signature of the parameterization method is as follows:

```
*AssociationEnd:userForwardIncludeGeneration (
    hxxInlcude_io : inout string
    hxxInlcudeSuffix_io : inout string
    cxxInlcude_io : inout string
    forward_io : inout string
    forwardPrefix_io : inout string ) return boolean
```

The ...parameter	contains in C++ ...
hxxInlcude_io	the name of the files to be included for primitive class type attributes in the class's header file
hxxInlcudeSuffix_io	the suffix of the inclusion files to insert them in the class's header file
cxxInlcude_io	the name of the files to be included for primitive class type attributes in the class's body file
forward_io	the name of the classes to add in the " <i>forward</i> " declarations zone
forwardPrefix_io	the prefix of the classes in " <i>forward</i> " declaration

Parameter type for operations used to access associated objects

The signature of the parameterization method is as follows:

```
*AssociationEnd:userGenerationRelationFullArgument -
  argType_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
argType_io	the name of the parameters used in the services for accessing associations

Return parameter type for operations used to access associated objects

The signature of the parameterization method is as follows:

```
*AssociationEnd:userGenerationRelationFullReturn (
  retType_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
retType_io	the name of the return parameter used in the services for accessing the associations

Producing code

The signature of the parameterization method is as follows:

```
*AssociationEnd:userIsCode () : return boolean
```

When the method returns FALSE, code is not generated for the current package.

Name of the class member that implements the association

The signature of the parameterization method is as follows:

```
*AssociationEnd:userVariableName (  
    varName : inout string ) return boolean
```

The ... parameter	contains in C++ ...
varName	the name of the attribute for implementing the association

Names of access methods

The signature of the parameterization method is as follows:

```
*AssociationEnd:userAccessMethodName () return string
```

This method allows the definition of a rule used to name the "get_" type attribute accessors.

Name of the method for obtaining the multiplicity

The signature of the parameterization method is as follows:

```
*AssociationEnd:userCardinalMethodName () return string
```

This method allows the definition of a rule used to name the "card_" type methods.

Instance management

After the generation that is related to the instance management ("*instance*" keyword), the parameterization method is launched with the C++ translations of the different characteristics:

- ◆ C++ name
- ◆ C++ type
- ◆ initial value
- ◆ access to the instance management elements.

When the method is returned, the final C++ formatting of the attribute is carried out taking into account the modifications that may have occurred to the C++ code.

Set of generated code zones for an association

The signature of the parameterization method is as follows:

```
*AssociationEnd:userInstanceGeneration (
  cxxName_io : inout string
  cxxType_io : inout string
  cxxInitialisation_io : inout string
  cxxGetLine_io : inout string
  cxxCardLink_io : inout string
  cxxAppendLine_io : inout string
  cxxEraseLine_io : inout string
  cxxFullGetLine_io : inout string
  cxxFullSetLine_io : inout string
```

The ... parameter	contains in C++ ...
cxxName_io	the name of the attribute that implements the instance management
cxxType_io	the name of the type that implements the instance management
cxxInitialisation_io	the deduced initialization code
cxxGetLine_io	the line for obtaining the nth element of the instance management
cxxCardLine_io	the line for obtaining the multiplicity of the instance management
cxxAppendLine_io	the line for adding an element to the instance management
cxxEraseLine_io	the line for deleting an element of the instance management
cxxFullGetLine_io	the line for obtaining the instance management set
cxxFullSetLine_io	the line for allocating the instance management set

Operations

Parameterizing

After a method is generated, one or more parameterization methods are launched with the C++ translations of the different method characteristics:

- ◆ C++ name
- ◆ C++ return
- ◆ C++ parameter list
- ◆ pre-condition
- ◆ the method's C++ code
- ◆ post-condition
- ◆ actual return

When the method is returned, the final C++ formatting of the attribute is carried out, taking into account the modifications that may have been made to the C++ code.

Set of code zones generated for an operation

The signature of the parameterization method is as follows:

```
Operation:userGeneration (
  cxxName_io : inout string,
  cxxReturnDecl_io : inout string
  parameterList_io : inout SetOfstring,
  cxxPre_io : inout string,
  cxxCode_io : inout string,
  cxxPost_io : inout string,
  cxxReturn_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
cxxName_io	the method's name
cxxReturnDecl_io	the method's return declaration
parameterList_io	the list of parameters
cxxPre_io	the pre-conditions
cxxCode_io	the method's code
cxxPost_io	the post-conditions
cxxReturn_io	the returned actual expression

Constructor

In the case of the constructors, another operation is activated in order to take into account specific parts of generation. The signature of the parameterization method is as follows:

```
Operation:userCreateGeneration -
  deduceConstruction : inout string
  deduceInitialisation : inout string,
  instanceHandling : inout string ) return boolean
```

The ... parameter	contains in C++ ...
deduceConstruction	the deduced constructions
deduceInitialisation	the deduced initializations
instanceHandling	the insertion of the object into the instance management

Destroyer

In the case of the constructors, another method is activated in order to take into account specific parts of generation. The signature of the parameterization method is as follows:

```
Operation:userDeleteGeneration (
  instanceHandling : inout string ) return boolean
```

The "*instanceHandling*" parameter contains, in C++, the deletion of the instance management object.

Return parameter

The signature of the parameterization method is as follows:

```
Operation:userconfReturnGeneration (
  typeName_io : inout string
  passingMode_io : inout string
  constSuffix_io : inout string
) return boolean
```

The ... parameter	contains in C++ ...
typeName_io	the name of the return parameter
passingMode_io	the return parameter's passing mode
constSuffix_io	the suffix of the return parameter

"forward" inclusions and declarations

The signature of the parameterization method is as follows:

```
Operation:userForwardIncludeGeneration -
  hxxInclude_io : inout string
  hxxIncludeSuffix_io : inout string
  cxxInclude_io : inout string
  cxxIncludeSuffix_io : inout string
  forward_io : inout string
  forwardPrefix_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
hxxinclude_io	the names of the files to be included in the class's header file
hxxIncludeSuffix_io	the suffix of the inclusion files to insert them in the class's header file
cxxInclude_io	the names of the files to be included in the class's body file
cxxIncludeSuffix_io	the suffix of the inclusion files to insert them in the class's body file
Forward_io	the name of the classes to be added in the "forward" declarations zone
forwardPrefix_io	the prefix of the classes in "forward" declaration

Producing code

The signature of the parameterization method is as follows:

```
Operation:userIscode () return boolean
```

When this method returns FALSE, code is not generated for the current package.

Adding a prefix to the operation's declaration

The signature of the parameterization method is as follows:

```
Operation:userPrefixMethodGeneration (
  prefix_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
prefix_io	the character string to be added as prefix to the class's methods

Return parameter

The signature of the parameterization method is as follows:

```
Operation:userReturnGeneration -
  hxxReturn_io : inout string
  cxxReturn_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
hxxReturn_io	declaration of the method's return parameter in the class's header file
cxxReturn_io	declaration of the method's return parameter in the class's body file

Return parameter type

The signature of the parameterization method is as follows:

```
Operation:userReturnTypeGeneration -
  typeName_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
typeName_io	the declaration of the method return type

Free text zones to be inserted

The signature of the parameterization method is as follows:

```
Operation:userBeforeAfterGeneration -
  beforeDeclaration_io :inout string
  afterDeclaration_io :inout string
  beforeImplementation_io : inout string
  afterImplementation_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
beforeDeclaration_io	the text to be inserted before the method's declaration
afterDeclaration_io	the text to be inserted before the method's declaration
beforeImplementation_io	the text to be inserted before the method's body
afterImplementation_io	the text to be inserted before the method's body

Parameters

The signature of the parameterization method is as follows:

```
Operation:userHeaderParamListGeneration -
  parametersHeader_io : inout string,
  initValues_io : inout string,
  ids_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
parametersHeader_io	the set of the method's parameters
initValues_io	the set of the default values of the method's parameters
ids_io	the set of the identifiers of the method's parameters

Trace starting message

The signature of the parameterization method is as follows:

```
Operation:userGenerateBeginningTrace (
  preTrace_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
preTrace_io	the character string displayed in Trace mode at the beginning of the method's execution

Trace end message

The signature of the parameterization method is as follows:

```
Operation:userGenreatedEndingTrace (  
    postTrace_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
postTrace_io	the character string displayed in Trace mode at the end of the method's execution

Virtual destructor

The signature of the parameterization method is as follows:

```
Operation:userVirtualDestructor -  
    virtualStr_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
virtualStr_io	the declaration of the class destructor

Parameters

Parameterization

After a parameter is generated, a parameterization method is launched with the C++ translations of the different attribute characteristics:

- ◆ C++ name
- ◆ C++ type
- ◆ initial value
- ◆ passing mode

When the method is returned, the final C++ formatting of the parameter is carried out, taking into account the possible modifications to the C++ code.

In the following sections, the J methods flagged with * are not called when the "*Ascending compatibility since version 4.2.0*" parameter is not selected.

Set of code zones generated for a parameter

The signature of the parameterization method is as follows:

```
*Parameter:userGeneration (
  cxxName_io : inout string,
  cxxType_io : inout string,
  initialValue_io : inout string,
  cxxPassing_io : inout string,
  cxxConst_io : inout string,
  userCxx_io : inout string ) return boolean
```

The ... parameter	contains in C++ ...
cxxName_io	the name of the parameter
cxxType_io	the name of the parameter type
initialValue_io	the parameter's initial value
cxxPassing_io	the parameter's passing mode
cxxConst_io	the keyword const or empty
userCxx_io	a complete declaration specific to the user

Re-defining the CR_Set macro

The signature of the parameterization method is as follows:

```
*Parameter:userCRSetGeneration (  
    typeName_io : inout string,  
    ) return boolean
```

The ... parameter	contains in C++ ...
typeName_io	the name of the class to instantiate the CR_Set macro

When the "typeName_io" parameter is equal to an empty character string, the macro is not called.

Inclusions and declarations "forward"

The signature of the parameterization method is as follows:

```
*Parameter:userForwardIncludeGeneration (
  hxxInclude_io : inout string
  hxxIncludeSuffix_io : inout string
  cxxInclude_io : inout string
  cxxIncludeSuffix_io : string
  forward_io : inout string
  forwardPrefix_io : inout string,
) return boolean
```

The ... parameter	contains in C++ ...
hxxInclude_io	the names of the files to be included in the class's header file
hxxIncludeSuffix_io	the suffix of the inclusion files to insert in the class's header file
cxxInclude_io	the names of the files to be included in the class's body file.
cxxIncludeSuffix_io	the suffix of the inclusions files to insert in the class's body file
forward_io	the name of the classes to be added in the "forward" declarations zone
forwardPrefix_io	the prefix of the classes in "forward" declaration

Data Type

Parameterization

At the end of basic type generation, two parameterization methods are activated to define the type (typedef).

When the method is returned, the final C++ formatting of the type is carried out, taking into account the possible modifications made to the C++ code.

Name

The signature of the parameterization method is as follows:

```
DataType:userGenerateClassCRType (
    typeName_io : inout string,
    ) return boolean
```

The ... parameter	contains in C++ ...
typeName_io	the name of the type to be defined

Prefix

The signature of the parameterization method is as follows:

```
DataType:userGenerateSchemaCRType (
    hxxPrefix_io : inout string,
    ) return boolean
```

The ... parameter	contains in C++ ...
hxxPrefix_io	the name of the prefix to use for the type to be defined

Header

Parameterization

After a class is generated, a parameterization method is activated for the redefinition of a header inserted at the beginning of the class' header and body file.

When the method is returned, the final C++ formatting of the file is carried out, taking into account the possible modifications.

Method

The signature of the parameterization method is as follows:

```
Unit:generateUserHeader (
    hxxUserHeader_io : inout string
    cxxUserHeader_io : inout string,
```

The ... parameter	contains in C++ ...
hxxUserHeader_io	the comment zone inserted at the beginning of the class's header file
cxxUserHeader_io	the comment zone inserted at the beginning of the class's body file

ModelElement

Parameterization

The parameterization available on a model element applies to:

- ◆ attributes
- ◆ associations
- ◆ parameters

Class to be used in a types package

The signature of the parameterization method is as follows:

```
ModelElement:userFilterDirective (
    label_o : inout string,
    ) return boolean
```

The ... parameter	contains in C++ ...
label_o	the name of the class for implementing the type

According to the tagged values present on an attribute, an association or a parameter, the user must explicitly specify the type to be used in the type package. The name of this type must be given to the "label_o" parameter.

Name of the parameter

The signature of the parameterization method is as follows:

```
ModelElement:userParameterName (
    name_io : inout string,
    ) return boolean
```

The ... parameter	contains in C++ ...
name_io	the name of the class for implementing the type

Pre-condition

The signature of the parameterization method is as follows:

```
ModelElement:userGeneratePreConditions (
    preCheck_io : inout string,
    ) return boolean
```

The ... parameter	contains in C++ ...
preCheck_io	the pre-condition code

Post-condition

The signature of the parameterization method is as follows:

```
ModelElement:userGeneratePostConditions (
    postCheck_io : inout string,
    ) return boolean
```

The ... parameter	contains in C++ ...
preCheck_io	the post-condition code

Enumeration

Parameterization

After an enumeration has been generated, a parameterization method is launched, with the C++ translations for adding a prefix to the different enumerate values.

Prefix to be added to the enumeration defined on a class

The signature of the parameterization method is as follows:

```
Enumeration:userGenerateClassEnumerate (
    hxxPrefix_io : inout string,
    ) return boolean
```

The ... parameter	contains in C++ ...
hxxPrefix_io	the value of the prefix of the enumeration values

Prefix to be added to the enumeration defined on a package

The signature of the parameterization method is as follows:

```
Enumeration:userGenerateSchemaEnumerate (
    hxxPrefix_io : inout string,
    ) return boolean
```

The ... parameter	contains in C++ ...
hxxPrefix_io	the value of the prefix of the enumeration values

Enumeration values

The signature of the parameterization method is as follows:

```
EnumerationLitteral:userEnumerateLitteral (  
    litteral_io : inout string,  
    ) return boolean
```

The ... parameter	contains in C++ ...
litteral_io	the enumerate values

Generalization

Parameterization

At the end of C++ code generation dedicated to a generalization between classes, a parameterization method is activated, with the C++ translations for modifying the code generated by default.

Parent class

The signature of the parameterization method is as follows:

```
Generalization:userClassInheritance (  
    className_io : inout string,  
    ) return boolean
```

The ... parameter	contains in C++ ...
className_io	the name of the basic class for the current class

"forward" inclusions and declarations

The signature of the parameterization method is as follows:

```
Generalization:userForwardIncludeGeneration (
    hxxInclude_io : inout string
    hxxIncludeSuffix_io : inout string
    cxxInclude_io : inout string
    cxxIncludeSuffix_io : inout string
    forward_io : inout string
    forwardPrefix_io : inout string
```

The ... parameter	contains in C++ ...
hxxInclude_io	the names of the files to be included in the class's header file
hxxIncludeSuffix_io	the suffix of the files to be included in the class's header file
cxxInclude_io	the names of the files to be included in the class's body file
cxxIncludeSuffix_io	the suffix of the inclusions files to insert them in the class's body file
forward_io	the name of the classes to be added in the "forward" declarations zone
forwardPrefix_io	the prefix of the classes in "forward" declaration

Member

Parameterization

At the end of generation of a class's members, a parameterization method is activated, together with the default C++ translations related to the visibility of each class member.

Visibility of the class's attributes

The signature of the parameterization method is as follows:

```
Member:userAttributeVisibility (
    vis_io : VisibilityMode,
    ) return boolean
```

The ... parameter	contains in C++ ...
vis_io	the C++ visibility of the class member

Visibility of an attribute's modification features

The signature of the parameterization method is as follows:

```
Member:userModifierVisibility (
    vis_io : VisibilityMode,
    ) return boolean
```

The ... parameter	contains in C++ ...
vis_io	the C++ visibility of the modification access method (set_<attributeName> of the current attribute)

Visibility of an attribute's accessors

The signature of the parameterization method is as follows:

```
Member:userAccessorVisibility (  
    vis_io : VisibilityMode,  
    ) return boolean
```

The ... parameter	contains in C++ ...
vis_io	the C++ visibility of the read only access method (set_<attributeName> of the current attribute)

Examples

Overview

This paragraph contains different examples of C++ generation parameterization at class, attribute, association, operation and parameter levels, in a sub-UML profile, "*User*".

Class: Modifying constants

This example presents the second definition of the header at the beginning of the include file.

```
Class:default#Code#Cxx#User#userInitialization ()    return
boolean
{
    InterfaceBeginningComment = "";
    InterfaceBeginningComment.concat ( "/*", Name, "*/" );
    Return:=true;
}
```

Class: Adding an inclusion

This example shows how to add the inclusion of a "userFile.hxx" file, according to the presence of a specific "userDirective" tagged value.

```

Class:default#Code#Cxx#User#userGeneration
  (interfaceIncludes:inout string,
   interfaceHeader:inout string,
   inheritedClasses:inout string,
   publicMembers:inout string,
   protectedMember:inout string,
   privateMember:inout string,
   inlineMethods:inout string,
   interfaceBottom:inout string,
   bodyIncludes:inout string,
   bodyHeader:inout string,
   classBody:inout string) return boolean
{
  cr : string = "
";

  // Adding of an include if userDirective tagged value
  { ( isTaggedValue ( "userDirective" ) ) then
    interface_includes.concat ( cr, "#include
<userFile.hxx>", cr );
  }
  return := true;
}

```

Attribute: Modifying the C++ name

This example shows how to modify the name of the attribute, according to the presence of an {alias} tagged value, containing a parameter that indicates the name that should be taken into account.

```
Attribute:default#Code#Cxx#User#userAtomicGeneration
(cxxName:inout string,
 cxxType:inout string,
 cxxExternalType:inout string,
 initialValue:inout string,
 cxxGet:inout string,
 cxxSet:inout string,
 cxxInclude:inout string) return boolean

{
  aliasDir : TaggedValue;
  paramDir : SetOfstring;

  return = true;
  // testing the existence of the alias tagged value
  { ( isTaggedValue ( "alias" ) ) then
    // recapturing of the tagged value and of the
parameter
    aliasDir = getTaggedValue( "alias" );
    paramDir = aliasDir.currentParameters();
    { ( paramDir.card() != 1 ) then
      // error in the usage of the tagged value
      StdErr.write ( "the use of alias on ", Name,
        " is incorrect" );
      // stopping the C++ code generation
      return = false;
    else
      // allocation of the C++ attribute name
      paramDir
      {
        {
          cxx_name = self;
        }
      }
    }
  }
}
```

Association: Initialization

This example shows the initialization of the association with a predefined static object *"userStaticObject"*, according to the existence of a {autoinit} tagged value and the association's multiplicity.

```
RelationLink:default#Code#Cxx#User#userGeneration (
  cxxName:inout string,
  cxxType:inout string,
  cxxInitialisation:inout string,
  cxxIndexLine:inout string,
  cxxCardLine:inout string,
  cxxAppendLine:inout string,
  cxxEraseLine:inout string,
  cxxFullGetLine:inout string,
  cxxFullSetLine:inout string)    return boolean
{
  { ( isTaggedValue ( "autoinit" ) ) then
    { ( originMultiplicityMax() = 1 ) then
      -- initialization with a predefined static object
      cxxInitialization = "";
      cxxInitialization.concat ( Name, " =
&userStaticObject;" );
    }
  }
  return = true;
}
```

Method: Adding C++ code

This example shows the addition of C++ code at the end of a method, according to the existence of the "addcode" characteristic on the current UML modeling project.

```
Operation:default#Code#Cxx#User#userGeneration (
    cxxName:inout string,
    cxxReturnDecl:inout string,
    parameterList:inout SetOfstring,
    cxxPre:inout string,
    cxxCode:inout string,
    cxxPost:inout string,
    cxxReturn:inout string)    return boolean
{
    { ( isProjectAttribute ( "addcode" ) ) then
        // adds a call to a user function
        cxxCode.concat ( "userAddCodeFunction (); " );
    }
    return = true;
}
```

Parameter: Modifying the passing mode

This example shows how to modify a passing mode, according to a specific {userpassing} tagged value and for a specific "usertype" type.

```
Parameter:default#Code#Cxx#User#userGeneration (
    cxxName : inout string,
    cxxRtype : inout string,
    initialValue : inout string,
    cxxPassingPrefix : inout string,
    cxxPassingSuffix : inout string,
    cxxConst : inout string,
    userCxx : inout string ) return boolean
{
    { ( isTaggedValue ( "userpassing" ) ) then
        { ( cxxType = "usertype" ) then
            //effectation of passing mode
            cxxPassingPrefix = "**&";
            //elation of the possible "const"
            cxxConst = "";
        }
    }
    return = true;
}
```

Chapter 16: Adapting makefile generation

Adapting the makefile

Overview

The standard generation of a makefile file is based on a document template. The generator calls in turn each document template method, which then returns a string. The resulting makefile is produced by concatenating these strings.

To adapt generation, there are three possible types of parameterization:

- ◆ changing the value of the standard module's parameters
- ◆ redefining certain methods of existing document templates
- ◆ redefining a document template

For the two latter cases, it is necessary to parameterize C++ makefile generation in the *Objectteering/UML Profile Builder* module. We will give details below on the second case, presenting the methods that can be redefined.

The services that are used for redefining modules are described in the *Objectteering/UML Profile Builder* module.

The services that are used for redefining document templates are described in the *Objectteering/Document Template Editor* module.

Principle of the makefile generation document template

The document template is a recent and very useful technique, allowing you to represent the structure of a target (in this case a makefile) in the form of a hierarchy. Each type of note in a makefile is represented by an element with the definition of the extracted information.

The *Objecteering/Document Template Editor* user guide explains how to handle generation templates. The chapter on document rule parameterization describes how to adapt a document template.

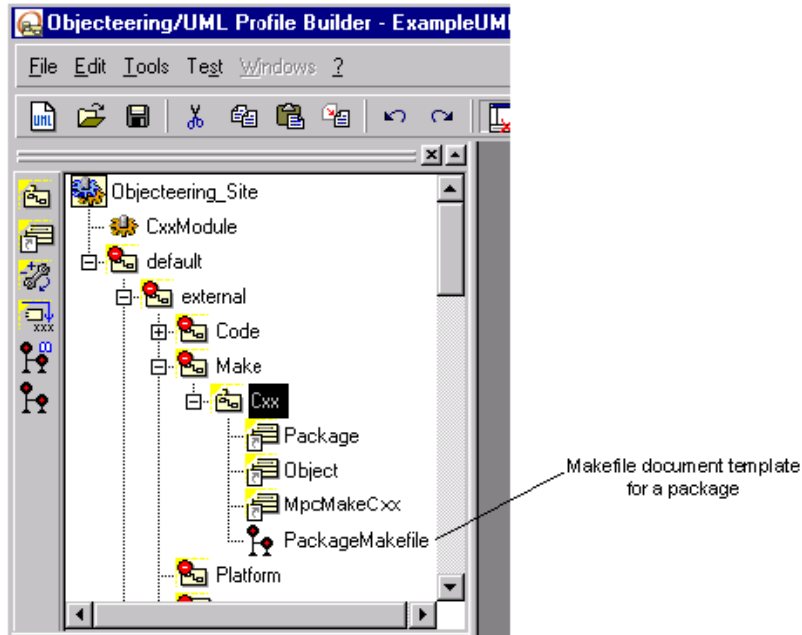


Figure 16-1. The Makefile document templates

The detail of a document template

The document template defines each zone that must be located in a generated makefile, for a given type of metaclass. Everything necessary for the creation of a makefile in an orderly way is presented, for example compilation options, target definition, or dependency calculations.

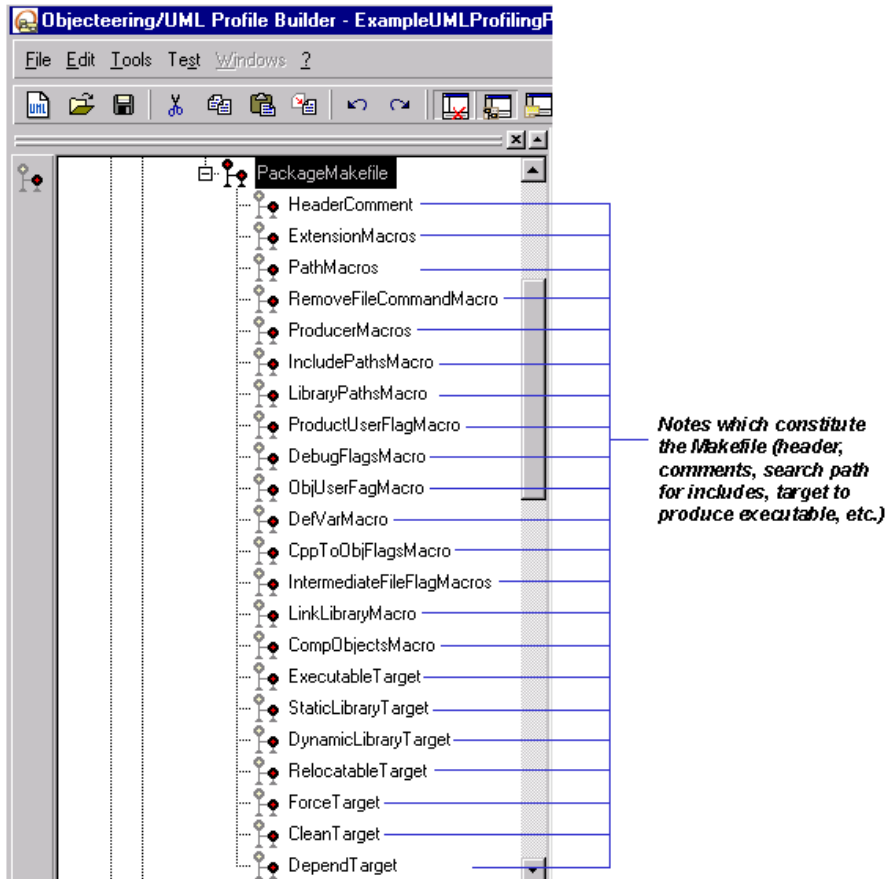


Figure 16-2. Detailed document template for the package

Procedure to be followed to adapt makefile generation

- 1 - Edit a UML profiling project.
- 2 - Import the standard *Objectteering/C++* module.
- 3 - Create a child profile in "default#external#Make#Cxx".
- 4 - In this sub-UML profile, redefine the desired methods, or create a new document template.
- 5 - Create a module.
- 6 - Make this module specialize CxxModule, (*Objectteering/C++* standard module).
- 7 - Reference the new UML profile from this module.

Procedure for using the parameterization in a UML modeling project

- 1 - Install the new module in a UML modeling project.
 - 2 - In this new UML modeling project, select the new module.
-

Accessing the compilation work product

Procedure

Makefile document templates call the methods linked to the model objects, for which a makefile is generated.

It is sometimes necessary to ask for the compilation work product, for which the makefile is being generated. The compilation work product can be accessed through the *currentProduct* variable.

Compilation work product attributes

List of attributes

The information of the compilation work product dialog box is accessible in the J language with the following attributes:

The ... attribute	corresponds to ...
Name (string)	the work product name
ProductPath (string)	the production directory of the makefiles and the final binary files
IntermediatePath (string)	the production directory of the intermediary files
ProductType (enumerate)	the type of binary to generate. It can be the MptExecutable, MptStatcclib, MptDynamicLib or MptObject values
IsDebug (boolean)	the Boolean indicating whether the compilation can be executed in debug mode
ObjUserFlags (string)	options for compiling the sources into objects
JumpThroughErrors (boolean)	the Boolean indicating whether to continue running the makefile in case of an error
UserDefVarsMpAssocRep (SetOfMpAssocRep)	the list of variable definitions. An object with the type MpAssocRep contains a field named Key. The field represents the variable value that can be empty
IncPathMpStringRep (SetOfMpStringRep)	the list of the search paths for the header files. An object with the type MpStringRep contains a field named Value. The field represents the path(s)
CompLibsMpStringRep (SetOfMpStringRep)	the list of libraries to link to get the final binary. An object of the type MpStringRep contains a field named Value. The field represents the library
LibPathMpStringRep (SetOfMpStringRep)	the list of paths for searching for libraries to be linked to the final binary. An object with the type MpStringRep contains a field named Value. The field represents the path
ExcludedClassesClass (SetOfClass)	the list of classes to exclude from the final binary

Module parameters

List of parameters

Any module which specializes the *Objectteering/C++* standard module specializes its parameters. The module can change the values of these parameters. The parameters of the *Objectteering/C++* standard module related to the makefile generation are listed below.

The ... parameter	corresponds ...
ObjectteeringLibraries	to the Objectteering/UML libraries (separated by ";").
ObjectteeringIncludes	to the files for searching the Objectteering/UML "include" files, (separated by ";").
SysIncludes	to the files for searching the Objectteering/UML "include" files, (separated by ";").
DefaultCompilerIncludes	to the files for searching the compiler's implicit "include" files, (separated by ";").
CRDepend	to the analyzer of dependencies between C++ sources
CppToObjFlag	to the options to compile an object C++ source
ExecutableFlag	to the options for producing an executable
StatLibFlag	to the options for producing a static library
DynLibFlag	to the options for producing a dynamic library
Compiler	to the command for producing an executable
StatLibProducer	to the command for producing a static library
DynLibProducer	to the command for producing a dynamic library

Access to a parameter

Any J method can get back a module parameter value, and, especially in the present case, the value of a C++ generation module parameter, for which the work product has been created.

For a parameter named "*ParamName*" and defined in the default#external#code#Cxx#MyCxx profile, for the "*MyCxxModule*" module, the call will have the following form:

```
String retVal;  
boolean res = getParameterValue ("ParamName", "MyCxx",  
"MyCxxModule", retVal);
```

At the end of this call, if "*res*" is "*true*", "*retVal*" will contain the module parameter's value. If "*res*" is "*false*", the "*ParamName*" parameter does not exist.

Modifying a parameter value

A module may have to modify a C++ generation module parameter value.

For a parameter named "*ParamName*", the call will have the following form:

```
module.setParameterValue ("ParamName",  
"default#external#Code#Cxx", "value")
```

Packages - part 1

Introduction

The methods described below are defined in the default `#external#Make#Cxx` UML profile for the metaclass *NameSpace*. They all return a string, have no parameters and apply to all packages.

List of methods

The ... method	offers the ... service	example ...
<code>getCppToObjOptionsPart</code>	returns the variable definition that sums up the compilation options	<code>OBJ_Flags=␣(DEBUG_FLAG) (USER_DEFVARS) \$(COMP_INC_PATHS) \$(OBJECTEERING_PATHS) \$(INC_PATHS)</code>
<code>getDebugPart</code>	returns the definition of the variable containing the debug options	(on UNIX) <code>DEBUG_FLAG=-g</code>
<code>getDefVarPart</code>	returns the definition of the variable containing the definitions of variables asked by the user. Usually, this method exploits the compilation product's <code>UserDefVarsMpAssocRep</code> attribute	(on UNIX) <code>USER_DEFVARS=-DVAR1 -DVAR2=12</code>
<code>getIncludedLibrariesPart</code>	returns the definition of the variable containing the library to add to the link edition. Typically, this method exploits the attribute <code>CompLibsMpStringRep</code> of the compilation product	(on UNIX) <code>COMP_LIBS=IX11 -Ixm</code>
<code>getObjUserFlagsPart</code>	returns the definition of the variable containing the user's compilation options. Usually, this method uses the compilation product's <code>ObjUserFlags</code> attribute and allocates its value to the <code>OBJ_USER_FLAGS</code> variable	

Packages - part 2

Introduction

The methods described below are defined in the "*default#external#Make#Cxx*" UML profile for the "*Package*" metaclass. All return a "*string*" type and have no parameters.

List of methods

The ... method	offers the ... service	example ...
getCompilerPart	returns the definition of the variable containing the compiler name (i.e. the binary that produces the objects from the C++ sources)	(on AIX) CC=xIC
getCppToObjFlagPart	returns the compilation options for producing the objects from the C++ sources	(on Solaris) OBJ_FLAG=-c
getEraseTargetPart	returns the cleaning target of the files produced by the makefile	clena : \$(REMOVE_COMMAND) \$(COMP_OBJS) \$(PRODUCT_NAME)
getHeaderPart	returns the header of the makefile file	
getIntermediateFileFlagPart	returns the flag to place in front of the name of the file resulting from a compilation, i.e. the object file	(with Visual/C++) INTERMEDIATE_FLAG=/o
getObjectFilesListPart	returns the definition of the variable containing the objects that compose the final binary	COMP_OBJS=Package1.cxx Class1.cxx Class2.cxx Class3.cxx with Class1, Class2, Class3 the trois classes of the package Package1
getObjsTargetsPart	returns a compilation target for the C++ package source and for each of its classes	

Chapter 17: The Objecteering/UML library

Overview of the Objectteering/UML library

Overview

A library is supplied with Objectteering/UML in order to:

- ◆ provide types which are absent from the C++ language (boolean and string)
- ◆ facilitate the management of the sets deduced from the Objectteering/UML model:
 - classes
 - relations
 - "set(-) of"

Objectteering/UML generates their use automatically, according to implementation requirements.

Example

Given a C1 class related to a C2 class, Objectteering/UML automatically generates:

- ◆ a class called "list_of_C2" with the same level as C2
- ◆ a C1 attribute with this type

Sources

Library sources are supplied with Objectteering/UML. You thus continue to have total control over the whole application generated by Objectteering/UML.

Primitive types

There are three primitive types:

- ◆ CR_boolean
- ◆ CR_String
- ◆ Set

Linking your application to the Objecteering/UML library

If you use Objecteering/UML types such as CR_boolean, you have to link your application to the Objecteering/UML library. The name of this library is libo.a (libo.lib on the PC platform), and it is located in the \$OBJECTEERING_PATH/genccx/lib directory.

CR_Boolean

Description

A Boolean type variable can have *TRUE* or *FALSE* values.

Default value: *FALSE*.

The operators applied to this type are those of the Boole algebra. They:

- ◆ conform to C++ syntax
- ◆ are combined with the classical C++ Boolean expressions
- ◆ convert into integers, and can be converted conversely (cast), which allows you to combine them in a transparent way within the C++ Boolean expressions

They can be displayed or read on C++ flows (streams), by providing or confirming "TRUE" or "FALSE" values (upper or lower case).

Example

```
CR_boolean check1, check2, check3;
//their default value is FALSE
int old_style1, old_style2;
...
check1 = check1 && check2 ;
cout << check1; //prints the ASCII value TRUE or FALSE

if ( check1 && old_style1 || old_style2)
//booleans may be mixed with integers in boolean expressions
{
...
}

cin >> check2;//read the ASCII value
//FALSE or TRUE (case insensitive)
if (check2 || check1) .....
```

CR_String

Description

The "*CR_string*" class replaces the classical and risky C "*char **", as it constitutes a specific type, that can be allocated and kept. This class has been built to improve:

- ◆ efficiency (no disadvantage relative to the use of a *char**)
- ◆ usage (allocation, comparison operators, etc...)
- ◆ reliability (errors checked by pre and post conditions)
- ◆ compatibility with the C++ language (combination and conversion with the *char**, usage with *istream* and *ostream* librairies)

A "*CR_string*" adapts its size according to its content. It increases in size according to its needs, thus allowing the user not to worry about size problems.

Example 1: Traditional programming

```
char* my_string1 = "hello world", * my_string2 ;

my_string1 [9] = 'k' ;
my_string1 [10] = ' ' ;
    //It's certainly an error!
    //These instructions modify the "hello world" constant

strcpy (my_string2, my_string_1) ;
    // error! There is no defined room in my_string2

if (!strcmp (my_string2, "hello worker ") )
{
    ...
}

sprintf (my_string1, "%d, %f", my_float, my_int) ;
    // Gee! You mixed the types int and float!

strcat (my_string1, " I like debugging") ;
    //Fine, you'll discover that there is no room for
    //the added text
```


Example 2: Programming strings

```
CR_string my_string1 = "hello_world", my_string2 (10) ;
// You can give a size to my_string2

my_string1 [9] = 'k' ;
my_string1 [10] = ' ' ;
//this is OK, because my string has its own value
//which is different from the "hello_world" constant

my_string2 = my_string1 ;

if (my_string2 == "hello work ")
//you may mix char* and string types
{
  ...
}

my_string1 << my_int << my_float ;
//don't worry, the compiler chooses the right operators

my_string1 += " I like debugging" ;
```

Set of primitive objects

Overview

Primitive objects (integer, boolean, string, etc.) are always handled by value, and not by pointer. They have a series of generic set libraries, which must be instantiated for each new type.

General set libraries

Objecteering/UML supplies set handling libraries to avoid coding:

- ◆ expandable libraries
- ◆ lists
- ◆ other lists containing an order

The basic rule is that whatever the computing structure used (list, array, etc.), it will always be considered as a discreet set of elements and will be handled in the same way by the same operator "[]". Furthermore, the validity of the markers is checked by the preconditions.

Sets: Structure

Sets have the following structure:

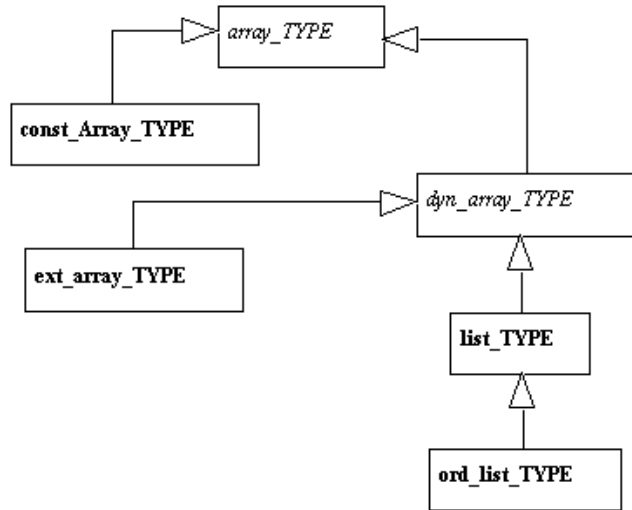


Figure 17-1. Graph representing the generalization of the general lists

Sets: Description

These structures are all generic. In the default Objecteering/UML version, sets are instantiated on the classical types (cr_string, boolean,...).

Structure	Description
array	Represents a table in a general way. Is handled by the operator []. Cannot have direct instances since the implementation must be determined by the programmer. When accessing an element of the table and when the preconditions are activated, the system checks that the markers are correct.
const_array	Is a constraint table the size of which is limited by its declaration.
dyn_array	Represents all the tables that have no size limit. Cannot have any direct instance.
ext_array	Increases according to the elements added to it.
list	Is a traditional string list. Owns an important number of operators : - "<<" to insert an element - ">>" to delete an element - "++", "--", "+", "-" to modify the current list index. It is an expandable queue owning a first and last event.
ord_list	Each new stored element is ranked according to its order. The elements in this list must have the operators : "<", ">", "=", "<=", ">=", "!=".

Set of non-primitive objects

Overview

Non-primitive objects are objects which come from classes defined in the model, which are always handled by reference. A library of object pointers set can thus be used by the users of an Objectteering/UML class, each time it is necessary.

These lists are:

- ◆ expandable
- ◆ easy to use
- ◆ optimized.

Note: These lists have evolved since version 3.4, in order to optimize the access time and memory usage.

Obtaining a set of objects

The C++ declaration is as follows:

```
CR_SET (class_name)
```

Objectteering/UML automatically proposes the class "*set of objects of class_name*", whose name will always be "*set_of_class_name*".

Chapter 18: Coverage of C++ by
Objecteering/UML

Objectteering/UML coverage of C++

Aim

This chapter lists all the forms of C++ language and specifies their correspondence in Objectteering/UML.

C++ references used

Today, the C++ language version 3.0.1 is considered the language reference. It is used as a base for the standardization proposal by the C++ANSI standardization committee. We base our chapter on:

- ◆ [2nd] "*The C++ programming Language*", second edition (1991) by Bjarne Stroustrup.
- ◆ [ARM] "*The Annotated C++ Reference Manual*" (1990), by Margaret A.Ellis and Bjarne Stroustrup.
- ◆ [Stroustrup, 1994] "*The Design and Evolution of C++*" (1994), by Bjarne Stroustrup.

Parts generated by Objectteering/UML

The C++ code generated by Objectteering/UML is divided into several sections:

- ◆ the declarative section (declaration of the classes, methods, attributes, etc.)
- ◆ specific mechanisms of the Objectteering/UML concepts (managing pre and post conditions, managing associations, etc.).
- ◆ the method implementation section written out by the user (texts with the "C++" types of local methods).

This chapter is exclusively dedicated to the C++ declarative section.

Adapting C++ generation

Default C++ code generation can be adapted by the user in several ways:

- ◆ tagged values (example :"@virtual")
- ◆ typed texts
- ◆ the "J" parameterization of code generation

This chapter is dedicated to the use of the first two modes, with "J" parameterization remaining available for:

- ◆ obtaining unexpected generation cases
- ◆ making the generation of cases which systematically concern the application

Methodology for defining the C++ coverage

It is not possible to determine, in an exhaustive way, the coverage of all the cases which are syntactically possible in C++. The chosen approach is, therefore, to study the different typical cases that put forward the concepts of the C++ language. The same approach has been used in [ARM] and [2nd].

Basic types

The integer type

C++ language	Objectteering/UML
int	integer
long int	@long integer
short int	@short integer
unsigned int	@unsigned integer
unsigned long int	@unsigned @long integer
unsigned short int	@unsigned @long integer

The float type

C++ language	Objectteering/UML
float	real
double	@long real

The character type

C++ language	Objectteering/UML
char	char
unsigned char	@unsigned char

Built types

The "struct" type

The "*struct*" declaration is not supported by the Objectteering/UML method. In the object context, the class replaces "*struct*". A typed text is provided for this case.

C++ language	Objectteering/UML
struct ...	Text typed "C++InterfaceHeader" on class or package

The "union" type

The declaration of "union" is not supported by the Objectteering/UML method. In the object context, the classes and generalization replace this construction. Only the "*low level*" needs still require "union". A typed text is provided for this case.

C++ language	Objectteering/UML
union ...	Text typed "C++InterfaceHeader" on class or package

The "class" type

The declaration of a named class is part of the Objectteering/UML method.

C++ language	Objectteering/UML
class identifier	class identifier

Templates

The *Template* parameter and the {bind} value is used to define the generic class or instantiate them.

The "enum" type

The declaration of a simple enumerate is part of the Objectteering/UML method.

C++ language	Objectteering/UML
enum Color { Red, Green };	enumerate.

The "typedef" type

The C++ language is used to redefine a type giving it another name. Several cases arise.

The definition of a type with C++ qualifications. In Objectteering/UML, this means making an Objectteering/UML type correspond to a C++ type. The type declaration is used to carry out this case.

C++ language	Objectteering/UML
typedef char* const <i>const_char_ptr</i> ;	type declaration with "C++" typed text containing the string "typedef char* const <i>const_char_ptr</i> ;"

Nested declarations

Overview

The C++ language is used to declare a new type while declaring a class or structure. A nested type can only be accessed by prefixing its name with the name of the class or the nested structure.

The "struct", "union" and "class" types

The nested declarations of a class, structure or union are not part of the Objectteering/UML method.

Typed texts are provided for this case.

C++ language	Objectteering/UML
class C{ class <i>Internal</i> {...	C class with the text typed C++PublicMember (or with C++PrivateMember)

The "enum" type

The enumerate types are generated in "public" visibility within the C++ class.

C++ language	Objectteering/UML
class C{ public: enum <i>Color</i> { <i>Red</i> };	Enumerate on a C class annotated @embeddedcontext

The C++ language can declare an enumeration in the protected or private part of the class. A typed text is provided for this case.

C++ language	Objectteering/UML
class C{ protected protected: enum <i>Color</i> { <i>Red</i> };	C class with the text typed C++PublicMember (or with C++PrivateMember or C++ProtectedMember)

Features

Overview

The Objectteering/UML method does not know the *function* concept. There are several possibilities to provide for this case:

- ◆ typed text use
- ◆ usage of the class methods
- ◆ definition of an adapted tagged value, and parameterization of the generation.
- ◆ parameterizing the generation for the cases of function use

Typed text

C++ language	Objectteering/UML
<code>int f ();</code>	"int f ();" in a text typed C++InterfaceHeader or C++BodyHeader of package or class

Class methods

In C++, "static" methods are implemented as C functions. Objectteering/UML class methods are provided for certain functional requirements.

C++ language	Objectteering/UML
<pre>class C_Interface { static f (); }</pre>	class C_Interface with class method P()

Defining a tagged value

It is possible to parameterize C++ code generation, by defining a new "@function" tagged value, used to annotate a class method, in order to generate a function. This function can simply be an "inline" function, which encapsulates the call to the class method. Its role is to define a functional interface.

C++ language	Objecteering/UML
<code>f();</code>	<i>Example</i> class annotated @function

In the current version of Objecteering/UML, the @function tagged value is not implemented. The parameterization of code generation and the J rules must be used to run this function.

Friendly operators

Using friendly operators

The operator here is a C++ "function", but with much richer semantics.

It is important to note that the operators' declaration (the first argument being a class type or a basic type) is a much used mechanism by C++. It allows operations to be extended or enriched on a class, which has already been defined. [Stroustrup, 1994]

The most widespread example is the "<<" operator on the ostream class:

```
ostream& operator << (ostream&, new_type);
```

This operator allows you to define how a new class is displayed on *cost*.

Definition in Objectteering/UML

We know how to represent most C++ "member" operators in Objectteering/UML, but this is not the case for "friendly" operators. Solutions for their representation are as follows:

- ◆ Use a typed text to declare an operator
- ◆ Use the parameterization of code generation when operators are systematically generated
- ◆ Define an adapted tagged value and generation

Expression tagged value

C++ language	Objectteering/UML
ostream& operator << (ostream&, Example);	text typed "C++InterfaceHeader" on a class containing the string "ostream& operator<<(ostream&, Example);"
ostream& operator << (ostream&, Example);	text typed "C++PublicMember" or "C++ProtectedMember" or "C++PrivateMember" on a class containing the string "friend ostream& operator<< (ostream&, Example);"

Specific parameterization

It is possible to parameterize code generation, to allow the annotation of a class method, in order to generate an operator. This operator can simply be an "inline" operator, which encapsulates the call to the class method.

C++ language	Objecteering/UML
ostream& operator << (ostream&, const &Example);	class method "display", annotated @operator("<<")

Note: In the current version of Objecteering/UML, the @operator tagged value is not implemented. The parameterization of code generation and the J rules must be used to run this function.

Generalization between classes

Generalization

The Objectteering/UML method, like the C++ language, allows the definition of generalization between classes, whether simple or multiple.

Virtual generalization

The *@virtual* tagged value is used to specify virtual generalization in the Objectteering/UML model.

C++ language	Objectteering/UML
<i>Derive: class public virtual Base {/*...*/}</i>	generalization annotated @virtual

Generalization with public or protected visibility

The C++ language is used to specialize another class in a private or protected way. The tagged values on the generalization allow this visibility to be expressed.

C++ language	Objectteering/UML
<i>Derive: class private Base {/*...*/}</i>	generalization annotated @private

Index

"class" type 18-6, 18-8
 "enum" type 18-7, 18-9
 "struct" type 18-6, 18-8
 "typedef" type 18-7
 "union" type 18-6, 18-8
 {&} tagged value 2-10, 10-4, 10-11, 12-8, 12-16
 {*} tagged value 2-10, 10-4, 10-11, 12-8, 12-16
 {Abstract} tagged value 12-5
 {access} tagged value 10-4, 10-11, 11-9
 {alias} tagged value 15-60
 {array} tagged value 10-4, 10-12, 11-9, 12-8, 12-16
 {array} tagged values 1-6
 {bind} tagged value 9-11, 10-4, 10-15, 11-10, 12-8, 12-17, 13-9, 13-13, 18-6
 {C++DLL } tagged value 9-3
 {C++DLL} tagged value 9-14
 {C++Name} tagged value 2-5, 2-6, 2-8, 2-10, 2-11, 2-12, 8-4, 8-7, 9-3, 9-14, 10-16, 11-12, 12-5, 12-8, 12-10, 12-17
 {C++NoNameSpace} tagged value 2-5, 8-4, 8-7
 {C++Root} tagged value 2-5, 8-4, 8-7
 {Class} tagged value 12-5
 {const} tagged value 2-10, 10-4, 10-12, 10-15, 12-8, 12-17
 {create} tagged value 10-4, 10-8, 10-12, 11-12
 {extern} tagged value 2-5, 2-6, 8-4, 8-6, 9-3, 9-13, 15-21
 {fullaccess} tagged value 10-4, 10-13, 11-10
 {in} tagged value 12-5
 {inline} tagged value 2-8, 12-5, 12-10
 {instanceHandling} tagged value 11-12
 {Jeval} tagged value 14-8, 14-9
 {long} tagged value 10-4, 10-13, 12-8, 12-18, 13-5
 {MFCDynamicMacro } tagged value 9-3
 {MFCDynamicMacro} tagged value 9-13
 {MFCInclude} tagged value 8-4, 8-7
 {mutable} tagged value 10-4, 10-16, 11-3, 11-13
 {nocode } tagged value 9-3
 {nocode} tagged value 2-5, 2-6, 2-10, 2-11, 8-4, 8-6, 9-14, 10-4, 10-16, 11-11, 12-5, 12-10
 {noCodeForAll} tagged value 8-7, 9-14
 {noconst} tagged value 10-4, 11-12, 12-5, 12-8, 12-10, 12-18
 {noinclude } tagged value 9-3
 {noinclude} tagged value 8-4, 8-6, 9-13
 {noinit} tagged value 11-10
 {noInline} tagged value 10-4, 10-16, 11-3, 11-13
 {noinvariant} tagged value 12-5, 12-11
 {own} tagged value 2-10, 2-11, 10-4, 10-16, 11-13
 {private} tagged value 9-6, 13-10
 {protected} tagged value 9-6, 13-10
 {public} tagged value 2-10, 2-11, 10-4, 10-7, 10-14, 11-11

- {short} tagged value 10-4, 10-13, 12-8, 12-18, 13-5
- {structure } tagged value 9-3
- {structure} tagged value 9-13
- {type(...)} tagged value 2-10, 2-11, 2-12
- {type} tagged value 10-4, 10-15, 11-12, 12-8, 12-19, 14-9
- {unsigned} tagged value 10-4, 10-14, 12-8, 12-19, 13-5
- {userpassing} tagged value 15-62
- {val} tagged value 12-8, 12-19
- {virtual} tagged value 2-8, 5-5, 9-5, 11-11, 12-5, 12-11, 13-10
- {virtual} tagged values 1-6
- <\$OBJING_PATH> 3-3
- <<create>> stereotype 2-8
- <<destroy>> stereotype 2-8
- <<throw>> stereotype 12-6
- Access methods 11-5
 - Complete access methods 11-6
 - Full access methods 10-3, 10-9
 - Modification methods 10-3, 10-9, 11-6
 - Read methods 10-3, 10-9, 11-6
- Accessing a parameter 16-10
- Adapting C++ generation 18-4
 - J parameterization 18-4
 - Tagged values 18-4
 - Typed texts 18-4
- Adapting makefile generation
 - Procedure 16-6
- Adapting Objectteering C++ code generation
 - Adding new commands 15-3
 - Adding new project parameters 15-3
- Defining new C++ specific notes 15-3
- Defining the basic types project 15-3
- Specializing generation behavior 15-3
- Adapting the makefile
 - Changing standard module parameter values 16-3
 - Redefining a document template 16-3
 - Redefining existing document template methods 16-3
- Adding a C++ note to an operation 5-7
- Analyzing compilation 4-27
- Associated
 - Generated zones of code 15-36
- Association
 - Access methods 11-6
 - Definition 11-3
 - Generation 11-5
 - Initialization 15-61
 - Multiplicity 11-5
 - Visibility 11-7
- AssociationEnd metaclass 15-30
- Associations 1-9
 - "forward" inclusions and declarations 15-32
 - Generated zones of code 15-30
 - Instance management 15-35
 - Name of access methods 15-34
 - Parameter types 15-31
 - Parameterizing 15-30
 - Producing code 15-33
 - Redefining the CR_Set macro 15-32
 - Return parameter types 15-31

- Attached tools
 - Configuration 7-16
- Attribute
 - Base type 10-3
 - Definition 10-3
 - Enumerate 10-3
 - Generation 10-3
 - Initialization 10-8
 - Modifying the C++ name 15-60
 - Primitive class 10-3
- Attributes 1-9
 - "forward" inclusions and declarations 15-28
 - Generated zones of code for a list type attribute 15-26
 - Generated zones of code for a simple attribute 15-25
 - Generated zones of code for an attribute annotated {array} 15-27
 - Name of the access methods 15-29
 - Name of the method used to obtain multiplicity 15-29
 - Parameterizing 15-24
 - Producing code 15-29
 - Redefining the CR_Set macro 15-28
- Basic types 13-3
 - Boolean 13-4
 - Char 13-4
 - Character 18-5
 - Float 18-5
 - Integer 13-4, 18-5
 - Real 13-4
 - String 13-4
- Body class 3-15
- Body file 8-5, 9-3, 9-4, 10-7, 15-17, 15-19, 15-49
- Body files 1-7
- Boolean conditions 9-7
- Built types
 - "class" type 18-6
 - "enum" type 18-7
 - "struct" type 18-6
 - "typedef" type 18-7
 - "union" type 18-6
- C++ 12-12
- C++ code generated by
 - Objectteering/UML 18-3
 - Method implementation 18-3
 - Specific Objectteering/UML concepts 18-3
 - The declarative section 18-3
- C++ code generation 4-3, 4-21
- C++ code generation and model consistency checks 4-21
- C++ namespace 8-3
- C++ note 2-8, 12-13, 13-6
 - C++PublicMember 9-17
- C++ note contents 5-9
- C++ note types 5-9
- C++ notes 4-25, 4-28, 5-3, 8-5, 9-4, 9-5, 10-3, 10-5, 11-4, 15-3
 - C++ 5-10
 - C++BodyHeader 9-15
 - C++ConstructorTransmission 5-10
 - C++Inheritance 9-16
 - C++InterfaceHeader 9-15
 - C++PrivateMember 9-16
 - C++ProtectedMember 9-17
 - C++Returned 5-10
 - MFCMessageMacro 9-18
- C++ properties
 - Const attribute 5-4
 - Inline method 5-4

- Virtual method 5-4
- C++ references used 18-3
- C++ sources 4-3
- C++ stereotypes 8-5
- C++ syntax 4-24
- C++ tagged value
 - The {bind} tagged value 13-9
- C++ tagged value types 5-6
- C++ tagged values
 - The {nocode} tagged value 8-6
 - The {noCodeForAll} tagged value 8-7
- C++ tagged values 5-3, 9-3, 10-4, 11-3
 - The {C++Name} tagged value 10-16
 - The {mutable} tagged value 10-16, 11-13
 - The {noInline} tagged value 10-16
 - The {own} tagged value 10-16
- C++ UML profile 16-6, 16-11, 16-12
- C++AccessDecl note 10-5, 10-18, 11-4
- C++AccessDef note 10-5, 10-18, 11-4
- C++BodyHeader 2-6
- C++BodyHeader note 2-5, 8-5, 9-4, 9-5
- C++ConstructorTransmission 12-12
- C++ConstructorTransmission note 12-6, 12-12
- C++DefaultValue note 12-9, 12-20
- C++Inheritance note 9-4
- C++InterfaceHeader 2-6
- C++InterfaceHeader note 2-5, 8-5, 9-4, 9-5
- C++Invariant 2-6
- C++Invariant constraint 2-5
- C++Invariant stereotype 8-5
- C++ParamExpr note 12-9, 12-21
- C++PostCondition 12-14
- C++PostCondition note 2-8
- C++PreCondition 12-14
- C++PreCondition note 2-8
- C++PrivateMember note 9-4
- C++ProtectedMember note 9-4
- C++PublicMember note 9-4
- C++Returned 12-12
- C++Returned note 2-8, 12-6, 12-13
- C++TypeExpr note 2-10, 2-12, 10-5, 10-17, 12-9, 12-20, 13-12
- C++Value note 10-5, 10-8, 10-17, 13-12
- Calling commands
 - Syntax 6-4
- Class
 - Adding an inclusion 15-59
 - Declaration 15-23
 - Encapsulation 15-23
 - forward inclusions and declarations 15-22
 - Generated zones of code 15-20
 - Initialization before generating code 15-21
 - Invariant 15-23
 - Modifying constants 15-58
 - Parameterizing 15-19
 - Producing code 15-22
 - Redefining the CR_Set macro 15-21
 - void start method 9-9
- Class attribute 10-7
- Class generalization 9-3
- Class invariant 9-3
 - Controlling an invariant clause 9-7

- Class invariants 12-14
- Class methods 18-10
- Class template
 - Instantiation 9-12
- Classes 1-7
- Clauses 9-7
- Code zones 4-25, 4-28
- Code/model consistency 1-3
- Command line mode 4-21
- Comparison window 4-26
- Compilation analysis 3-26
- Compilation errors 1-3
- Compilation results
 - Error messages 4-28
 - Information messages 4-28
 - Warning messages 4-28
- Compilation work product 1-5, 1-8, 3-16, 3-20, 3-25, 4-3, 4-19, 4-29
 - Accessing the compilation work product 16-7
 - Attributes 16-8
 - Context menu 4-18, 4-27
 - Creating a compilation work product 3-16
 - Definition 4-4
 - Detailed description 4-8
 - Dialog box 4-19
 - Entering a compilation work product 3-18
- Compilation work product dialog box
 - Definition tab 4-10
 - Excluded classes tab 4-15
 - Header tab 4-11
 - Libraries to be linked tab 4-12
 - Properties tab 4-8
- Compilation work products 7-4, 7-5
- Compiling the generated file 3-25
- Configuring the Objectteering/C++ module 7-3
- Consistency checks 4-21
- Constants
 - AccessMethodPrefix 15-8
 - AppendMethodPrefix 15-8
 - AttributeDeclarationsComment 15-11
 - AttributePrefix 15-8
 - BodyBeginningComment 15-10, 15-11
 - BodyEndComment 15-10, 15-11
 - CardinalMethodPrefix 15-8
 - ClassInvariantCheckMethodName 15-7
 - ConstantDefinitionsComment 15-10, 15-12
 - CxxAtt_ConstReturnString 15-15
 - CxxAtt_RefReturnString 15-15
 - CxxBoolean 15-9
 - CxxChar 15-9
 - CxxDCR_ConstMethodString 15-15
 - CxxDCR_IndexMethodString 15-15
 - CxxFileSuffix 15-9
 - CxxInteger 15-9
 - CxxObj_CR_Check 15-13
 - CxxObj_CR_Endif 15-13
 - CxxObj_CR_Trace 15-13
 - CxxReal 15-9
 - CxxString 15-9
 - EnumerateDeclarationsComment 15-10, 15-12
 - EraseMethodPrefix 15-8
 - ForwardClassDeclarationsComment 15-10
 - FreeCodeComment 15-10, 15-12

- FullAccessMethodPrefix 15-8
- FullSetMethodPrefix 15-8
- HxxFileSuffix 15-9
- indentBeginEnd 15-14
- indentBodyMember 15-14
- indentClass 15-14
- indentCode 15-14
- indentHeaderMember 15-14
- indentPreprocessor 15-14
- indentVisibility 15-14
- InheritedSchemaIncludesComment 15-12
- InterfaceBeginningComment 15-10, 15-11
- InterfaceEndComment 15-10, 15-11
- InvariantDeclarationsComment 15-11
- InvariantDefinitionComment 15-12
- InvariantDefinitionsComment 15-11
- MethodComment 15-11
- MethodDeclarationsComment 15-11
- RelationshipDeclarationsComment 15-11
- SchemaInvariantCheckMethodName 15-7
- SetInstantiationsComment 15-10
- SetMethodPrefix 15-8
- TypeDeclarationsComment 15-10, 15-12
- Constraints 5-3, 15-5
 - C++Invariant 2-6
 - C++Invariant constraint 2-5
 - C++PostCondition 12-15
 - C++PreCondition 12-14
- Constructors 10-8
- Context menu 4-16
- Context menu commands 4-17, 4-18, 4-29
- Context menu functions
 - Attribute management 4-19
 - File related services 4-19
 - Propagation action 4-19
- Correcting errors 4-28
- CR_Boolean
 - Description 17-5
 - Example 17-5
- CR_string
 - Description 17-6
 - Example 17-6
- Creating a compilation work product 3-16
- Creating a generation work product 3-6
- Creating a tagged value 5-4
- Creating a UML modeling project 3-4
- Creating an invariant clause 9-8
- Creating constraints 5-10
- CRType
 - Name 15-48
 - Parameterization 15-48
 - Prefix 15-48
- currentProduct variable 16-7
- Default model types 2-4
- DefaultTranslations
 - FiniteAssociation 14-15
 - FiniteAttribute 14-15
 - FiniteIOPParameter 14-15
 - FiniteReturnParameter 14-15
 - MandatoryMultipleAssociation 14-15
 - MandatorySimpleAssociation 14-15

- MultipleAttribute 14-15
- MultipleLOParameter 14-15
- MultipleReturnParameter 14-15
- OptionalMultipleAssociation 14-15
- OptionalSimpleAssociation 14-15
- SimpleAttribute 14-15
- SimpleLOParameter 14-15
- SimpleReturnParameter 14-15
- Defining a tagged value 18-11
- DescribedCRElement
 - Class to be used 15-50
 - Name of the parameter 15-50
 - Parameterization 15-50
- DescribedCRElement
 - Post-condition 15-51
 - Pre-condition 15-51
- Design patterns 1-9
- Development steps
 - Annotating model elements 1-6
 - Creating a UML model 1-6
- Different kinds of member functions 12-3
- Directories
 - Configuration 7-9
- Document template 16-3
- Document templates
 - Details 16-5
- Dynamic dependent attribute 10-8
- Dynamic library 4-4
- Encapsulation 10-9
- Enterprise edition 1-9, 6-3, 15-3
- Enumerate 13-3
 - Definition 13-7
- Enumeration
 - Parameterization 15-52
 - Prefix 15-52
 - Values 15-53
- Enumeration type dialog box 13-8
- Error messages 12-20
- Error messages in the compiler 1-8
- Explorer 1-10
- External edition
 - Configuration 7-5
- External editor 4-22, 4-24, 4-25, 5-7
- Faulty invariant clauses 9-8
- Friendly operators
 - Definition in Objectteering/UML 18-12
 - Specific parameterization 18-13
 - Using friendly operators 18-12
- General set libraries 17-8
- Generalization 9-4, 9-5, 13-3
 - "forward" inclusions and declarations 15-55
 - Definition 13-9
 - Parameterization 15-54
 - Parent class 15-54
 - Public or protected visibility 18-14
- Generated code header 3-13
- Generating C++ code 3-9
- Generating more than one executable from your model 4-3
- Generating the makefile 4-21
- Generation attributes 4-3
- Generation constant
 - J methods 15-6
- Generation options
 - Configuration 7-6
- Generation work product 1-5, 1-7, 3-6, 4-3, 4-19, 4-21, 4-25, 4-29
 - Context menu 4-16, 4-17
 - Definition 4-3
 - Dialog box 4-19

- Generation work product dialog box 4-5
- Generation work products 7-4, 7-5
- Grouping compilation attributes 4-4
- Header
 - Method 15-49
 - Parameterization 15-49
- Header file 8-5, 9-4, 15-49
- Header files 1-7, 15-58
- Importing the contents of the C++ first steps project 3-4
- Initializing the First Steps UML modeling project 3-4
- Installing the Objectteering/C++ module 2-3
- Instance
 - Definition 13-11
- Instance handling
 - Declaration 11-8
 - Definition 11-8
 - Generation 11-8
- Interface file 9-3
- Invariant clause controls 9-7
- Invariant clauses
 - Creating an invariant clause 9-8
 - Faulty invariant clauses 9-8
- Invariant rules 8-3
- J language 15-3, 16-8
- J methods 1-9, 15-3, 15-16, 15-24, 15-30, 15-45, 16-10
- Libraries
 - Configuration 7-13
- Linking an application to the Objectteering/UML library 17-4
- Main class 9-3
- Maintaining code/model consistency 4-3
- Makefile 3-20, 3-22, 3-24, 4-3, 4-4, 4-21, 4-29
 - Visualizing 3-22
- Makefile generation document template
 - Main principles 16-4
- Makefiles 1-3, 1-8, 3-19
- Managing a work product's attributes 4-19
- Markers 4-24, 4-25
- Member
 - Parameterization 15-56
 - Visibility of an attribute's accessors. 15-57
 - Visibility of an attribute's modification features. 15-56
 - Visibility of the attributes 15-56
- Method
 - Adding C++ code 15-62
- MFC macro generation 9-3
- MFC naming rules 10-9, 11-6
- MFC specific macros 9-13
- MFCMessageMacro note 9-4
- MFCs 8-7, 9-3, 9-13, 11-6
 - Configuration 7-17
- Microsoft foundation classes 1-9
- Model dialog boxes 5-7
- Model root 3-5
- Modifying a parameter value 16-10
- Modifying the configuration of module parameters 2-3
- Module configuration groups
 - Attached tools 7-4
 - Directories 7-4
 - External edition 7-4
 - Generation options 7-4
 - Libraries 7-4

- MFC 7-4
- Production options 7-4
- Suffixes 7-4
- UML profiles 7-4
- Visibility declarations for attributes and associations 7-4
- Visibility for accessors 7-4
- Module parameters 16-9
 - Accessing parameters 16-10
 - Modifying a parameter value 16-10
- NameSpace metaclass 16-11
- Naming and filtering constants
 - Accessor prefixes 15-8
 - Basic types name 15-9
 - Comment zones 15-10
 - Invariant methods 15-7
 - Line indents 15-14
 - Passing modes for operation parameters 15-15
 - Pre-compilation tagged values 15-13
 - Suffix files 15-9
- Naming rules 14-14
- Non-primitive objects 17-11
 - Obtaining a set of objects 17-11
- Notes 1-3, 1-4, 1-6, 1-10, 5-7, 15-5
 - C++ 4-22
 - C++ note 2-8, 12-6, 12-13, 13-6
 - C++AccessDecl note 10-18, 11-4
 - C++AccessDef note 10-18, 11-4
 - C++BodyHeader 1-6, 2-5, 2-6, 4-22
 - C++BodyHeader note 8-5, 9-4, 9-5
 - C++ConstructorTransmission 4-22
 - C++ConstructorTransmission note 12-6, 12-12
 - C++DefaultValue note 12-9, 12-20
 - C++Inheritance note 9-4
 - C++InterfaceHeader 2-5, 4-22
 - C++InterfaceHeader note 8-5, 9-4, 9-5
 - C++ParamExpr note 12-7, 12-9, 12-21
 - C++PostCondition note 2-8
 - C++PreCondition note 2-8
 - C++PrivateMember 4-22
 - C++PrivateMember note 9-4
 - C++ProtectedMember 4-22
 - C++PublicMember 4-22
 - C++PublicMember note 9-4
 - C++Returned 4-22
 - C++Returned note 2-8, 12-6, 12-13
 - C++TypeExpr note 2-10, 2-12, 10-5, 10-17, 12-7, 12-9, 12-20, 13-12
 - C++Value 10-17
 - C++Value note 10-5, 10-8, 13-12
 - Entry 5-7
 - MFCMessageMacro note 9-4
- Notes associated with a translation class 14-11
- Notes for a translation class 14-13
- Notes on a parameter 12-9
- Notes on an instance 13-12
- Notes on an operation 12-6
- Object instance 13-3
- Objecteering/Document Template Editor 16-3, 16-4
- Objecteering/Introduction 3-3, 15-5
- Objecteering/UML console 12-20
- Objecteering/UML explorer 3-9
- Objecteering/UML Modeler 1-10
- Objecteering/UML Profile Builder 1-9, 15-3, 16-3

- Objecteering/UML repository 1-7, 1-8, 4-25
- ObjecteeringTypes UML modeling project 9-9
- Objingcl 6-3
- Obtaining a binary from a model 4-3
- Operation
 - Declaration 12-5
 - Generation 12-12
- Operation implementation 5-10
- Operations
 - "forward" inclusions and declarations 15-40
 - Adding a prefix to an operation's declaration 15-41
 - Constructor 15-39
 - Destroyer 15-39
 - Free text zones to be inserted 15-42
 - Generated zones of code 15-38
 - Parameterizing 15-37
 - Producing code 15-41
 - Return parameter 15-40
 - Trace end message 15-44
 - Trace starting message 15-43
 - Virtual destructor 15-44
- Organizing classes 4-3
- Organizing your model 4-21
- Package
 - C++ constraint stereotypes 8-5
 - Initialization before generating code 15-18
 - Parameterizing 15-17
 - Producing code 15-18
 - Zones of generated code 15-17
- Package metaclass 16-12
- Packages 1-7
- C++ notes 8-5
- C++ tagged values 8-4
- Correspondence 8-3
- Definition 8-3
- Generalization and use 8-4
- Initializing instances 8-4
- Invariant 8-3
- Methods 16-11, 16-13
- Parameter
 - "forward" inclusions and declarations 15-47
 - Default generation 12-8
 - Definition 12-7
 - Generated zones of code 15-45
 - Modifying the passing mode 15-62
 - Passing mode 12-7
 - Re-defining the CR_Set macro 15-46
- Parameter implementation notions
 - Passing values 12-7
 - Pointers 12-7
 - References 12-7
- Parameter types
 - Basic types 12-7
 - Classes 12-7
 - Enumerates 12-7
- Parameterization
 - Mechanism 15-16
 - Return value 15-16
- Parameterizing basic types 1-9
- Parameterizing by creating a type project 1-9
- Parameterizing the Objecteering/C++ module 1-9
- Parameterizing through tagged values 1-9

- Parameterizing using Objecteering/UML Profile Builder 1-9
- Parameters 1-9
 - Parameterization 15-45
- Post-conditions 5-3, 5-10, 12-11
- Pre-conditions 5-3, 5-10, 12-11
- Primitive objects 17-8
 - boolean 17-8
 - integer 17-8
 - string 17-8
- Primitive types
 - CR_boolean 17-3
 - CR_String 17-3
 - Set 17-3
- Private and protected generalization 9-6
- Production options
 - Configuration 7-14
- Propagation 4-19, 4-21, 4-25
- Properties editor 3-7, 5-4, 9-5, 9-8
 - Adding notes 1-10
 - Adding stereotypes 1-10
 - Adding tagged values 1-10
 - C++ tab on a class 2-6
 - C++ tab on a package 2-5
 - C++ tab on a parameter 2-12
 - C++ tab on an association 2-11
 - C++ tab on an attribute 2-9
 - C++ tab on an operation 2-7
 - Overview 1-10
 - Tabs 1-10
- Read only mode 4-29
- Retrieving edited code 4-24
- Running compilation 4-4
- S package 8-3, 8-6
- Selecting the C++ default model type 2-4
- Selecting the Objecteering/C++ module 2-3, 3-4
- Sets
 - Primitive objects 17-9
- Standard template library 1-9
- Static attributes 10-7
- Static library 4-4
- Static member variable 11-5
- Stereotypes 1-10, 5-3
 - <<create>> stereotype 2-8
 - <<destroy>> stereotype 2-8
 - <<throw>> stereotype 12-6
 - C++Invariant stereotype 8-5
- Suffixes
 - Configuration 7-11
- Tagged values 1-3, 1-4, 1-6, 1-9, 1-10, 5-4, 13-5, 15-5
 - {&} tagged value 2-10, 10-3, 10-11, 12-8, 12-16
 - {*} tagged value 2-10, 10-3, 10-11, 12-8, 12-16
 - {access} tagged value 10-3, 10-11, 11-9, 14-11
 - {alias} tagged value 15-60
 - {arr ay} tagged value 10-12
 - {array} tagged value 1-6, 10-3, 11-3, 11-9, 12-8, 12-16
 - {autoinit} tagged value 15-61
 - {bind} tagged value 9-11, 10-3, 10-15, 11-10, 12-8, 12-17, 13-9, 13-13, 18-6
 - {C++DLL} tagged value 9-3, 9-14
 - {C++Name} tagged value 2-5, 2-6, 2-8, 2-10, 2-11, 2-12, 8-4, 8-7, 9-3, 9-14, 11-3, 11-12, 12-5, 12-8, 12-10, 12-17

{C++NoNameSpace} tagged value 2-5, 8-4, 8-7
 {C++Root} tagged value 2-5, 8-4, 8-7
 {const} tagged value 2-10, 10-3, 10-12, 12-8, 12-17
 {create} tagged value 10-3, 10-8, 10-12, 11-3, 11-12
 {extern} tagged value 2-5, 2-6, 8-4, 8-6, 9-3, 9-13
 {fullaccess} tagged value 10-3, 10-13, 11-3, 11-10, 14-11
 {fullmodify} tagged value 14-11
 {generic} tagged value 11-3
 {inline} tagged value 2-8, 12-5, 12-10
 {instanceHandling} tagged value 11-3, 11-8, 11-12
 {Jeval} tagged value 14-8, 14-9
 {long} tagged value 10-3, 10-13, 12-8, 12-18, 13-5
 {MFCDynamicMacro} tagged value 9-3, 9-13
 {MFCInclude} tagged value 8-4, 8-7
 {modify} tagged value 14-11
 {nocode} tagged value 2-5, 2-6, 2-10, 2-11, 8-4, 9-3, 9-14, 10-3, 10-16, 11-3, 11-11, 12-5, 12-10
 {noCodeForAll} tagged value 9-14
 {noconst} tagged value 10-3, 10-15, 11-3, 11-12, 12-5, 12-8, 12-10, 12-18
 {noinclude} tagged value 8-4, 8-6, 9-3, 9-13
 {noinit} tagged value 11-3, 11-10
 {noInline} tagged value 10-4, 11-3
 {noinvariant} tagged value 12-5, 12-11
 {own} tagged value 2-10, 2-11, 11-3
 {private} tagged value 9-6, 13-10
 {protected} tagged value 9-6, 13-10
 {public} tagged value 2-10, 2-11, 10-3, 10-7, 10-14, 11-3, 11-11
 {short} tagged value 10-3, 10-13, 12-8, 12-18, 13-5
 {structure} tagged value 9-3, 9-13
 {type(...)} tagged value 2-10, 2-11, 2-12
 {type} tagged value 10-3, 10-15, 11-3, 11-12, 12-8, 12-19, 14-9
 {unsigned} tagged value 10-3, 10-14, 12-8, 12-19, 13-5
 {userpassing} tagged value 15-62
 {val} tagged value 12-8, 12-19
 {virtual} tagged value 1-6, 2-8, 5-5, 9-5, 11-3, 11-11, 12-5, 12-11, 13-10
 The {noInline} tagged value 11-13
 The {own} tagged value 11-13
 Tagged values on a generalization 13-9
 Tagged values on a parameter 12-8
 Tagged values on an instance 13-13
 Tagged values on an operation 12-5
 Template class
 Implementation 9-11
 Template class 9-3, 9-11
 Declaration 9-11
 The C++ UML profile 15-4
 The properties editor for C++ 2-4
 Translation class naming rules 14-10
 Type
 Correspondence 13-4

Typed text 18-10
UML model types 2-4
UML profiles 4-3, 15-3
 Configuration 7-12
 The C++ UML profile 15-4
Use links on an operation 12-6
Using the Objectteering/C++ module
 Installing the module 2-3
 Selecting the module 2-3
Virtual generalization 9-5, 18-14
Visibility for accessors
 Configuration 7-19
Visibility for attributes and associations
 Configuration 7-18
Visibility of access methods 10-10
Visualizing the body of the generated
code 3-14
Visualizing the header of the
generated code 3-11
Visualizing the makefile 4-29