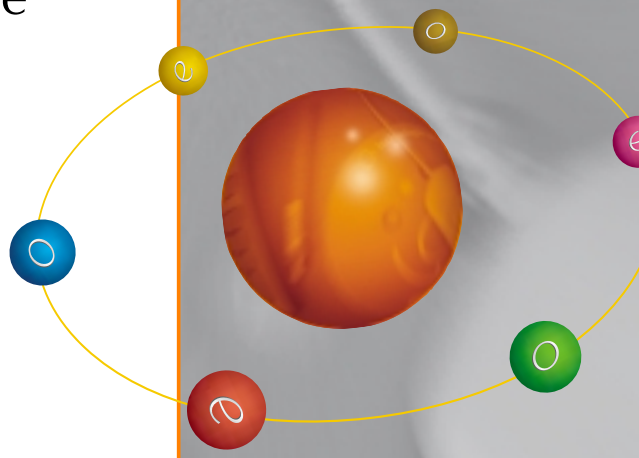
Objecteering/UML

Objecteering/C++ Reverse User Guide

Version 5.2.2



Objecteering Software

www.objecteering.com

Taking object development one step further

Objecteering/UML version 5.2.2 - CODOBJ 001/001

Contents

Chapter 1: Overview

Principle

Overview

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

The *Objecteering/C++ Reverse Engineering* module is used to reconstruct a model in Objecteering/UML, from the analysis of C++ code carried out using the *SNiFF+* tool.

The Objecteering/UML *Reverse Engineering* tool works in two distinct phases:

- The first phase analyzes the C++ code which is to be reversed. This first phase is divided into two steps, the first of which consists of creating a *SNiFF+* project (version 3.2) containing the C++ sources of the classes which are to be reintroduced into Objecteering/UML, and the second of which is the launch, in Objecteering/UML, of the code analysis phase itself.
- In the second phase, all or some of the classes identified by the code analysis are imported, and the user chooses the mode which best suits his needs:
  - the "*complete import* " mode retrieves the entire contents of classes.
  - the "*interface import*" mode only retrieves the public parts of classes (their interface).
  - the "*structural import*" mode retrieves empty classes, with their links (generalizations, associations, etc) to other classes.

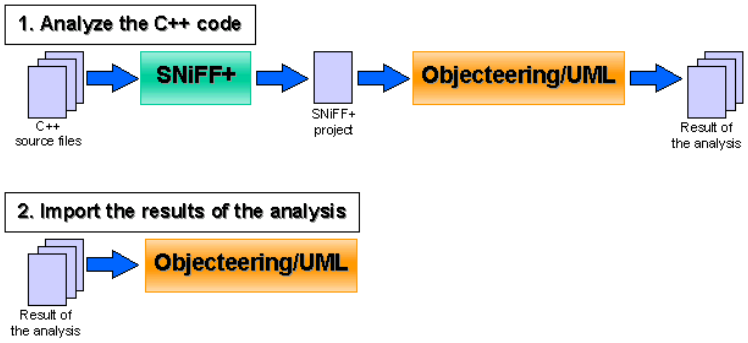


Figure 1-1. Reconstructing a model

Code reverse and model consistency checks

A model can be reversed from files external to Objecteering/UML regardless of whether consistency checks are active or inactive. However, when the reverse is launched, a message informs the user that consistency checks are active and that construction of the model, which may potentially not conform to the UML modeling rules checked by Objecteering/UML, may be refused (as shown in Figure 1-2).

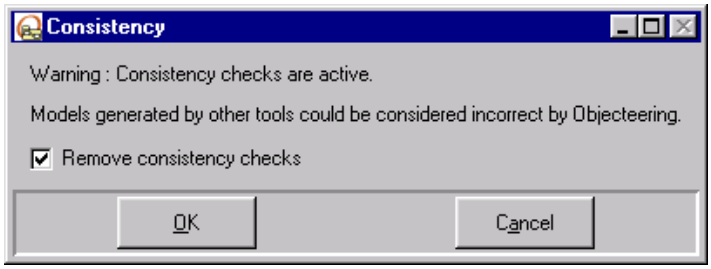


Figure 1-2. Message informing the user that consistency checks are active

If the user wishes to keep consistency checks activated, he should simply uncheck the "*Remove consistency checks*" box.

Note: It should be noted that code reversal in command line mode (please see objingcl) is guaranteed, whatever the state of the consistency checks when the reverse command is run.

Composition of the Reverse Engineering module

The *Objecteering/Reverse Engineering* module is delivered in externalized form, and contains the following elements:

- the "*ReverseTool*" binary (for Solaris) or the "*ReverseTool*.exe" binary (for Windows NT/95/98/2000)
- HTML documentation (the present document)
- SNiFF format configuration files of the Reverse Engineering module
- Sets and types bindings description files

The C++ Reverse Engineering problem

The mechanism which constantly checks consistency between the model and the Objecteering/UML code means that unlike many CASE tools, C++ code generated by Objecteering/UML must never be reversed in Objecteering/UML. If this is done, all the benefit of permanent consistency checks is lost.

The Objecteering/UML reverse is dedicated to operations used to reverse existing code developed in contexts other than Objecteering/UML, often manually, and used for various ends, such as reverse documentation, code reversal and development advancement, or the reverse of a library to be used from Objecteering/UML. For example, MFCs (Microsoft Foundation Classes) are provided in reversed form.

The task of reversing an existing C++ source resulting from external sources is not an easy one. It can almost correspond to the transformation of sources, so as to be able to compile and analyze them. The SNiFF tool (from the TakeFive company) helps organize and analyze C++ sources, which is a preliminary step essential to reverse engineering with the Objecteering/UML tool.

Chapter 2: Using the C++ Reverse Engineering module

Overview

To be able to use the *Objecteering/C++ Reverse Engineering* module, the following operations should be carried out:

- the selection of the module
- the configuration of the module

For further details on selecting the *Objecteering/C++ Reverse Engineering* module, please refer to the "*Selecting modules in a UML modeling project*" section in the current chapter of this user guide.

Selecting modules in a UML modeling project

In order to be able to use the *Objecteering/C++ Reverse Engineering* module, it must be selected for the current UML modeling project. This selection is made by transferring the *C++ Reverse Engineering* module into the right-hand "*Modules used*" column of the "*Modules*" dialog box (as shown in Figure 2-1).

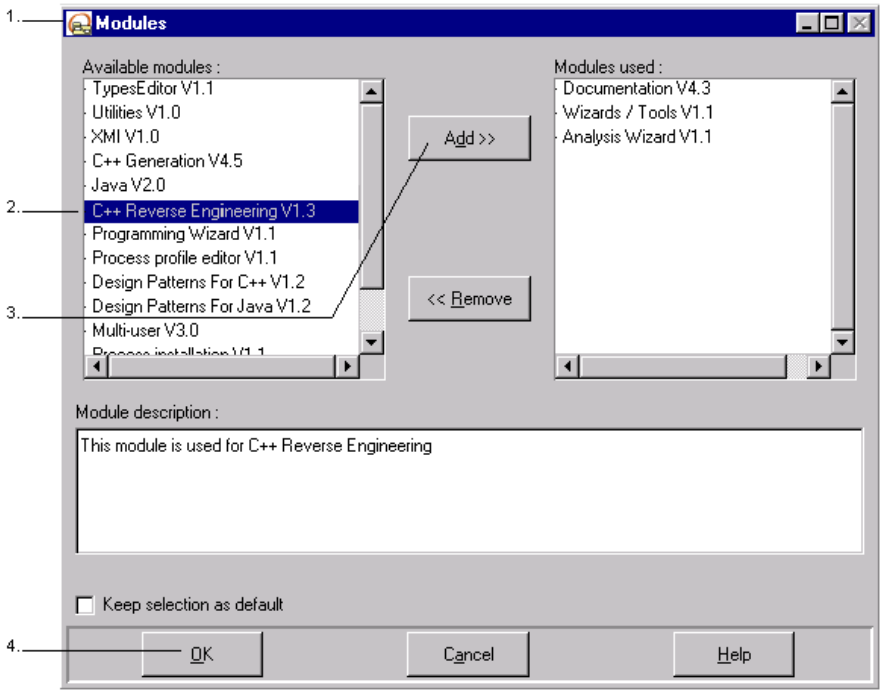


Figure 2-1. Selecting the *Reverse Engineering* module

Steps:

1 - To open the "*Modules*" window, either click on the  "*UML modeling project modules*" icon or click on "*Tools/Modules...*".

2 - Select the "*C++ Reverse Engineering*" module in the left-hand "*Available modules*" column.

3 - Click on the "*Add*" button. The module is then transferred into the right-hand "*Modules used*" column.

4 - Confirm by clicking on "*OK*".

Chapter 3: Configuring SNiFF+ and Objecteering/UML

Configuring SNiFF+

After having installed the tool, a *SNiFF+* project which integrates all the C++ files which are to be taken into account must be created. To do this, carry out the following steps.

In the *SNiFF+* main window, open the "*Project/New Project.../with Template...*" window (as shown in Figure 3-1):

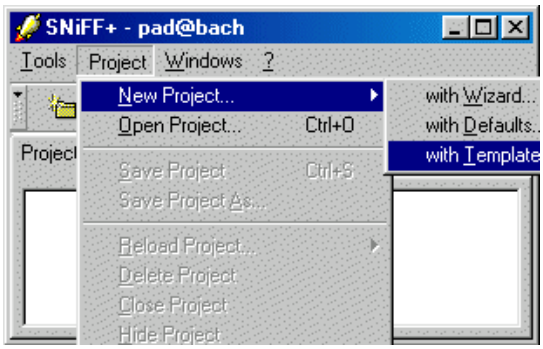


Figure 3-1. The "*Project/NewProject... with Template*"... menu

The following window then opens (as shown in Figure 3-2). Select the "*template 04Reverse.ptmppl*" option, and then click on the "*Change Directory...*" button.

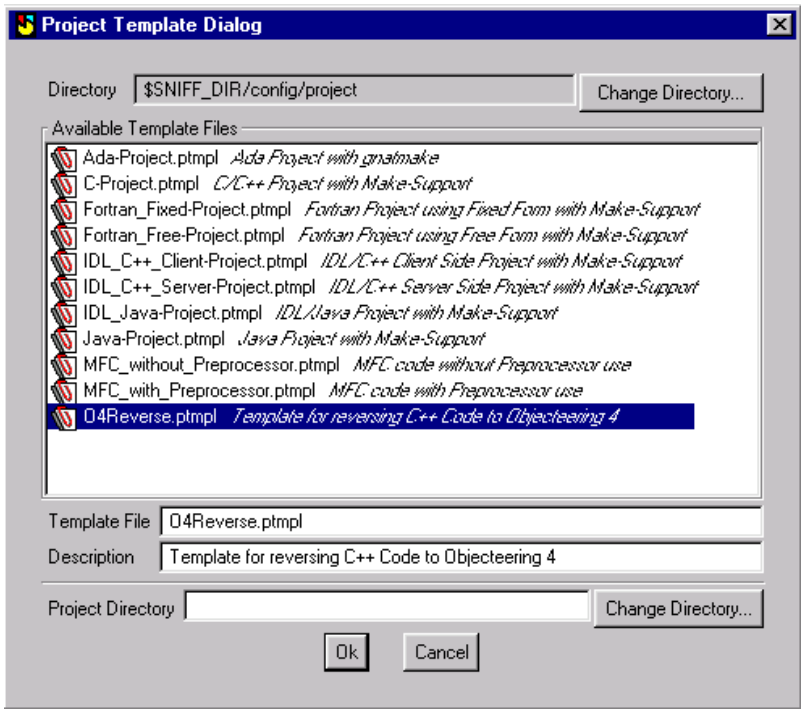


Figure 3-2. The "*Project Template Dialog*" window

Select the directory which contains the sources which are to be reversed, and then click on the "*Select*" button.

At the bottom of the project window, each sub-directory is represented by a nested project. Each sub-project must be checked, if you wish to reverse it. By checking it, you will see the corresponding directory files appear in the top window.

Warning:All sources must be located in the same directory!

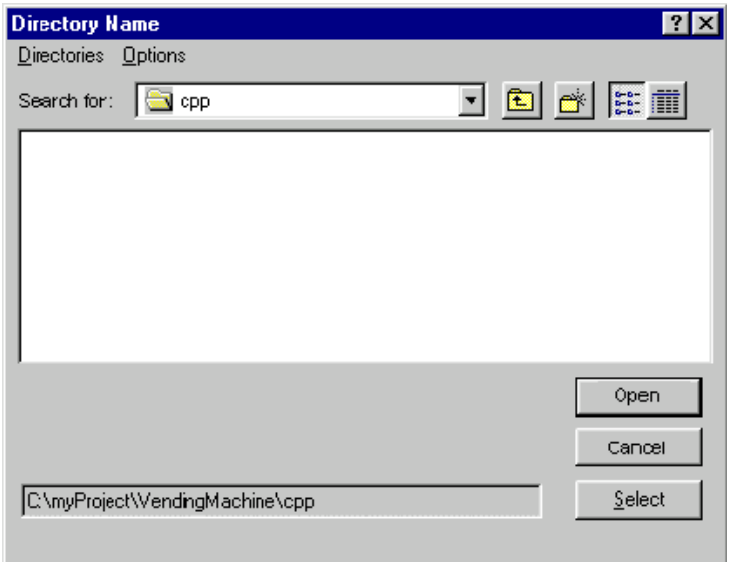


Figure 3-3. Window for the selection of the directory containing the sources which are to be reversed

Once the project has been created, the default attributes have to be modified through the following window (as shown in Figure 3-4). On the right, select the "*Parser*" item. Be sure that the "*Extended Symtab API Positioning*" tickbox is checked.

If you wish your code to be pre-processed before reverse, check the "*Preprocess Source Code before Parsing*" tickbox and then select the "*Directives*" tab in the right window. Continue by providing the directives and includes necessary to the correct parsing of the code.

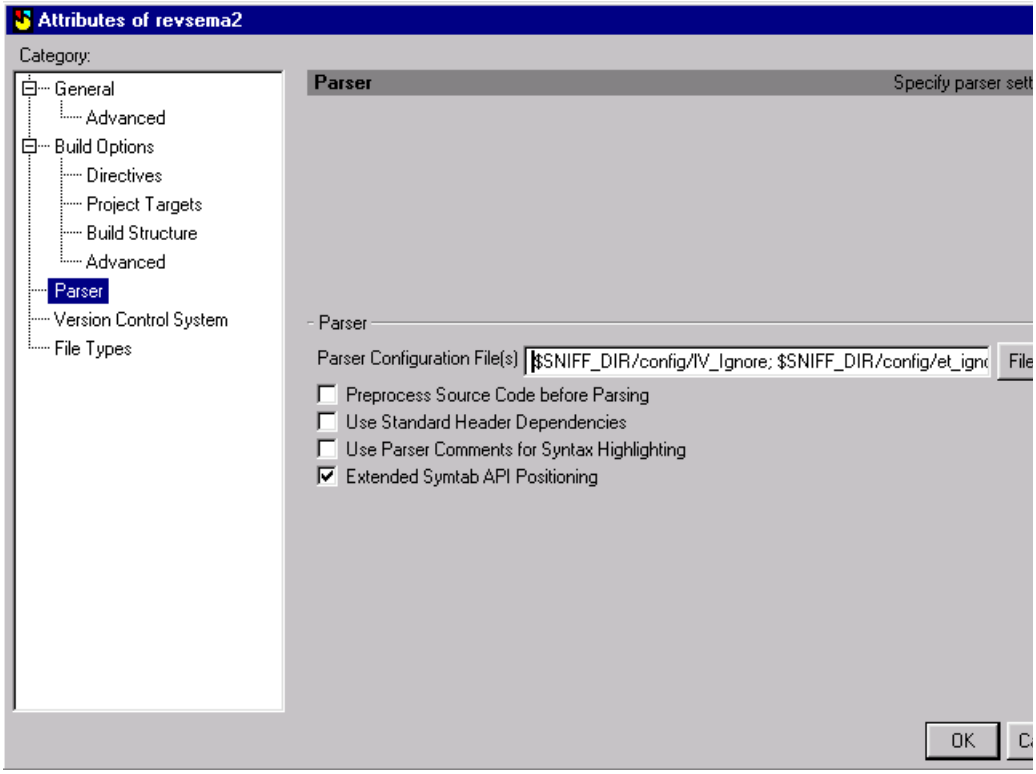


Figure 3-4. Window for modifying the attributes of the new project

Remarks

Note 1: It can be useful not to provide SNiFF+ with include directories, in order to limit the number of classes to be "*reversed*". If your application uses MFCs in Windows, for example, it is not necessary to provide the path of the MFC includes, in order to avoid completely "reversing" the MFCs, which is not the usual aim of this operation. All the classes used by your application and whose definition has not been found by SNiFF+ will be reversed in the form of empty classes.

Note 2: Certain environment add new key words (for example, *_export by Visual C++*). These key words interfere with SNiFF+ parsing, and SNiFF+ therefore provides a means of ignoring them. In the "*Parser Configuration File(s)*" field, enter one or several configuration file names. The standard files are provided in the SNiFF+ "*config*" directory. In particular, in order to avoid problems with MFCs, use the "*winapi_ignore*" file.

Note 3: In UNIX, remember to define in your environment the SNiFF+ "*SNiFF_DIR*" variable and to add access to the SNiFF+ "*bin*" directory to your PATH variable.

Note 4: Pre-processing your code before reversing (by checking the "*Preprocess*" tickbox, as described above) will slow down the reverse process and may give warnings in the SNiFF+ log windows, if all the include paths (see Note 1) have not been given. However, if code is not pre-processed, certain constructs will not be reversed. For example, classes defined in macros (often done for collections) will not be reversed.

Configuring Objecteering/UML

When the *Objecteering/ReverseEngineering* module has been selected in your UML modeling project, a certain amount of information must then be defined (as shown in Figure 3-5).

- *Centralize Reverse Data*: Check this box, in order to have all reverse data centralized in one directory. If this tickbox is not checked, reverse data is stored in a sub-directory of the SNiFF+ project.
- *Centralized Data Directory*: This is the directory where reverse data is stored, if the above tickbox is not checked.
- *Create Diagrams*: If this box is checked, reversed class diagrams are created and opened at the end of the reverse process.
- *SNiFF+ Project*: This is the name of the SNiFF+ project which you created during the previous reverse action.
- *Console command (UNIX only)*: This is the console launch command, which allows you to follow the progress of the process. This command is only necessary in UNIX, since the standard console is used for Windows.

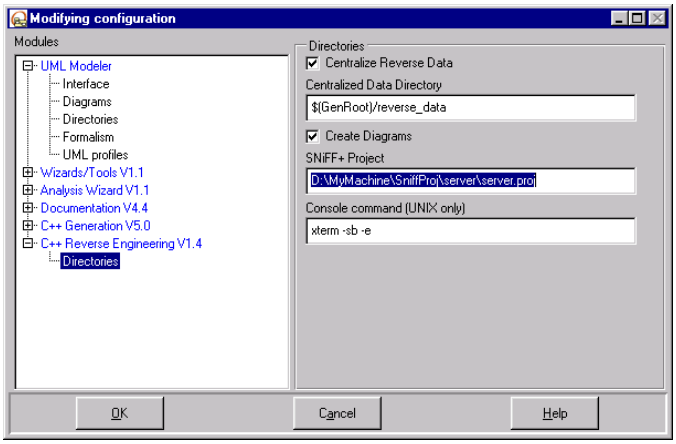


Figure 3-5. Objecteering/UML configuration window

Configuring basic type and collection translation

The Objecteering/UML reverse tool is configured to translate most basic types and collections to their correct UML equivalent. Where specific types or collections are used, Objecteering/UML can be configured to provide the correct translation.

Basic type translation

To configure basic type translations, edit the file named "*TypeModelBindings.ini*" in your reverse data directory (for further details, please refer to the "*Managing identifiers*" section in chapter 4 of this user guide).

Note: This file is copied into this directory during the first reverse. If no reverse operations have yet been carried out, it can be copied from the <OBJING_PATH>/modules/ReverseEngineering/res directory.

The basic file has the following contents:

```
# Type<>Model Bindings

# void types

void                    undefined

any                     undefined

# char types

char                    char

character               char

# integer types

integer                 integer

int                     integer

short                   integertags=short

long                    integertags=long

unsigned                integertags=unsigned

# real types

real                    real

double                  realtags=long

float                   real
```

```
# boolean types

boolean                 boolean

CR_boolean              boolean

bool                    boolean

BOOL                    boolean

# string types

string                  string

CR_string               string

RWCString               string

Cstring                 string
```

As you can see, these contents simply consist of pairs made up of the C++ type name and the corresponding UML basic type. Where necessary, you can add tagged values to the elements using the following syntax:

```
C++Type UMLType tags=UMLTag1,UMLTag2,...
```

For example, the C++ "*short*" type is mapped to the UML "*integer*" type using the *{short}* tagged value. If you have defined the "*PositiveNumber*" type as being an "*unsigned int*", you can define it here as follows:

```
PositiveNumber          integertags=unsigned
```

Please note that in this case, the "*PositiveNumber*" type will no longer be used.

Collections

To configure collection translations, edit the "*SetBindings.ini*" file in your reverse data directory (for further details, please refer to the "*Managing identifiers*" section of chapter 4 of this user guide).

Note: This file is copied into this directory when the first reverse is carried out. If you have not yet performed a reverse operation, it can be copied here from the <OBJING_PATH>/modules/ReverseEngineering/res directory.

The basic file has the following contents:

```
#########
# arrays
#########

@T[]                    size=*tags=array

@T[@S]                  size=@Stags=array

######################
# Objecteering types
######################

# sets

set_of_@T               size=*

# lists

list_@T                 size=*

cr_list<@T>             size=*
```

```
##################################
# Standard Template Library (STL)
##################################

# vector

vector<@T,@A>           size=*type=vector,@A

vector<@T>              size=*

# map

map<@K,@T,@P,@A>        size=*type=map,@K,@P,@A


map<@K,@T,@P>           size=*type=map,@K,@P

map<@K,@T>              size=*type=map,@K

# multimap

multimap<@K,@T,@P,@A>   size=*type=multimap,@K,@P,@A

multimap<@K,@T,@P>      size=*type=multimap,@K,@P

multimap<@K,@T>         size=*type=multimap,@K

# set

set<@T,@P,@A>           size=*type=set,@P,@A

set<@T,@P>              size=*type=set,@P

set<@T>                 size=*type=set

# not implemented in Objecteering

multiset<@T,@P,@A>      size=*type=multiset,@P,@A

multiset<@T,@P>         size=*type=multiset,@P

multiset<@T>            size=*type=multiset

# list

list<@T,@A>             size=*type=list,@A
```

```
list<@T>                 size=*type=list

# not implemented in Objecteering

deque<@T,@A>             size=*type=deque,@A

deque<@T>                size=*type=deque

#####################################
# Microsoft Foundation Classes (MFC)
#####################################

CList<@T>                size=*

CPtrList<@T>             size=*

####################
# Ilog Server (ILS)
####################

IlsSmartPointer<@T>      size=*tags=*

IlsDictionary<@T>        size=*type=map

IlsOwnsList<@@,@T>       size=*type=own

IlsUsesList<@@,@T>       size=*type=uses

IlsInvertedRelationList<@@,@T>  size=*type=mutual
```

As you can see, it simply consists of triples composed of the C++ type name in the form of a pattern, the multiplicity of the collection and the corresponding {*type*} tagged value.

The syntax is as follows:

- To say vector<type> is an association 0..* to type, write:

  ```
  vector<@T> size=* type=vector
  ```

- To say list_type is an association 0..* to type, write:

  ```
  list_@T size=*
  ```

- To say typeP is an association 0..1 to type, write:

  ```
  @TP size=1
  ```

Authorized separators are space and tab.

Note: Fields must contain no spaces.

Warning: Put more precise declarations first, e.g place.

```
vector<@T,@A> size=* type=vector,@A
```

before

```
vector<@T> size=*
```

The predefined symbols used in pattern matching are:

- @@: current class in which the association/attribute is declared
- @T: the referenced type:
- @A: the allocator (STL) :
- @K: the key (STL maps, for example) :
- @S: the size of the element:
- @P: the predecessor comparator (STL) :

Please note that some collections cannot be correctly reversed as their type is not explicit. For example, an MFC CObArray written as follows:

```
class C {
CObArray elems ;
}
```

cannot be transformed into a type in a 0..* association as the type of the elements stored in the array is not known at the time of compilation.

Chapter 4: Reverse engineering functions provided by Objecteering/UML

Description of services on packages

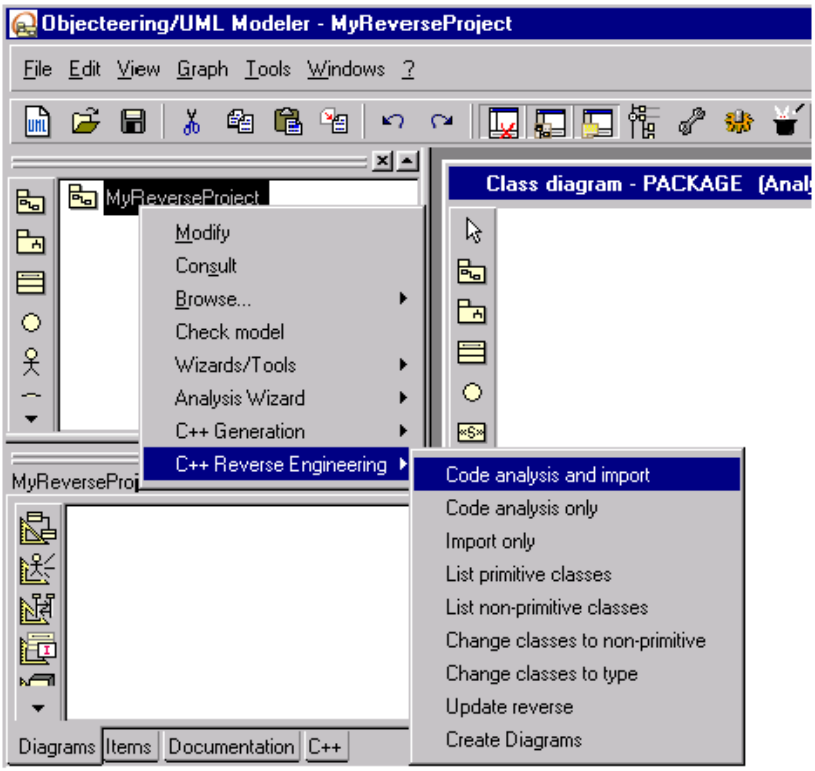


Figure 4-1. Application of "*Reverse Engineering*" to a package

Code analysis services are accessible from packages, as are import services on analyzed classes. "*Reversed*" classes will be added to the package.

A package of the same name as the SNiFF+ project will be created and the "*reversed*" classes will then be added to this new package. Each C++ namespace will be transformed into a package.

The ... command	**triggers ...**
Code analysis and import	the automatic linking of the two reverse phases: "Code analysis" and "Complete import". The import mode is selected through a GUI.
Code analysis only	the first phase of the reverse, which analyzes the C++ code which is to be reversed.
Import only	the second phase of the reverse, which imports a selection of classes and/or packages found during code analysis. The import mode is selected through a GUI.
List primitive classes	the display of the list of the package's primitive classes in the console.
List non-primitive classes	the display of the list of the package's non-primitive classes in the console. .
Change classes to non-primitives	the transformation of selected primitive classes into non-primitive classes. Classes are selected using a GUI which presents primitive classes with at least one Operation and one "extern" tag.
Change classes to type	the transformation of selected primitive empty classes into types. Classes are selected using a GUI which presents primitive classes with no Features and no "extern" tags.
Update reverse	the re-import of the contents of the selected package. This command must be launched on a previously reversed package. It does not re-launch code analysis, but simply allows the readjustment of the import for each imported element.
Create Diagrams	the creation of diagrams on the reversed classes. This menu can be used if the "*Create diagrams*" tickbox has not be checked during module configuration (for further details, please refer to the "*Configuring Objecteering/UML* " section in chapter 3 of this user guide).

When Objecteering/UML cannot determine which SNiFF+ project file to use, the dialog box shown in Figure 4-2 is displayed.

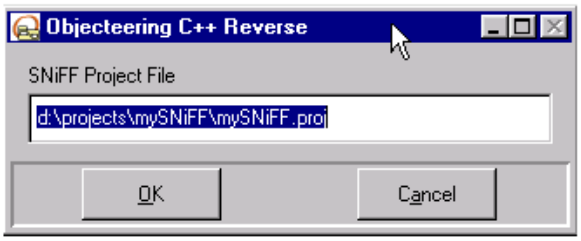


Figure 4-2. Selecting the SNiFF+ project

If you have already run a reverse operation, the last value will either be repeated in the field or else the field itself will be empty. If this is not the case, you should enter the full name of the SNiFF+ to be reversed.

When a reverse operation begins, the mode should be selected in the dialog box shown in Figure 4-3.

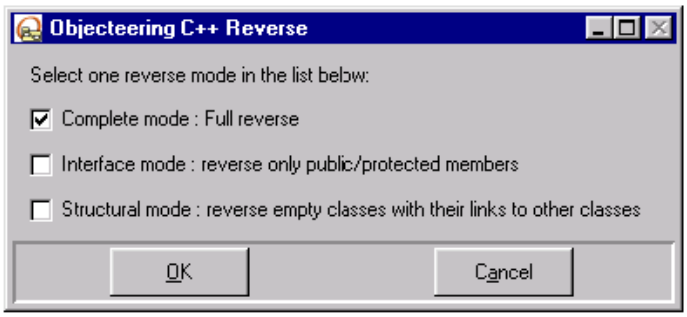


Figure 4-3. Selecting the import mode

Before importing elements into Objecteering/UML, the following dialog box (shown in Figure 4-4) will appear:

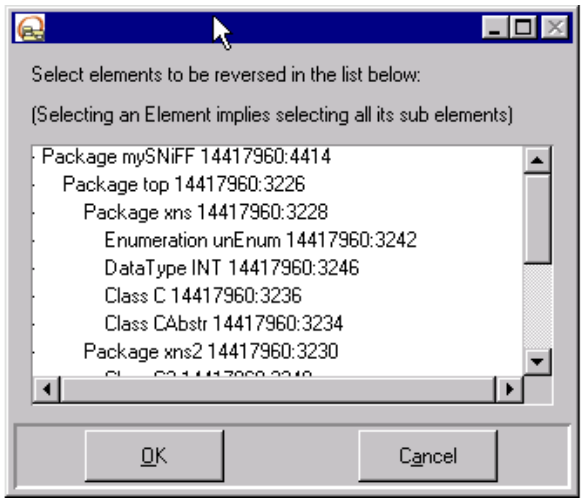


Figure 4-4. Selecting those elements to be imported

The elements are displayed in a tree structure, with one element per line. Each line displays the type, the name and the internal identification of the element. Firstly, the outer package, which contains all the other elements, is found. If you wish to import all reversed elements, you should simply select this package.

However, it is possible to import only one or certain reversed elements. It should be noted that all sub-elements of those elements selected will also be imported. For example, if you select the "*xns*" package, the "*anEnum*" enumeration, the "*INT*" data type and the "*C*" and "*Cabstr*" classes, as well as the "*xns*" package itself, will be imported.

In the complete and interface modes, "*necessary*" elements are also imported. For example, if the "*C*" class specializes the "*CParent*" class in the "*CPack*" package, then the structural versions of "*CPack*" and "*Cparent*" are also imported.

If you try to import only the "*C*" class, the structural versions of the "*xns*", "*top*" and "*mySNiFF*" packages are also imported.

All reversed packages are annotated {*extern*} and "*are directed*" to the SNiFF+ project file from which they were reversed. Subsequent reverses on packages annotated with this tagged value assume that the SNiFF+ project file to be used is that stored in this tagged value. If you wish to use a different SNiFF+ project file, you must delete this tagged value before running the reverse.

Description of services on classes

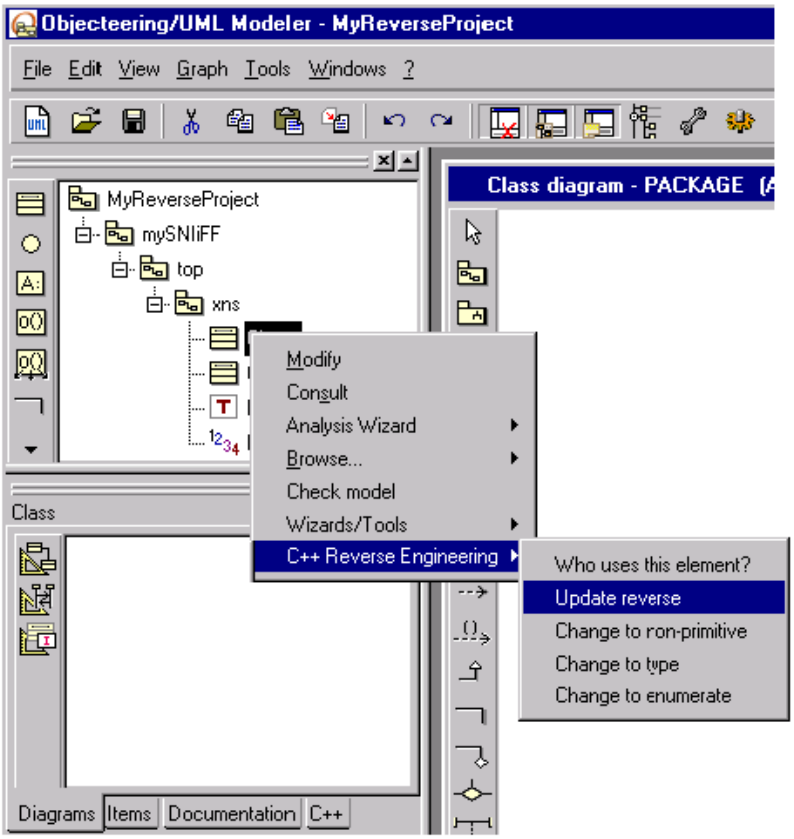


Figure 4-5. Application of the "*Reverse Engineering*" module to a class

Update services are accessible from classes which have already been reversed. These services do not re-launch code analysis, but simply allow the readjustment of the import for each imported class.

The ... command	triggers ...
Who uses this element	the display in the console of classes which use the selected element
Update reverse	the re-import of the class and its sub-elements
Change to non-primitive	the transformation of the selected primitive class into a non-primitive class
Change to type	the transformation of the selected primitive class into a type
Change to enumeration	the transformation of the selected primitive class into an emumeration

All reversed classes are annotated *{extern}* and "*are directed*" to the include C++ file from which they were reversed.

Structures are reversed as classes annotated with the <<structure>> stereotype. Unions are reversed as classes annotated with the <<union>> stereotype.

Managing identifiers

Objecteering/UML stores modeling elements in a database and every element has a unique identifier attributed to it. During a reverse session, the module attributes an identifier to each new element.

Reverse data can be stored either in a centralized or a non-centralized way, depending on whether or not the "*Centralize data*" tickbox has been checked (please refer to the "*Configuring Objecteering/UML*" section in chapter 3 of this user guide for further details). This data is made up of intermediate files produced by the code analysis and certain files used to manage identifiers. These files are SPIdents.ini and ClassIdents.ini and contain the identifiers of each of the reversed elements. Upon subsequent launches, these files are re-read and the identifiers re-used in such a way as not to give several different identifiers to the same object. A log file containing the contents of the console is also generated in the reverse data directory.

In a centralized configuration, these files are re-used over different reverse of different C++ projects. The reversed identifiers are thus shared between different reverses. This is the best choice if you wish to reverse several different C++ projects which use the same classes. However, if two people reverse engineer two different classes with the same name, there may be a collision resulting from two different classes with the same identifier.

In a non-centralized configuration, reverse data is stored in the .o4reverse directory in the SNiFF+ project directory. Identifiers are, therefore, not shared between different SNiFF+ projects.

We recommend that you always use centralized data and different centralized data directories, when reversed C++ projects do not use the same C++ classes.

Known bugs and restrictions

The reverse of some types (*typedef*, *enum*) is implemented by the transformation of the model after reverse. A new reverse either destroys the previous transformation or fails if an element of the same name and of a different type already exists.

The reverse of the following elements is not implemented:

- forward declaration
- include files
- comments
- private, protected and virtual generalization

Typedefs and enumerations not nested either in a class or a namespace are reversed as empty classes and only if they are used.

Classes that have no operations are reversed as primitive classes. The reverse of certain pointer type elements (or reference types) on a "*class*" is wrong.

Conversely, *a*ll reversed elements are accompanied by a "*Reverse*" note. This note, deliberately called "*Reverse*" in order not to interfere with Objecteering/UML generation (documentation, C++ code), contains the information sent back by the SNiFF+ tool. Non-named structures (for example, *struct { int x, y; } pointA, pointB;*) are automatically named and are reversed under the following name:

```
unnamed_<name of the file where the declaration is
located>_<position of the declaration in this file>
```

Default values for parameters and enumerations are not reversed.

Generalization is not reversed when the super class does not belong to the SNiFF+ project.

In Windows, if you cannot connect to SNiFF+ even though the tool is running, you should look at the SNiFF+ session name in the SNiFF+ log window. If this name is not "session0" (which might happen after a system crash, for example), you must either delete the .sniffdir2 in the Windows temporary directory or set the SNiFF_SESSION_ID variable (warning, small 'i') to the real session name and restart the reverse.

Index