2

The EiffelCOM Wizard

2.1 OVERVIEW

COM is a standard that allows software components written in different languages to communicate with each other. Unfortunately, building COM compliant applications requires the developments of huge amount of code only dedicated to support the technology. The EiffelCOM wizard was designed to free programmers from writing all the plumbing code.

The EiffelCOM wizard is a powerful tool that enables the fast development of COM components in Eiffel. It also helps accessing existing COM components from Eiffel systems. It consists of a series of dialogs which ask about the properties of a component. This information is used to produce an Eiffel system skeleton including all the code needed to access or create a component. It also produces component-specific runtime libraries.

The wizard is intended to allow Eiffel developers with little COM knowledge to develop or reuse COM components. The design of the generated code follows the Eiffel standards and should be familiar to any experienced Eiffel user. The only prerequisite to use the EiffelCOM wizard is an understanding of the *Interface Definition Language*. *IDL* is the main tool used to describe a component and can be processed by standard compilers to generate *Type Libraries.* They can be analyzed by tools, such as the EiffelCOM wizard, that need information on a given component. The IDL syntax is very close to C and easy to learn.

The wizard will generate code from a Type Library and additional information given by the user. This code will consist of Eiffel classes, C/C++ files, and library files. The library files are produced automatically from the generated C and C++

code. These are given for information only and you will not need to work with them to build your EiffelCOM system.

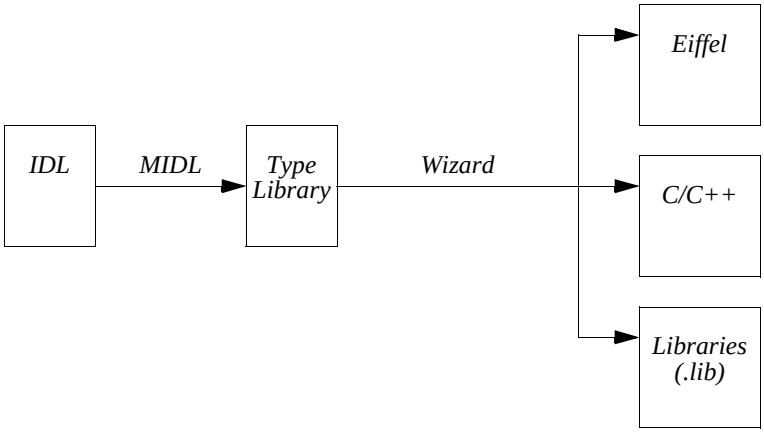

Figure 4: Code Generation Process

The wizard can automatically compile the generated C and Eiffel code. *MIDL*, the Microsoft IDL compiler, is used by the wizard to produce the Type Library corresponding to the given IDL file. You may also provide the wizard with the type library directly. For the remainder of the manual *Definition File* will refer to the input file given to the wizard (either an IDL file or a Type Library).

2.2 THE WIZARD

Let's focus on the wizard itself and the different questions that need to be answered to generate the code. There are five different dialogs but each session will use only four of them. The third dialog is different depending on whether the Definition File is an IDL file or a Type Library. Once the dialogs have been completed, the wizard will start analyzing the Type Library and will eventually generate the code.

Main Window

The EiffelCOM wizard can be launched from the Windows start menu:

Start->Programs->EiffelXX->EiffelCOM Wizard

where *EiffelXX* corresponds to your Eiffel installation (e.g. Eiffel45). The following window will be displayed:

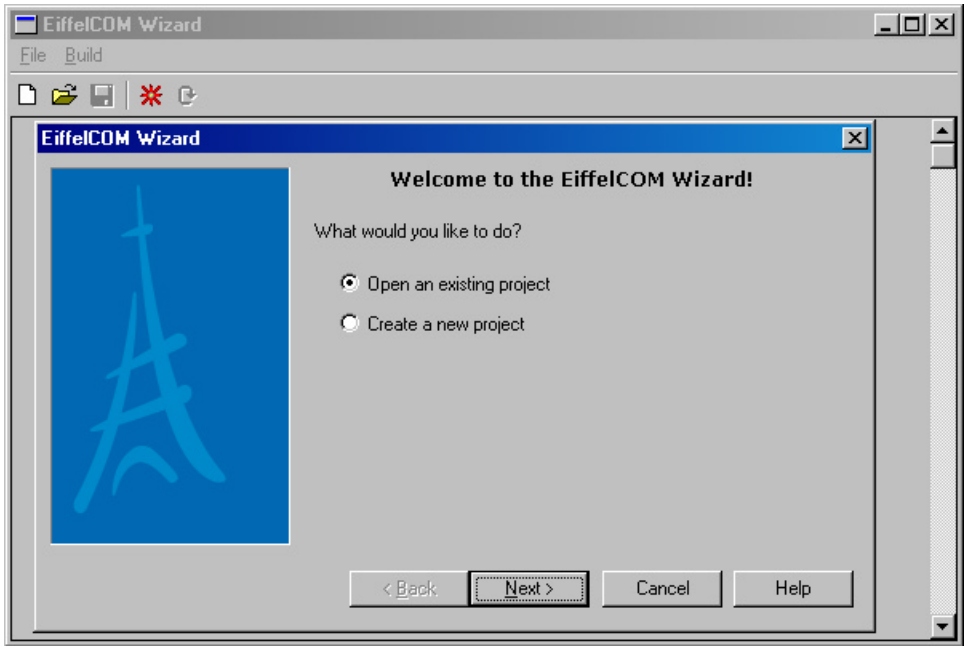

Figure 5: EiffelCOM Wizard Main Window

The introduction dialog lets you choose between opening an existing project or creating a new one. Creating a new project will open the Introduction Dialog. Opening an existing project will display an open file dialog from which you can select a previously saved EiffelCOM project.

The main window includes a toolbar and a menu. The first three buttons on the toolbar correspond to the first three entries in the File menu: *New*, *Open* and *Save*. *New* resets all the information previously entered in the wizard. *Open* brings up an Open File Dialog that can be used to retrieve a previously saved EiffelCOM project. *Save* is used to save the current project. A project is defined by all the values entered in the wizard. A project can be saved only after the wizard has been run. The file extension for an EiffelCOM project is *ewz*.

The second menu, *Build*, includes the entries *Launch Wizard* and *Generate (no wizard)* corresponding respectively to the last two toolbar buttons. The former activates the Introduction Dialog while the later launches the generation with the current settings and bypasses the dialogs. This last button can be used only when a project has been loaded or when the wizard has been run once.

The last four buttons on the dialog *Back*, *Next*, *Cancel* and *Help* are common to all dialogs displayed throughout the execution of the wizard. *Next* validates all the

values entered in the current dialog and activates the next one. *Back* discards all the values entered in current dialog and displays the previous one. *Cancel* exits the dialog and discards all the values entered. Finally, *Help* brings up this manual.

Required File

Before you launch the wizard you need to make sure you have a definition file ready for the component you want to access or create.

Introduction Dialog

The first dialog asks if you want to access or build a component. If you want to access an existing component then the generated code will be for a client. If you choose to build a new one, the generated code will be for a COM server. Choose the server or client check box to specify which kind of project you want to work on. You may specify both in the case where both the component and its client(s) will be written in Eiffel.

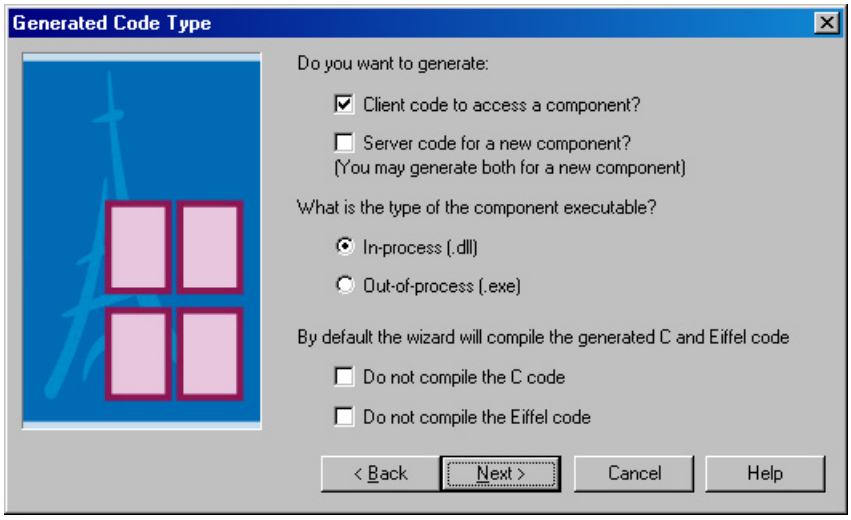

Figure 6: Introduction Dialog

This dialog also asks for the location of the component. EiffelCOM supports all location types:

- *In-Process*: These components are Dynamic Link Libraries (DLLs) that will be loaded inside the client process. The server runs in the same process as the client.
- *Out-of-process*: These components are executable files that can be accessed through the network. Clients and servers run in different processes and may even run on different machines

Choose the kind of component you want to access or create. In-process components are Active-X like components, they are usually smaller than local or remote components and used by bigger application (often through a high level

language). Remote components can act as middleware in a three tier client server architecture. See Location for additional information on possible component locations.

Definition File Dialog

This dialog is used to specify the location of the definition file for the project. An IDL file is usually provided when building a new component since all the sources are available. However, when it comes to accessing an existing component, the sources might not be available. The Type Library is often embedded in the component itself and includes enough information for the wizard to generate the code.

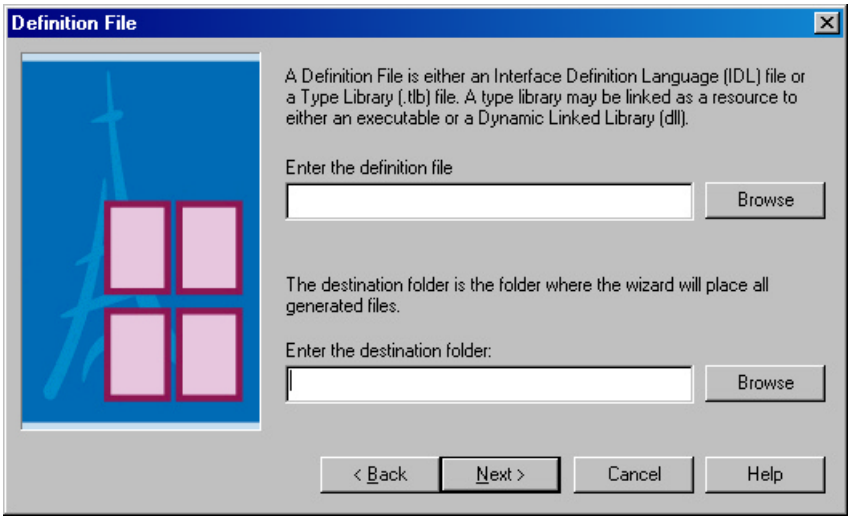

Figure 7: Definition File Dialog

This dialog is also used to enter the destination folder, i.e. the directory where all the files will be generated, preferably empty. If the wizard was to overwrite an existing file it would first back it up and emit a warning message. If any of the entered values are not correct when the *Next* button is pressed the wizard will display a warning message.

IDL Marshalling Definition Dialog

This dialog is displayed only for a server project and if the chosen definition file is an IDL file. It is used to specify how marshalling will work for the component. The first choice that has to be made is whether the component will be accessed through

Automation (using *IDispatch*) or through the interface's virtual table (for additional information on Automation versus virtual table access, please consult COM).

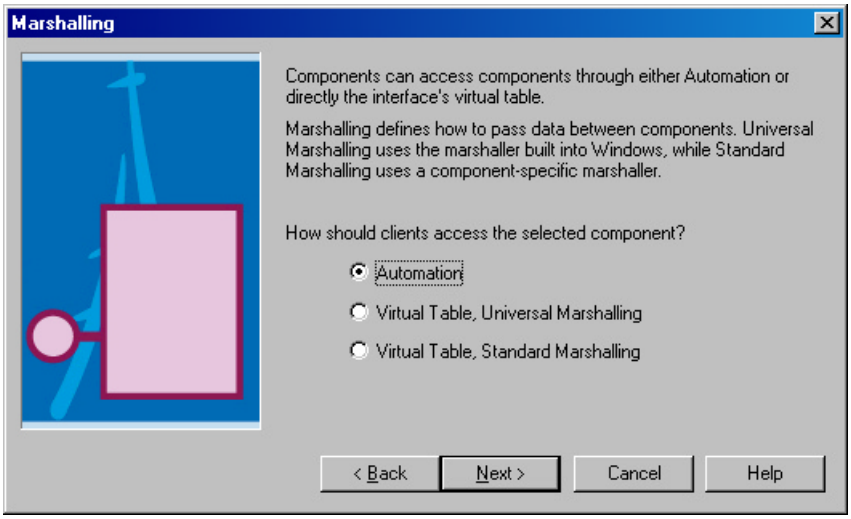

Figure 8: IDL Marshalling Definition Dialog

If Automation is chosen then the Universal marshaller will be used (for additional information on Marshalling please see COM). If Virtual Table access is specified then you have the choice between using the Windows Universal marshaller or the marshaller generated from the definition file. Since this dialog is displayed only when the definition file is an IDL file, choosing Standard Marshalling will force the wizard to compile the marshaller from the code generated with the MIDL compiler. Standard marshalling should be used whenever some interface functions make use of non Automation compatible types (see Automation for a complete listing of these types).

Type Library Marshalling Definition Dialog

This dialog is displayed only for a server project and if the definition file is a Type Library. It includes the same controls as the previous one. You have to choose

between Automation and Virtual Table access and between Universal and Standard marshalling.

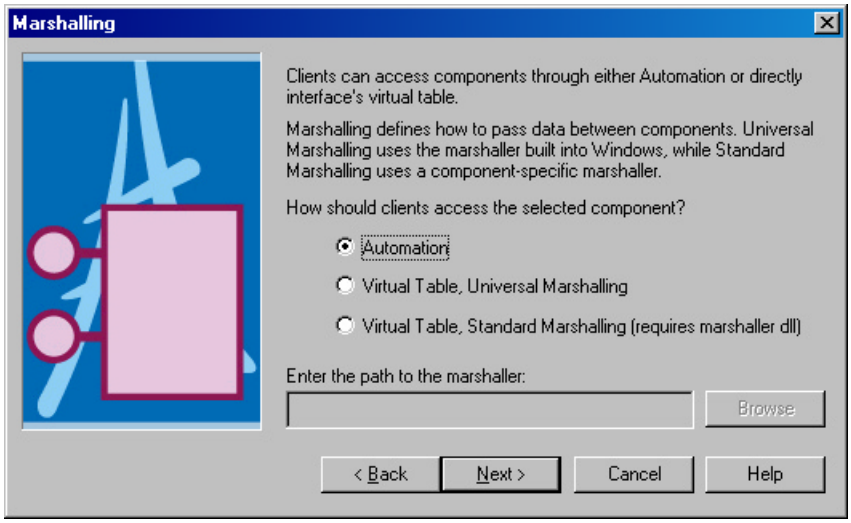

Figure 9: Type Library Marshalling Definition Dialog

Because the definition file is a Type Library, the wizard cannot compile the Standard marshaller by itself. This is the reason for having an extra text field for the path to the marshaller (also known as Proxy/Stub pair or just Proxy/Stub). The Proxy/Stub is a Dynamic Link Library that is used to marshall the data on the wire for a given component (for additional information on Proxy/Stubs, please see Marshalling).

Final Dialog

The last dialog offers a choice of different output levels. By default, the wizard will display errors, warnings and generic information. You can choose not to see warnings or extra informations.

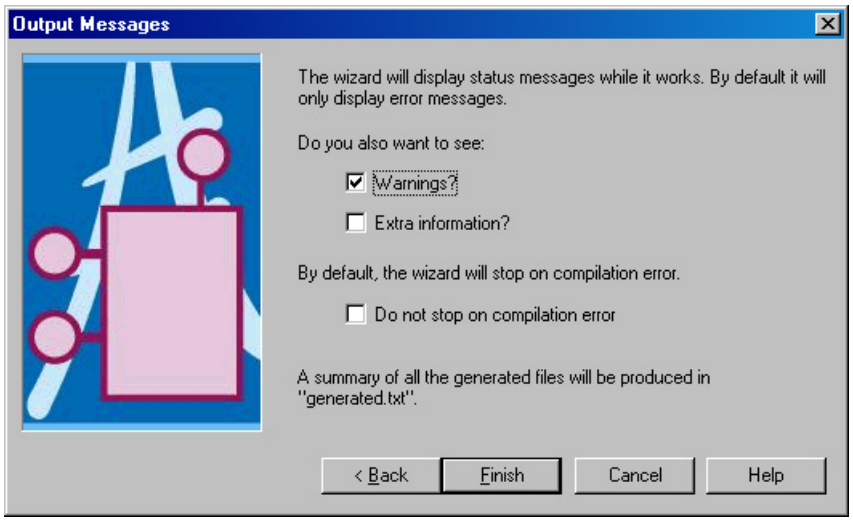

Figure 10: Final Dialog

This dialog also asks whether you wish to continue even though an error occurred while compiling a file.

The *Finish* button will close the dialog and start the processing of the definition file. The project can be saved after the processing is finished.

Definition File Processing

There are six phases involved in the definition file processing:

- IDL Compilation: will occur only if the definition file is an IDL file. The wizard will compile the IDL file into a Type Library and produce the marshaller from the generated C files if Standard Marshalling was chosen.
- Type Library Parsing: The wizard analyze the type library and all its components and builds all the information it needs to generate the code.
- Code Generation: The wizard generates both the Eiffel and C/C++ code from the information gathered during last step.
- C/C++ Compilation: The wizard compiles the C and C++ code generated during last phase into object files and libraries that will be linked with the Eiffel system.
- Eiffel Compilation: The wizard compiles the generated Eiffel code into a precompiled library that can be reused from any project for a client project. In the case of a server project the generated Eiffel code will be compiled into a standard project with the registration class as root class. If the location is In-process then the project corresponds to a DLL whereas if the location of the server is out-of-process then the project corresponds to a standard executable.
- Finally, the wizard will launch EiffelBench and automatically open the generated Eiffel system.

During processing, the name and progress of each phase is displayed.

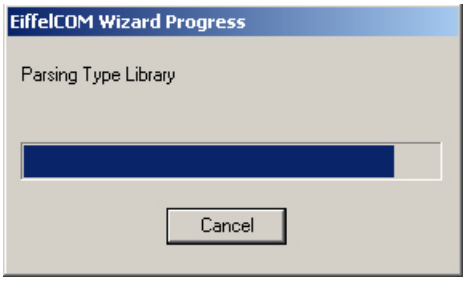

Figure 11: Wizard Progress Dialog

While the wizard processes the definition file it will also display information in real time in the main window if this option was chosen in the Final Dialog. Displayed information includes output of calls to external compilers (C, Eiffel and IDL) and description of the current analyzed or generated Type Library item.

Generated Files

The wizard will generated code in the specified destination folder. The file hierarchy is the following:

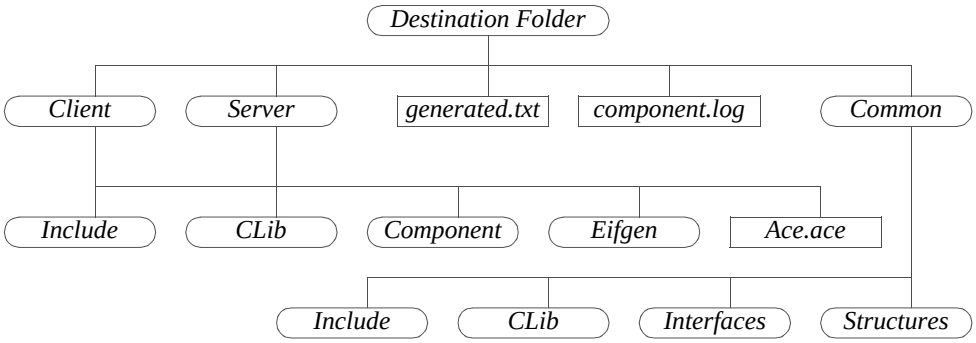

Figure 12: Generated Files Hierarchy

The root folder includes two files and three subclusters.

- The file *generated.txt* includes a list of all the files generated by the wizard.
- The file *component.log* includes a summary of the processing done by the wizard. The name of the file is the name of the definition file appended with *log* (so *Figure 10* presumes that the definition file was e.g. *component.idl*)
- The folders *Client* and *Server* include the files generated respectively for reusing a component or creating a new component. Each includes three subdirectories: *Include* contains all the header files needed to compile the Eiffel code, *CLib* contains the generated C and C++ code as well as the library files. *Component* includes the code that wraps or defines the component. The *Component* subfolder of *Server* will also include the registration class. This Eiffel class includes the code needed to activate the component, its content depends on its location and will differ whether the component is in-process or out-of-process. You will not need to read or edit the C and C++

code included in *CLib* since the wizard will automatically compile it. It is given for information only and can be deleted (you will need to keep the library file though). The *Client* and *Server* folders also include the Ace file used to compile the generated Eiffel code. In the case of a client, the generated code is precompiled whereas in the case of a server it is compiled in a normal system with the registration class being the root class of the system. In the case of a in-process server the Eiffel system is compiled into a DLL whereas in the case of an out-of-process server, it is compiled in a standard executable.

- The *Common* folder includes code that will be used for both the server and the client part. The *Include* and *CLib* directory contain respectively the header files and the C and C++ code. Again the C and C++ sources are not needed and can be deleted, only the library file needs to be kept for the Eiffel system to compile. The *Interfaces* subdirectory include Eiffel classes corresponding to the component interfaces and the *Structures* subdirectory includes Eiffel classes wrapping data structures specified in the definition file.

Class Hierarchy

The generated Eiffel code reflects the architecture of the component described in the definition file. Each interface corresponds to a deferred Eiffel class that includes one deferred feature per interface function. This deferred feature is implemented in the heir of the Eiffel class inheriting from all these interfaces. This central class will be referred to as *Eiffel coclass* in the rest of this document.

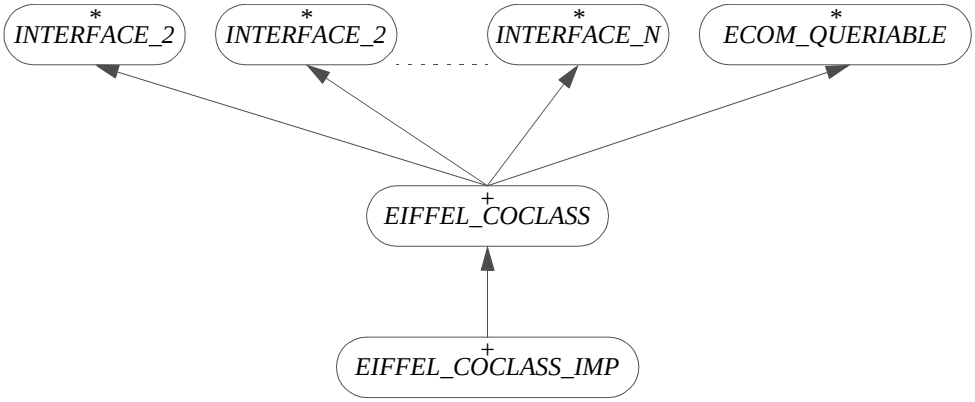

Figure 13: EiffelCOM System Basic Architecture

The Eiffel coclass inherits from the class *ECOM_QUERIABLE* which is part of the EiffelCOM library. This class includes the feature *make_from_other* that can be used to initialize the component from another instance of *ECOM_QUERIABLE*. The *Component* folder also includes Eiffel classes wrapping interfaces that are sent to or received by the component. Such interfaces will be referred to as *implemented*

interfaces in the rest of the document. These classes inherit from both the deferred interface class and *ECOM_QUERIABLE*.

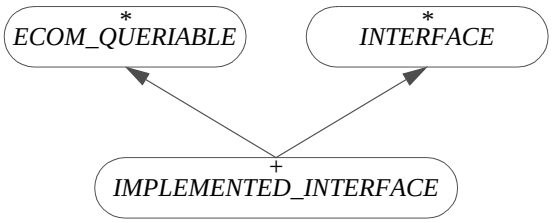

Figure 14: Implemented Interfaces

For both Eiffel coclass and Implemented interfaces, the *INTERFACE* class contains no implementation, it only defines the signatures of the functions that are part of the interface. The actual implementation lies in the heir of that class.

How you should use these generated classes in your system depends on whether you want to access an existing component (client) or build a new component in Eiffel (server).

2.3 ACCESSING A COMPONENT

The wizard will generate all the necessary code to access the existing component. All the plumbing is already done, so instantiating the Eiffel coclass will actually initialize all the necessary COM internals.

Using the Generated Code

To access the component, you need to call features of the coclass. The interface functions signatures data types are either Eiffel types defined in Eiffel data structure libraries (EiffelBase) or wrappers of COM data types specified in the definition file. For example, the following IDL line

```
HRESULT InterfaceFunction ([in] int a, [out, retval] MyStruct * b)
```

will generate the following feature in the Eiffel coclass:

```
interface_function (a: INTEGER): MY_STRUCT
```

where *MY_STRUCT* is a generated Eiffel class wrapping *MyStruct*.

Contracts

The wizard cannot generate fully specified contracts. Indeed, the tool has no domain specific knowledge and can only generate contracts that are domain independent. Such contracts, although useful, are not enough to describe entirely the behavior of the component. Generated contracts include void Eiffel objects as well as C pointer

validity (for wrappers) checking. There might be other conditions to allow calls to an Eiffel coclass feature. Invariants and postconditions can be enforced in a heir of the generated Eiffel coclass. Preconditions, however, cannot be strengthened. A workaround provided by the wizard is to generate a precondition function for each feature in the interface. The default implementation of these functions always return *True*. They should be redefined to implement the correct behavior:

```
interface_function (a: INTEGER): MY_STRUCT is
        -- Example of a generated Eiffel coclass feature
    require
        interface_function_user_precondition:
                interface_function_user_precondition
    do
        ...
    ensure
        non_void_my_struct: Result /= Void
    end
```

So the complete class hierarchy for an Eiffel client coclass is the following:

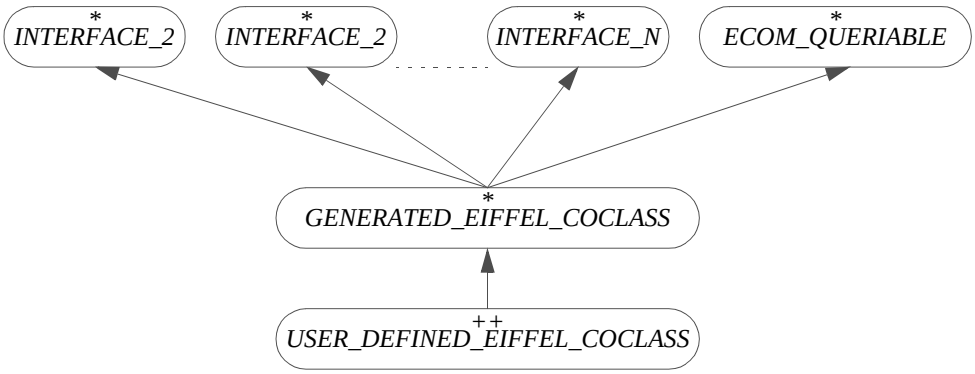

Figure 15: EiffelCOM Client System

Another advantage of the previous hierarchy is that it adds incrementality to the EiffelCOM system. Indeed, should the definition file be modified and the wizard run once more against it, your code would not be changed. Only the generated Eiffel coclass would be, and it would suffice to adapt your heir accordingly.

Exceptions

Another issue is the COM requirement that any interface function should return a status value (known as a *HRESULT*). This leads to side effect features which the Eiffel methodology tends to avoid. The workaround used in EiffelCOM systems is to map these return values into Eiffel exceptions. Should the server the EiffelCOM system is accessing return an error code, the EiffelCOM runtime will raise an Eiffel exception that your code should catch.

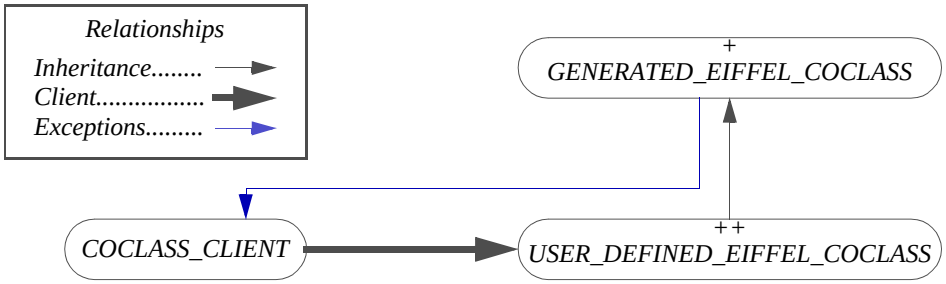

Figure 16: EiffelCOM Client System Exception Raising

As a result, any feature in the coclass client making calls to the user defined Eiffel coclass should include a rescue clause. The processing done in this clause might depend on the nature of the exception. All the standard COM exceptions can be found in the library class *ECOM_EXCEPTION_CODES* which is inherited from by *ECOM_EXCEPTION*. The later also inherits from the kernel class *EXCEPTIONS* and can consequently be used by the coclass client to catch the exceptions.

The following code snippet illustrates how a client can process exceptions raised in the Eiffel coclass:

```
indexing
        description: "Eiffel coclass client example"

class
        COCLASS_CLIENT

inherit
        ECOM_EXCEPTION
                export
                        {NONE} all
                end

create
        make

feature {NONE} -- Initialization

        make is
                        -- Initialize Eiffel coclass.
                do
                        create coclass.make
                end
```

```
feature -- Basic Operations

        coclass_feature_client is
                -- Example of a coclass feature caller
            local
                retried: BOOLEAN
                coclass: EIFFEL_COCLASS_PROXY
            do
                create coclass.make
                if not retried then
                    coclass.coclass_feature -- Actual call
                end
            rescue
                if exception = E_notimpl then
                    -- Process non implemented function error.
                    retried := True
                    retry
                elseif exception = E_invalidarg then
                    -- Process invalid argument error.
                    retried := True
                    retry
                else
                    -- Forward exception to caller.
                end
            end

end -- class COCLASS_CLIENT
```

Summary

There are a few rules to follow when building an Eiffel coclass client but they are straightforward and do not add any constraints. The first rule consist in inheriting the generated Eiffel coclass to implement the preconditions if needed and to ensure better incrementality. The second rule applies to the client: any feature call to the Eiffel coclass should include a rescue clause.

2.4 BUILDING A COMPONENT

Accessing components from Eiffel is only half what the EiffelCOM wizard can do.
The other part is to enable the development of COM components in Eiffel.

Using the Generated Code

The generated Eiffel coclass features are empty features that should be redefined to implement the intended behavior. Unlike client generated code, the server generated code will differ whether you have chosen to implement an in-process or an out-of-process component. The difference lies in the component activation code in the class *ECOM_<Name_of_system>_REGISTRATION*. If the component is in-process then this class includes the four functions that need to be exported from an in-process COM component (*DllRegisterServer*, *DllUnregisterServer*, *DllGetClassObject* and *DllCanUnloadNow*). If the component is out-of-process then the registration class includes call to initialize the component and its graphical user interface (see Component's GUI).

The architecture remains the same as when accessing a component: the generated Eiffel coclass should be inherited from and the contract features redefined. The default implementation for features from the generated Eiffel coclass are empty. They should also be redefined to implement the intended behavior. These features will be called by the EiffelCOM runtime whenever a client access an interface.

> Note: For this first release, the name of the user defined coclass has to be *<Name_of_generated_coclass>_IMP*. So if the generated coclass name is *MY_COCLASS* then the user defined coclass name must be *MY_COCLASS_IMP*.

Component's GUI

In the case of an out-of-process server, you might want to add a Graphical User Interface to your component. There are two different scenarios in which the component can be activated: either its user launched it explicitly (e.g. by double clicking the executable icon) or it was launched by the COM runtime to satisfy a client request. The GUI should appear only in the former case, when the application has been explicitly launched by the user. The generated registration class for an out-of-process server includes the feature:

```
main_window: WEL_FRAME_WINDOW
```

This feature is a once function that can be redefined in a child class to return the class corresponding to the component window. This window is displayed only if the component is not started by COM. When COM loads an out-of-process component, it appends the option "*-embedding*" to the executable. The generated registration class looks for this option and if it is part of the process argument list then it sets the default window appearance to hidden.

As a summary, when building a server you need to implement classes that will inherit from the coclasses and implement the interfaces functions. The names of the

children classes should be the names of the parent classes appended with *_IMP*. You will also have to inherit from the registration class in the case of an out-of-process component to provide the class that implements the component GUI.

Exceptions

The COM standard way of returning error status to the client is by returning an *HRESULT* from the interface function. Such behavior is not acceptable in Eiffel and is replaced with exceptions. In the case of accessing an existing component, exceptions will be raise by the EiffelCOM runtime and caught by your code (see Exceptions for details). While when creating a component it will be your code that will raise exceptions and the EiffelCOM runtime that will catch them. Here is what the code for an Eiffel coclass should look like:

```
indexing
    description: "Eiffel coclass server example"

class
    ECOM_SERVER_COCLASS_IMP

inherit
    ECOM_SERVER_COCLASS -- Generated by the wizard

    ECOM_EXCEPTION
        export
            {NONE} all
        end
```

```
feature -- Basic Operations

    coclass_feature (an_argument: ARGUMENT_TYPE) is
            -- Example of a coclass feature
        do
            if not is_valid (an_argument) then
                trigger (E_invalidargument)
            else
                -- Normal processing
            end
        end

feature {NONE} -- Implementation

    is_valid (an_argument: ARGUMENT_TYPE): BOOLEAN is
            -- Is an_argument a valid argument?
        do
            -- Test of validity of an_argument
        end

end -- class ECOM_SERVER_COCLASS_IMP
```

This class inherits from the generated Eiffel coclass and from *ECOM_EXCEPTION*. It redefines the feature *coclass_feature* from the generated coclass. This feature is part of the interfaces functions that can be called by clients of the component. Its implementation uses the feature *trigger* from *ECOM_EXCEPTION* to raise exceptions in case the feature cannot be executed normally (invalid argument e.g.). This exception will be catch by the EiffelCOM runtime and mapped into an *HRESULT* that will be sent back to the client.

Summary

Implementing an EiffelCOM components consists in inheriting from the generated Eiffel coclasses and implementing their features. The only specific rules to follow relate to the redefinition of precondition features and the use of exceptions to return error status to the client. In the case of an out-of-process server, the registration class should be inherited from and the feature corresponding to the component window redefined to return the correct class.