EiffelCOM
and the
EiffelCOM wizard

Interactive Software Engineering

This manual describes EiffelCOM version 4.5.

Corresponds to release 4.5 of the ISE Eiffel environment, November 1999.

Interactive Software Engineering
ISE Building, 2nd floor
270 Storke Road
Goleta, CA 93117 USA
805-685-1006, fax 805-685-6869@

<*info@eiffel.com*>, *http://eiffel.com*

Contents

1

Getting Started

The following step by step tutorials will help you create your first EiffelCOM projects. An EiffelCOM project can consist either in accessing an existing COM component from Eiffel or in creating a new COM component.

1.1 CREATING A NEW COM COMPONENT

This first tutorial describes the steps involved in creating a COM component in Eiffel. These components can be either DLLs (in-process) or EXEs (out-of-process). This tutorial will focus on creating an in-process component. This component, called *StringManipulator*, will expose one interface *IString* that include the functions *ReplaceSubstring* and *PruneAll* corresponding respectively to the features *replace_substring* and *prune_all* of the EiffelBase *STRING* class. *IString* also includes the property *String* that represents the string being manipulated. This property can be set or read. Other languages will then be able to access these features using this component.

Step by step instructions

- Launch the EiffelCOM Wizard (from the *Start* menu, open *Programs* then *EiffelXX* and click *EiffelCOM Wizard*)
- Choose *Create a new project*
- Click *Next*
- Choose *Server code for a new component*
- Click *Next*
- Click the top *Browse* button and open the file *$EIFFEL4\examples\com\wizard\string_manipulator\string_manipulator.idl* where *$EIFFEL4* represents the path to your EIFFEL45 directory.
- Click the bottom *Browse* button and select the directory where you want to create the project (later referenced as *destination folder*). Choose *$EIFFEL4\examples\com\wizard\string_manipulator\generated* where *$EIFFEL4* represents the path to your EIFFEL45 directory.
- Click *Next*
- Choose *Virtual Table, Standard Marshalling*

- Click *Next*
- Click *Finish*
- Wait until the wizard is done.

First look at the generated code

The first interesting file you might want to look at is the file *generated.txt* in the destination folder. It includes a list of all the files that the wizard generated. When done, the wizard automatically launches EiffelBench with the generated project, you can use EiffelBench to browse the system and get an idea of the class hierarchy. The most interesting classes are the ones in relation with the generated coclass *STRING_MANIPULATOR_COCLASS*. This class inherits from *ISTRING_INTERFACE* corresponding to the IString interface. The coclass is deferred since we are creating a project for a server. It has one descendant *STRING_MANIPULATOR_COCLASS_IMP* that implements the deferred features of the interface. The default implementation returns the error *E_notimpl* to the client. This error means that the function is not implemented on that server.

implementing the component

We want to do something more interesting than just returning an error to our client an are going to edit the implementation of the features in *STRING_MANIPULATOR_COCLASS_IMP. This class will not be overwritten by future executions of the wizard*. There is an implementation of the coclass in *$EIFFEL4\examples\com\wizard\string_manipulator\server*. Just copy this file over to *$EIFFEL4\examples\com\wizard\string_manipulator\generated\server\component* and quick melt your project. Voila!, you have your first EiffelCOM component up and running.

Tips

- If you are to develop an EiffelCOM component, you will most probably need to set the registry entry *MELT_PATH*. This value needs to be setup in the case the system is not launched from the directory containing the byte code (by default *EIFGEN\W_code*). The EiffelCOM component client will launch the server from its current location and will cause the server to crash if the *MELT_PATH* key is not setup properly. To put this value in your registry, create a key under *HKEY_CURRENT_USER\SOFTWARE\ISE\Eiffel45* with the same name as the name of your project (e.g. *string_manipulator*) and add the string value named *MELT_PATH* containing the path to the byte code (*.melted* file). You will find an example of such entry in the file *melted_path.reg* in *$EIFFEL4\wizard\config*. You can edit this file as needed (replace *string_manipulator* with the name of your system and change the path to point on the correct location using '\\' as directory separator) and double click it to enter the information in your registry.

- To test your component, you need first to register it. If the component is in-process then it is registered via the *regsvr32* utility using the following syntax: *regsvr32 system_name.dll* where *system_name* is the name of the dll (e.g. *string_manipulator*). If the component is out-of-process then it is register using the following syntax: *system_name.exe -Regserver* where *system_name* is the name of the component executable file.

For the purpose of this tutorial you will need to register the component with the following command (run from a dos console): *regsvr32 string_manipulator.dll* and to enter the information in *$EIFFEL4\wizard\config\melted_path.reg*. You will have to change the content of this file if you did not install *Eiffel* under *C:*.

1.2 ACCESSING A COMPONENT

Now that we have build a component, we are going to reuse it from another Eiffel project. The same exact steps should be followed for components written in any language.

Step by step instructions

- Launch the wizard
- Choose *Create a new project* and click *Next*
- Click *Next* again
- Click the top *Browse* button and open the file *$EIFFEL4\examples\com\wizard\string_manipulator\string_manipulator.idl* where *$EIFFEL4* represents the path to your EIFFEL45 directory. You could also choose to open the type library (*.tlb*) that was generated by the wizard when CREATING A NEW COM COMPONENT in the destination folder of that first project.
- Click the bottom *Browse* button and select the directory where you want to create the project (later referenced as *destination folder*). Choose *$EIFFEL4\examples\com\wizard\string_manipulator\generated* where *$EIFFEL4* represents the path to your EIFFEL45 directory.
- Click *Next*
- Click *Finish*
- Wait until the wizard is done.

First look at the generated code

In the case of a client the wizard will precompile the generated code and open EiffelBench on the precompilation project. *Note: a precompilation project is read-only, you will need to start another EiffelBench to reuse the generated classes.* The interesting classes are all related to the coclass proxy *STRING_MANIPULATOR_PROXY*. The proxy is the Eiffel class that gives access to the component. Each feature on the proxy calls the corresponding interface function

on the component. You can use the EiffelBench opened by the wizard to browse through the generated classes and get an idea of the class hierarchy.

Implementing a client

We are now going to implement a client of the *StringManipulator* component. Open a new EiffelBench that will be used to create the client project. Create the project in *$EIFFEL4\examples\com\wizard\string_manipulator\client* using the ace file found in that directory. Freeze and run the project. You are now accessing the previously built component and calling functions on its interfaces!. The interesting class is *MY_STRING_MANIPULATOR* which inherits from the generated *STRING_MANIPULATOR_PROXY* and redefine the feature *replace_substring_user_precondition*. The generated interfaces will include contracts for each exposed function. You can redefine the *user_precondition* features to implement your own preconditions.

Contracts

Contracts can be broken directly on the proxy in which case you will get a standard contract violation or in the server. If contracts are broken on the server then the exception will be forwarded by the EiffelCOM runtime to the client. The feature *replace_substring_user_precondition* in *MY_STRING_MANIPULATOR* includes the following commented line:

```
-- Result := True
```

Un-comment this line by removing the preceding '--' and comment out the rest of the feature. Now the contract of the *replace_substring* feature is wrong and erroneous calls can be made. Quick melt the changes and run the client. Enter some invalid numbers in the fields used to call this feature. After you click *Replace* you will see an error message box warning you that a precondition was violated on the server side. This is how you can use contracts 'over the wire'. The preconditions was violated in the server, this exception was caught by the EiffelCOM runtime and sent back to the client.

Summary

You now have the basic knowledge needed to run the EiffelCOM wizard and produce or access COM components. The benefits of using the wizard are numerous including a fast and easy generation of "plumbing" code as well as enhanced debugging capabilities through the use of contracts. The wizard is also very generic, it can generate or wrap any kind of components. We hope you enjoy using it as much as we enjoyed developing it,

The EiffelCOM team.

2

The Component Object Model

2.1 OVERVIEW

The goal of this chapter is to cover enough information on COM so that Eiffel developers can use the EiffelCOM wizard in an effective way. It will not cover all of COM since it would require an entire book but will present the main concepts needed to understand how to build an EiffelCOM system.

Briefly said, the Component Object Model is a Microsoft binary standard that establishes how two binary units can access each other at runtime. Such binary units can run in the same process, in different processes on the same machine or even on different machines. Components can be implemented in any language as long as the compiler produces COM standard compliant binaries.

The advantages of such an approach include an increased reusability (binary reuse), a better version management (COM standard implies that new component versions will still be compatible with older ones) and a built-in runtime environment. The binary reuse aspect of COM added with the source reuse ability of Eiffel offer the developer an ideal platform to increase considerably their productivity.

2.2 GENERALITIES

This paragraph will present briefly the principal notions of COM needed to understand the rest of this chapter. COM is a binary standard that describes how the component can communicate with the outer-world. Communication is done through well defined *interfaces*. Each interface is a *specification* of a group of methods and functions. An interface does not contain the implementation of these routines and functions but only their specification (signatures). The actual implementation lies in the *coclass*. There can be different implementations of a same interface in different coclasses. Finally each coclass can be instantiate using a *class object* or *class factory.* These three notions will be discussed further in the forthcoming paragraphs.

Interfaces

They are at the heart of any COM component. Interfaces are described in the definition file of a component. They consist of a group of semantically related

functions that can be accessed by the clients of the component. Although they are a specification they also have a physical representation. A client can request a pointer on an interface and will access the component functions through that pointer. Interfaces are the only possible way to access functions from a component. They enforce information hiding by providing only the public functions to the client.

Interfaces also define the *type* of a component. Each interface corresponds to a specific *view* of the component. It can be compared to polymorphism in the Object Oriented world. Whenever an interface from a component is requested, only the functions defined on that interface are accessible as if the component was polymorphically cast into an object of the type of that interface.

The COM specification requires that any interface provides access to all interfaces on the same component. All the interface should include a specific function called *QueryInterface* that will provide a pointer on any other interface of the component. Interfaces are identified with a globally unique identifier (GUID) guaranteed to be unique in time and space. Since this function has to be on every interface, it has been abstracted into a specific interface called *IUnknown* which all other interfaces must inherit from. The two other functions exposed by *IUnknown* are *AddRef* and *Release*. These functions should be called respectively when a client gets a reference on an interface or when it discards that reference. These two functions define the lifetime of the component: each interface keeps track of clients keeping a reference on them and when no clients have references anymore, the component *can* be unloaded from memory. You might start to worry thinking that this business of reference counting will imply lots of headaches, memory leaks, etc... and you would be right should you choose a low-level language to implement your components. Fortunately, you will never have to implement or use these functions in Eiffel: all the processing related to *IUnknown* is provided by the EiffelCOM runtime. Calls to *QueryInterface* are done “behind the scene” and only when needed. The lifetime of the component is also taken care of by the EiffelCOM runtime.

Coclass

We have seen that interfaces can be perceived as views of a component. This conceptual representation actually maps the implementation of an EiffelCOM component since the coclass inherits from the interfaces and implements their deferred features. Indeed, interfaces are deferred classes with all features accessible from outside deferred. The coclass is an Eiffel class that inherits from these interfaces and implements all the features. This design is not specific to Eiffel though and can be found in other languages as well. The coclass defines the behavior of the interfaces functions.

Class Object

We have seen that interfaces are accessed through interface pointers. But how does a client get hold on one of these?

The answer lies in the class object. The name of this module should really be coclass factory since its goal is to spawn instances of the coclass on request. Class objects are accessed by COM whenever a client request a new instance of the associated component. COM loads the class object and asks it to provide the interface pointer requested by the client.

The way a class object is loaded in memory (this process is called *activation*) depends on the location of the component (See Location for a description of the possible locations of a component). If the component is an in-process server then the class object is called directly through the functions exported from the DLL. If the component is an out-of-process server then it provides COM with a pointer to the class object. In both cases, once the component is loaded, COM has access to the class object and can call it would a client request a new instance of a component.

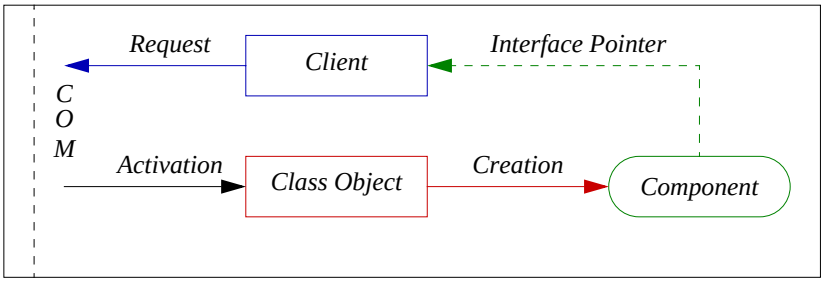

Figure 1: Component Creation

The code for the class object is generated by the EiffelCOM wizard so that Eiffel programmers will not have to worry about it.

2.3 TYPES OF COMPONENTS

ActiveX, DirectX, OCX, COM+, ADO+, ASP etc.... who never heard of these technologies? They all represent yet another use of the COM standard. This paragraph will focus on categorizing COM components according to their own properties as well as the context in which they are used. The categorization will define how the EiffelCOM wizard should be used to wrap or create a component.

Location

The first criteria that defines the type of component is from where it will be accessed: will the component be loaded in the client process or will the component be a remote server for a distributed application? In the former case, the component is compiled as a Dynamic Link Libraries (DLL) while in the latter case it is a standard executable.

In-process Components

Typical instances of DLL components are found in technologies such as OCX, ActiveX or ASP. These are small, downloadable binaries that will be loaded and

executed in a *container*. The container acts as a client of the component. The EiffelCOM wizard provides the ability to wrap such components by providing access to its interface to any Eiffel container. It is also possible to create a new In-process component in Eiffel. One main difference with out-of-process component (other than the nature of the module, DLL versus executable) is the way in-process components are activated. In the case of out-of-process components, the component will specify COM when it is ready to receive calls from client. In the case of an in-process server the call is coming directly from COM: COM first loads the DLL into the client process and then calls the exported function *DllGetClassObject* to access the component class object. The other three exported functions of an in-process component are *DllRegister* to register the component in the Windows registry, *DllUnregister* to unregister the component from the registry and finally *DllCanUnloadNow* which gets called by COM whenever it tries to unload the component from memory. These four functions must be accessible from outside the DLL for the in-process component to work properly.

Out-of-process Components

These components are standard executable acting as servers that can be accessed locally or over a network. Typically used in a three tier client server architecture, the major difference with in-process servers (other than the module type - executable instead of DLL) is their lifetime. In-process components are typically loaded to achieve a specific task and unloaded just after while out-of-process components are servers supposed to run continuously. The EiffelCOM wizard allows to build clients for such servers in Eiffel. It also provides the ability to create such servers.

Access Type

Regardless of its location, a COM components can be accessed either directly through its interfaces or by using Automation.

Automation

Automation consists in using a well known interface to provide access to a group of *methods* and *properties*. This interface called *IDispatch* includes the method *invoke* that allows to call a method, set or get a property on the Automation server. One advantage of that approach is that the interface is a *standard interface* whose functions and methods are specified. As a result, Windows can include a built-in marshaller for that interface (See Marshalling for information on what a marshaller is). The supported types (known as Automation types) and their Eiffel equivalents are listed in the following table:

COM Type	*Eiffel equivalent*	*Description*
boolean	*BOOLEAN*	Standard boolean
unsigned char	*CHARACTER*	Standard character

double	*DOUBLE*	Standard double
float	*REAL*	2 bytes real
int	*INTEGER*	Standard integer
long	*INTEGER*	Standard integer
short	*INTEGER*	2 bytes integer
BSTR	*STRING*	Standard string
CURRENCY	*ECOM_CURRENCY*	Currency value
DATE	*DATE*	Standard date
SCODE	*INTEGER*	Return status
Interface IDispatch *	*ECOM_QUERIABLE*	Automation interface
Interface IUnknown *	*ECOM_QUERIABLE*	Generic interface
dispinterface	*ECOM_QUERIABLE*	Automation interface
Coclass *Typename*	*TYPE_NAME*	Component main class
SAFEARRAY(*TypeName*)	*ARRAY [TypeName]*	Array
TypeName*	*CELL [TypeName]*	Pointer to type
Decimal	*ECOM_DECIMAL*	Decimal value

The other advantage is a more dynamic discovery of the methods and properties of a component at runtime. Indeed the *IDispatch* interface also includes methods to check whether a method or property is available and, in that case, get its identifier. This process is called *late binding* and allows component to discover at runtime what are other components functionality.

This approach has also a lot of drawbacks: firstly, *late binding* is not an efficient way of calling a function on an interface since its identifier must first be requested and then the function called. That's two round trips which can be expensive in a distributed environment. Secondly, since the marshaller is built-in, it has to know in advance all the possible types that a function can accept to be able to marshall the corresponding data. There are consequently a limitation on the number of types that one can use in signatures of functions on an Automation compatible interface. The set of available types is called *Variant* and cover most of the standard types. It does not allow however the passing of complex user defined data types. For these reasons Automation is mostly used in scripting environments (where speed is not an important factor) to accomplish simple tasks.

Direct Access

Direct interface access is the preferred way to access remote servers where speed becomes a concern and data types are specific to the application. The first interface

pointer on the component is obtained through the class object (see Class Object). Other interfaces on the component are obtained by calling the *QueryInterface* function.

As information on any interface cannot be accessed dynamically, the description of the interfaces must be provided to tools that need to handle the components such as the EiffelCOM wizard. The official way of describing components and interfaces is through IDL. Once an IDL file has been written to describe a component it can be compiled with *MIDL* to generate both a type library and the code for the marshaller specific to that interface.

EiffelCOM

The idea in EiffelCOM is that the way a component is accessed is implementation detail that the user should not have to deal with. Of course he should be able to choose what kind of access he wants to use but this choice should have no impact on the design of the Eiffel system itself. For that reason, the Eiffel code generated by the wizard follows the same architecture independently of the choice made for interface access and marshalling. The difference lies in the runtime where the actual calls to the components are implemented.

2.4 DEEPER INTO COM

The next paragraph gives a bit more details on the internals of COM. The understanding of these details are not required to use the EiffelCOM wizard but might help making better decisions when designing new EiffelCOM components.

Apartments

The first interesting subject that requires more in-depth cover is the execution context of a component. Components can be run in the same process as the client but can also run in a separate process even on a different machine.

This superficial description only take into accounts processes. What happens if a component uses multithreading to achieve it tasks? In a case of a remote server, this scenario does not seem too esoteric. The problem is that a server does not (and should not) know in advance what its clients will be. It cannot assume that the client will be able to take advantage of its multithreading capabilities. Conversely a multithreaded client should not rely on the server ability to handle concurrent access.

The solution chosen in the COM specification is to define an additional execution context called an *apartment*. When COM loads a component it creates the apartment in which the component will run. Multiple instances of a multithreaded component will leave together in the same apartment since asynchronous calls will be handled correctly and there is no need to add any synchronization layer. On the other hand, singlethreaded component will be alone in their apartment and any concurrent calls coming from clients will be first synchronized before entering the apartment. These

two behaviors define two different kinds of apartments: *Multi Threaded Apartments* (MTA) and *Single Threaded Apartments* (STA).

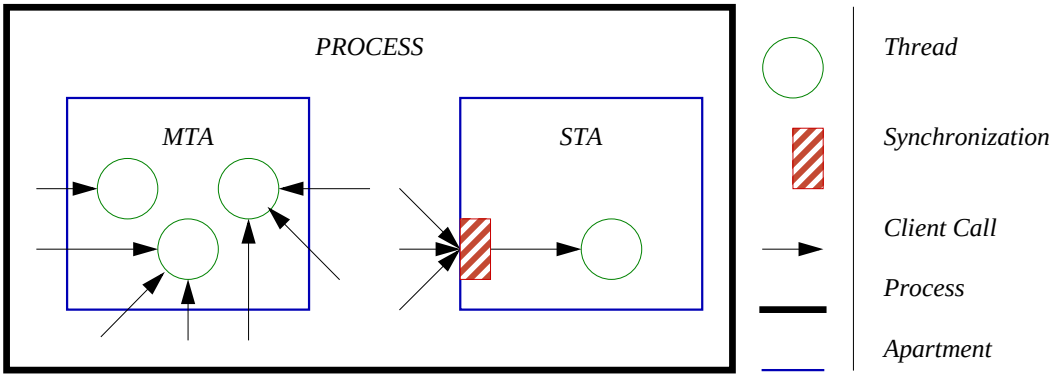

Figure 2: Apartments

Apartments solve the problem of concurrency by removing the necessity of knowing the multithreaded capability of a component and its clients. Multithreaded clients can always make asynchronous calls and depending on whether the component handles concurrent access or not, they will be forwarded or first synchronized. There can be multiple instances of STA running in one process while there will be at most one MTA.

Marshalling

At this point you might wonder how calls can “cross” the apartments boundaries. Components from a STA can make calls to components running in a MTA and vice versa. These apartments might be running in different processes or even on different machines. The approach chose in the COM specification is using the *proxy* and *stub* patterns.

The idea is to trick the client of an interface by providing an interface proxy in its apartment. The proxy include exactly the same function as the interface itself but their implementation will just forward the call to the actual interface. The client has no idea whether the entity it is dealing with is the actual interface or just a proxy. One of the main interest of that approach is that the client implementation is independent from the location of the component.

Last explanation is not totally accurate: the call will not be forwarded to the actual interface but to its stub. The stub is the counterpart of the proxy, it represents the client for the interface. The interface doesn’t know either whether it is communicating with the actual client or a stub. Although it is not totally true that the component implementation is independent from the location of the client, the stub pattern still helps keeping code identical for the implementation of the interface themselves. The implementation of a component will still be different whether it is

an in-process or out-of-process component since it will have to be a DLL in one case and a executable in the other. The design of the interfaces might also differ since out-of-process servers will tend to avoid too many round trips.

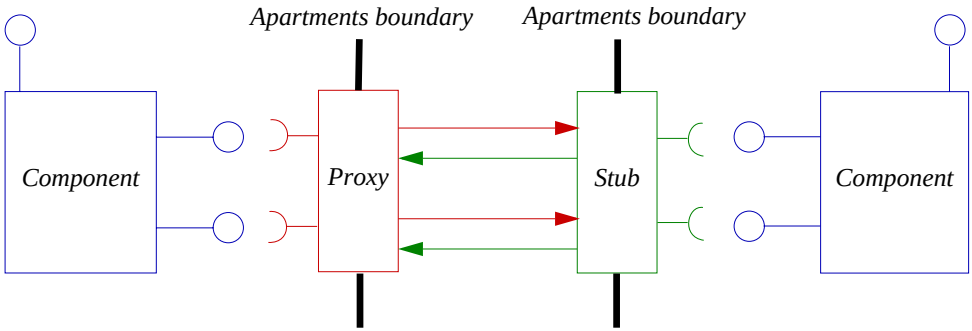

Figure 3: Cross Apartment Calls

There is one proxy/stub pair per interface. The proxy or the stub is loaded dynamically only when needed. This proxy/stub pair constitute the *marshaller*. The reason for having a single name for two different things come from how MIDL generates its code. MIDL will produce files for one DLL in which both the proxy and the stub will be included. This DLL is the marshaller.

Summary

This very brief introduction to the Component Object Model should be enough to get started with the EiffelCOM wizard. It specifies the main characteristics that define the type of a component and that need to be given to the wizard along with the definition file.

3

The EiffelCOM Wizard

3.1 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.

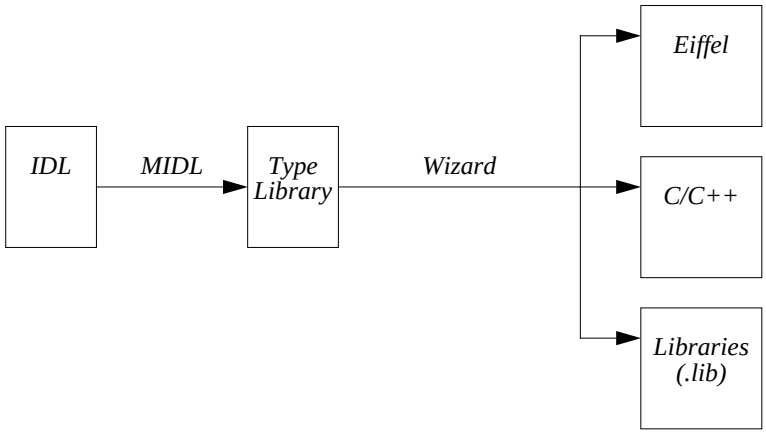

Figure 4: Code Generation Process

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).

3.2 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:

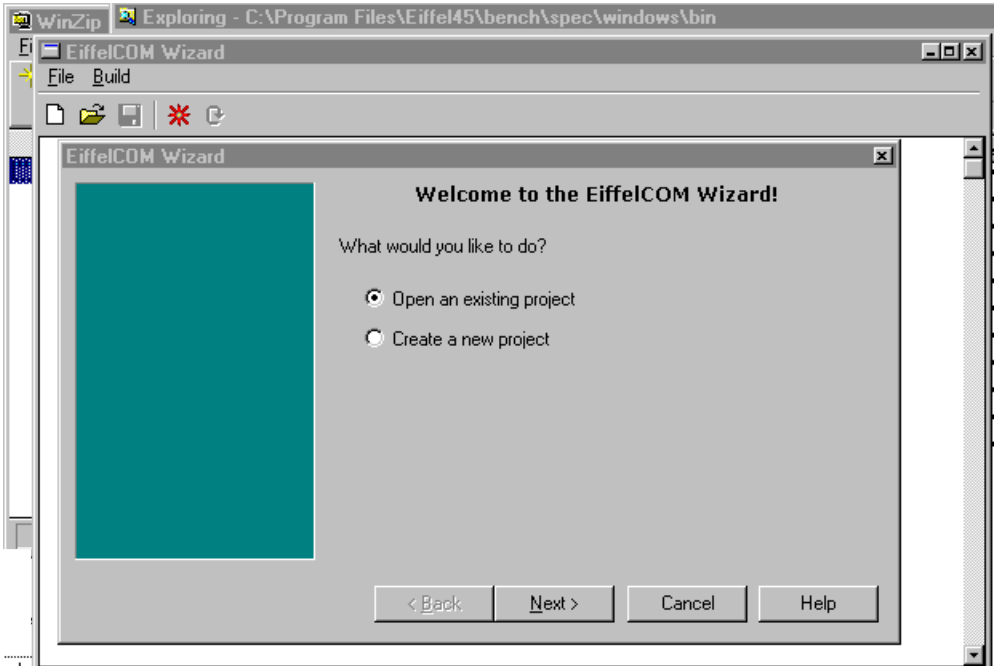

Figure 5: EiffelCOM Wizard Main Window

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.

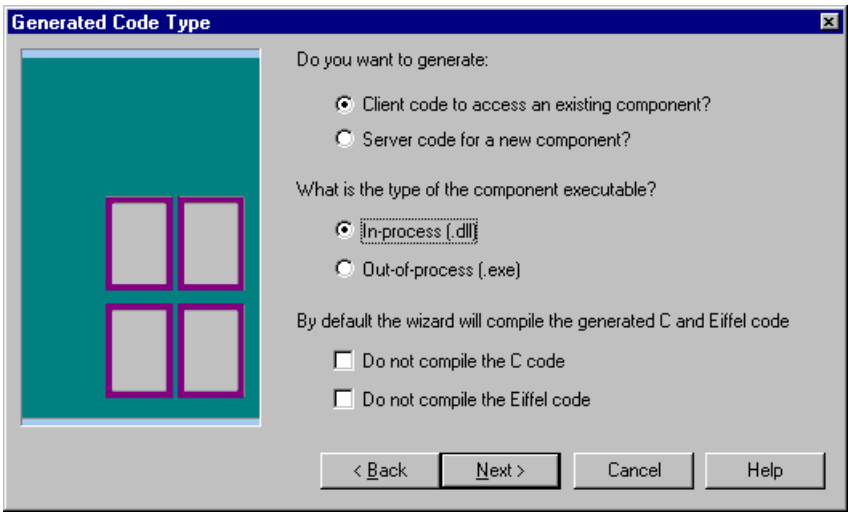

Figure 6: Introduction Dialog

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.

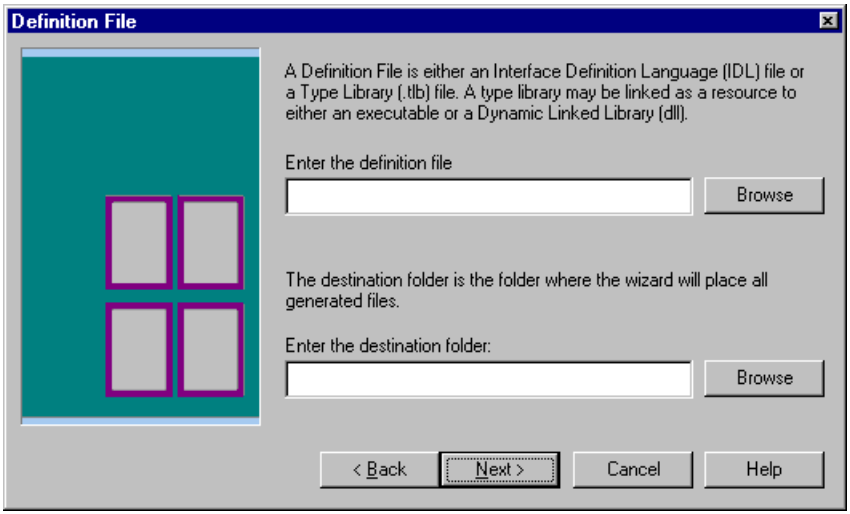

Figure 7: Definition File Dialog

This dialog also serves to enter the destination folder, i.e. the directory where all the files will be generated, preferably empty. If the wizard was used 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 serves 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).

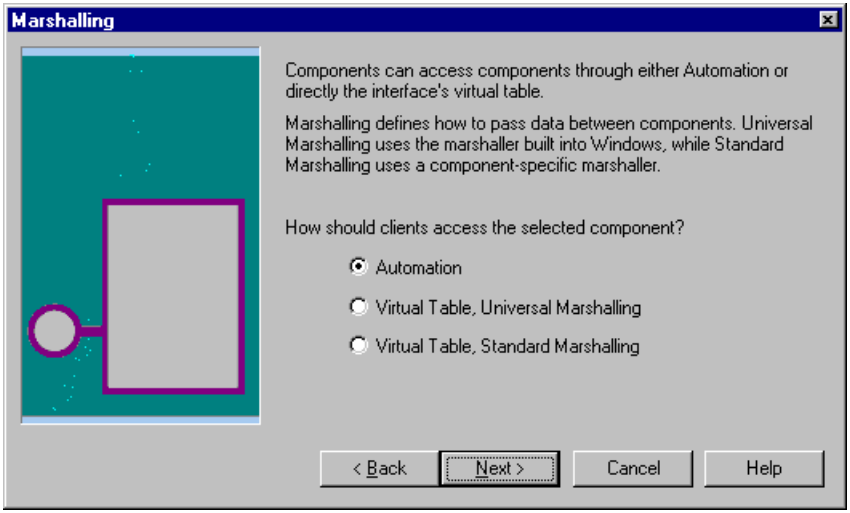

Figure 8: IDL Marshalling Definition Dialog

If you choose Automation then the Universal marshaller will be used (for additional information on Marshalling please see COM documentation). 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.

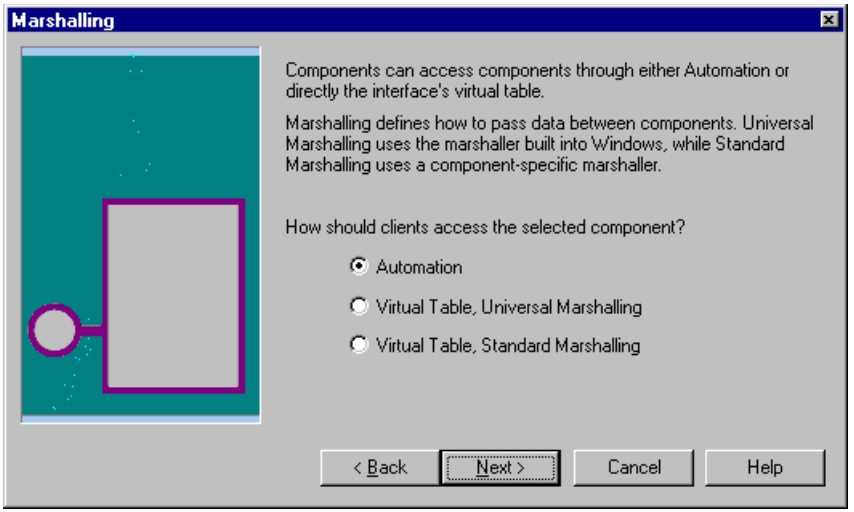

Figure 9: Type Library Marshalling Definition Dialog

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 information.

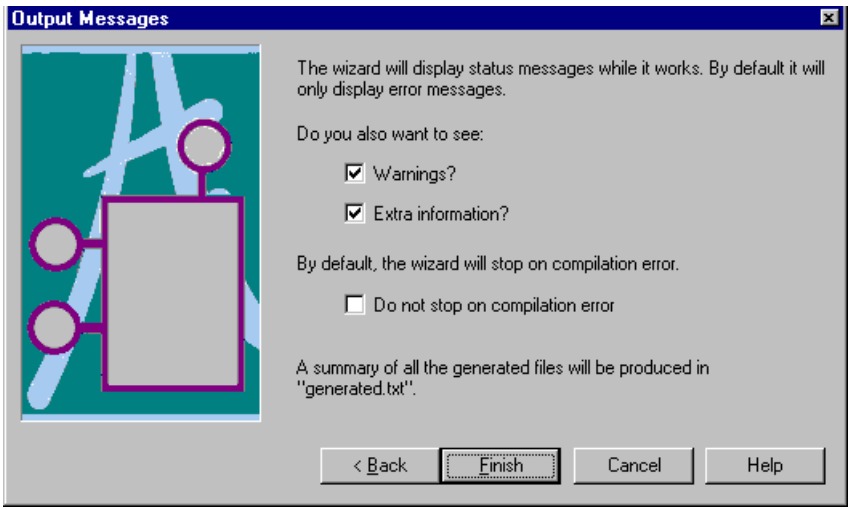

Figure 10: Final Dialog

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.

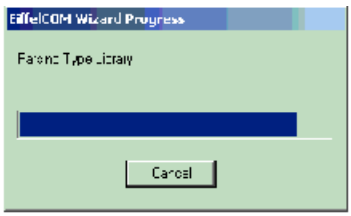

Figure 11: Wizard Progress Dialog

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:

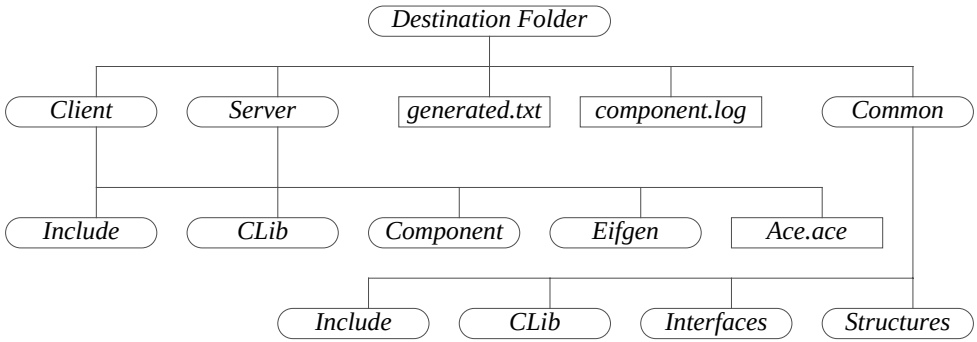

Figure 12: Generated Files Hierarchy

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.

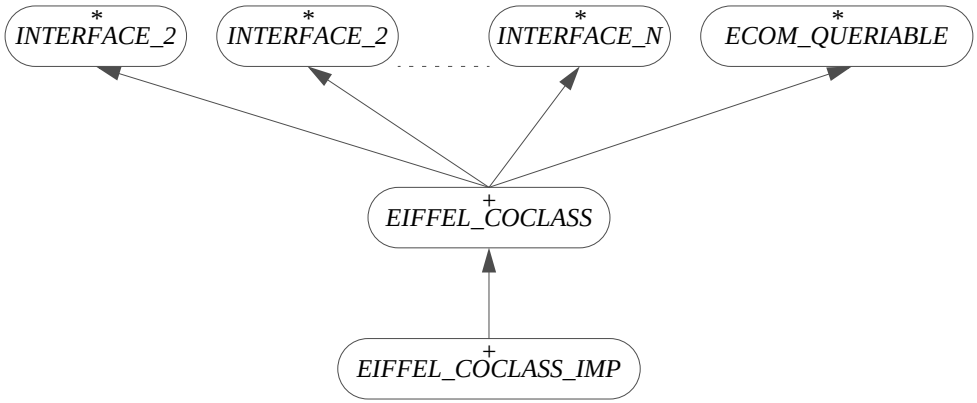

Figure 13: EiffelCOM System Basic Architecture

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*.

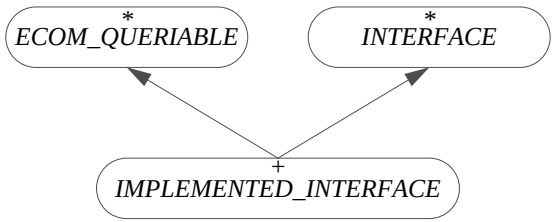

Figure 14: Implemented Interfaces

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).

3.3 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:

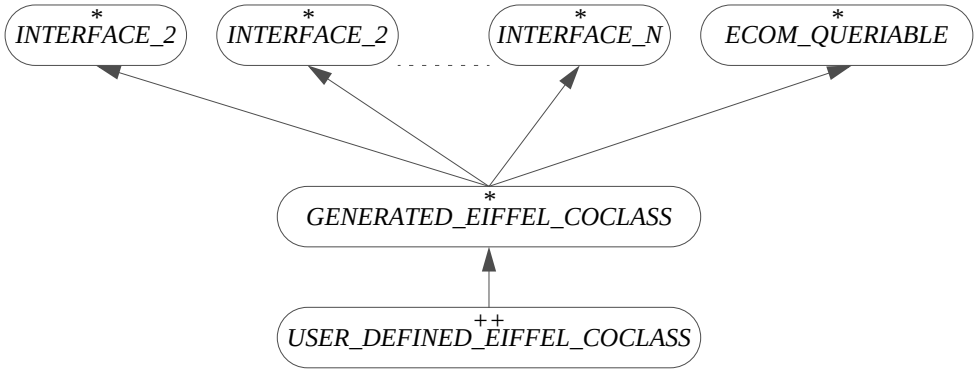

Figure 15: EiffelCOM Client System

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.

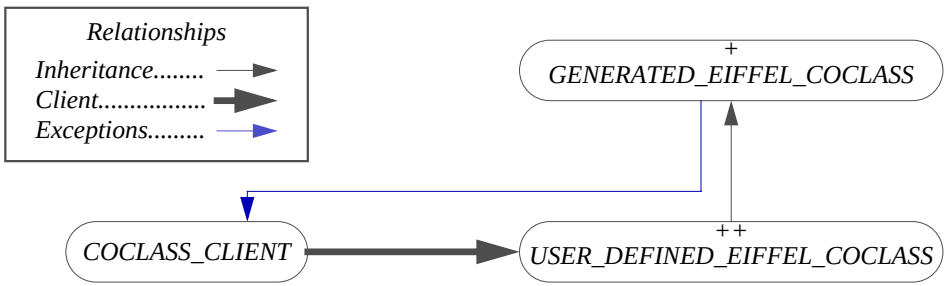

Figure 16: EiffelCOM Client System Exception Raising

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.

3.4 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.

4

The EiffelCOM Library

The EiffelCOM library adds compound files support to your Eiffel applications. Compound files are structured files that can embed different types of data in a single file. It is the ideal format to store documents including different types of information. It is the format used by Microsoft® Office applications. This chapter will not cover the compound files architecture itself but will focus on its support in EiffelCOM. It requires that the user be familiar with compound files and the compound files API.

The EiffelCOM library enables the creation and use of the **IRootStorage**, **IStorage** and **IStream** interfaces and includes a wrapping of all necessary structures for handling compound files. It also includes classes that encapsulates all necessary flags and constants.

4.1 COMPOUND FILES

The cluster **ecom_storage** includes the files *ecom_root_storage.e*, *ecom_storage.e* and *ecom_stream.e*. These files correspond to the **IRootStorage, IStorage** and **IStream** interfaces, respectively.

Storages

Storages are to compound files what directories are to a standard file system. They can include nested storages and/or streams. Streams are the equivalent of files in a standard file system. They include the data itself. The following features are available on *ECOM_STORAGE* objects:

- *destroy_element* (*element_name*: **STRING**) — removes the stream or the substorage *element_name* of **Current**. You can cancel this action using a call to *revert* (described later). Thus, *commit* (described later) will be called to confirm the deletion.
- *copy_to* (*stg_dest*: *ECOM_STORAGE*) — copies **Current** into *stg_dest*.
- *move_element_to* (*element_name*: *STRING*; *stg_dest*: *ECOM_STORAGE*; *new_element_name*: *STRING*; *movmode*: *INTEGER*) — moves the stream or substorage element_name to *stg_dest* and renames *element_name* as *new_element_name*. *movmode* can take one of the values described in **ECOM_STGMOVE**.

- *rename_element* (*old_element_name*, *new_element_name*: *STRING*) — renames the stream or substorage *old_elment_name* as *new_element_name*.
- *set_class* (*clsid*: *STRING*) — set the class identifier of **Current** using the *clsid* value.
- *commit* (*mode*: *INTEGER*) — commits all changes made in **Current**. The commits all streams and substorages included in **Current**. *mode* can take one of the values described in **ECOM_STGC**.
- *revert* — discards all changes made to **Current**.
- *is_compound_file* (*filename*: *STRING*): *BOOLEAN* — checks if the system file *filename* is a compound file (**Result** = **True**) or not (**Result** = **False**).
- *enum_elements*: *ARRAY* [*ECOM_STATSTG*] — enumerates the **ECOM_STATSTG** structures associated with the streams and substorages included in **Current**.

Streams

Streams are where the data is actually stored. Different streams can store different types of data. The following features are available on *ECOM_STREAM* objects:

- *seek* (*offset*, *origin*: *INTEGER*) — sets the position of the seek pointer by adding *offset* to *origin*. *origin* can contain one of the values described in **ECOM_STREAM_SEEK**.
- *set_size* (*new_size*: *INTEGER*) — sets the size of the stream to *new_size*. If the *new_size* value is larger than the current size of the stream, then the new bytes fill with undefined values. If the *new_size* value is smaller than the current size, the stream truncates.
- *copy_to* (*stream_dest*: *ECOM_STREAM*; *num_bytes_to_copy*: *INTEGER*) — copies *num_bytes_to_copy* to *stream_dest*.
- *commit* (*mode*: *INTEGER*) — commits all changes made to the stream. *mode* can take one of the values described in **ECOM_STGC**. Compound file implementation does not support opening streams in transacted mode, so this method primarily flushes memory buffers.
- *revert* — discards all changes made to the stream. Compound file implementation does not support opening streams in transacted mode, so this method has no effect.
- *stat* (*stat_flag*: *INTEGER*) — fills an **ECOM_STATSTG** structure that corresponds to current stream.
- *read* (*buff*: *POINTER*; *num_bytes_to_read*: *INTEGER*): *INTEGER* — reads *num_bytes_to_read* bytes from the stream into *buff*. Returns the actual number of bytes read.
- *write* (*buff*: *POINTER*; *num_bytes_to_write*: *INTEGER*): *INTEGER* — writes the *num_bytes_to_write* bytes of buffer pointed by *buff* to stream, and the returns the number of bytes actually written.

- *lock_region* (*offset*, *count*: *ECOM_LARGE_INTEGER*; *lock*: *INTEGER*) — restricts access to the range of bytes defined by *offset* and *count*. *lock* can take one of the values described in **ECOM_LOCKTYPES**.
- *unlock_region* (*offset*, *count*: *ECOM_LARGE_INTEGER*; *lock*: *INTEGER*) — removes the access restriction previously set using *lock_region*. The arguments are the same as for *lock_region*.

The last two features may of may not be available, depending on the COM interface that the Eiffel class wraps. If the interface does not support region locking, then the status code is set to *Stg_e_invalidfunction*. The standard Windows implementation of compound file does not support region locking.

All of the previous features directly encapsulate the COM **IStream** interface. Thus, read/write is not very effective, since it requires a pointer on a buffer, which is not directly available in Eiffel. As a result, the **ECOM_STREAM** class offers the following read/write features:

- *read_string* — reads a string at the current seek position in the stream and sets *last_string* accordingly.
- *read_character*— reads a character at the current seek position in the stream and sets *last_character* accordingly.
- *read_integer*— reads an integer at the current seek position in the stream and sets *last_integer* accordingly.
- *read_real*— reads a real at the current seek position in the stream and sets *last_real* accordingly.
- *read_boolean*— reads a Boolean at the current seek position in the stream and sets *last_boolean* accordingly.
- *write_string* (*s*: *STRING*) — writes *s* at the current seek position in the stream.
- *write_character* (*c*: *CHARACTER*) — writes *c* at the current seek position in the stream.
- *write_integer* (*i*: *INTEGER*) — writes *i* an integer at the current seek position in the stream.
- *write_real* (*r*: *REAL*) — writes *r* at the current seek position in the stream.
- *write_boolean* (*b*: *BOOLEAN*) — writes *b* at the current seek position in the stream.
- *end_of_stream*: *BOOLEAN* — **True** when the end of the stream is reached.

The attribute *end_of_stream* automatically updates when there is a call to any of the read/write features.

Other classes

The EiffelCOM compound file interfaces also need several other classes. These classes handle the **ECOM_STATSTG** structure included in the *ecom_structure* cluster and require some of the constants defined in the *ecom_flags* cluster.

The **ECOM_STATSTG** class is a direct encapsulation of the **STATSTG** structure. The following attributes can be set/accessed

- *element_name*: *STRING* — name of element (storage or stream).
- *element_type*: *INTEGER* — type of element (storage, stream). **Result** can take one of the values described in **ECOM_STGTY**.
- *element_size*: *INTEGER* — size of element in bytes (stream).
- *modification_time*: *WEL_FILE_TIME* — last modification time.
- *creation_time*: *WEL_FILE_TIME* — creation time.
- *access_time*: *WEL_FILE_TIME* — last access time.
- *open_mode*: *INTEGER* — mode in which element was opened. **Result** can take one of the values described in **ECOM_STGM**.
- *locks_supported*: *INTEGER* — A group of flags relevant only for stream. For each lock item, indicates whether or not a call to *lock_region* is worthwhile. **Result** can take one of the values described in **ECOM_LOCK_TYPE**.
- *clsid*: *STRING* — class identifier associated with the storage.
- *state_bits*: *INTEGER* — last state bits set on element (storage).

The following are the classes that encapsulate the COM constants:

- **ECOM_LOCK_TYPES —** defines the different lock types available for a storage element.
- **ECOM_STAT_FLAGS —** specifies which field of the **ECOM_STATSTG** structure associated with the element to fill.
- **ECOM_STGC —** SToraGe Commit mode; defines the various available commit modes.
- **ECOM_STGM —** SToraGe Mode; defines the opening mode for an element (storage or stream).
- **ECOM_STGMOVE —** SToraGe MOVE; defines how to move a storage.
- **ECOM_STGTY —** SToraGe TYpe; defines the element type: storage, stream or lockbyte.
- **ECOM_STREAM_SEEK** — defines the current position of the seek pointer in a stream.

Summary

The EiffelCOM library gives access to the main compound files functionality. It allows to create, edit and delete compound files. If your application needs to save different types of data in one single file then compound files provide an easy and straightforward support.