Cape Open Interface Revisited In Term Of Class-Based Framework: An Implementation In .Net

Maurizio Fermeglia and Marco Parenzan
DICAMP-MOSE, University of Trieste
Piazzale Europa 1, 34127, Trieste, ITALY
mauf@mose.units.it; marco.parenzan@capeopentoolkit.net

The CAPE-OPEN specifications are the standard for interfacing process modelling software components developed by users into process simulation software (1). CAPE-OPEN standards are defined as a series of interfaces described in a formal documentation set. They are based on universally recognized software technologies developed in the nineties, such as COM and CORBA. Version 1.0 of CAPE-OPEN specifications is implemented in COM (Microsoft Component Object Model). In the last years, Microsoft has designed and developed a new specification for interoperability among different platform for implementing 'cross platform software'. The new development environment is called .NET and it is a great step forward into interoperable and easy-to-develop applications. Since .NET is the evolution of COM technology, it allows the execution of COM codes and the integration of COM code and native .NET code. COM interoperability is the fundamental runtime functionality that allows traditional COM applications using components developed with .NET languages.
In this paper we present the design and the implementation of a software layer developed in .NET framework with C#. This software layer talks with the standard CAPE-OPEN COM interfaces, but allow a much more efficient development of codes for the process engineers. All this, without modifying the CAPE-OPEN standard interfaces included into most process simulation software available in the market today.

1. Introduction

The user who has to design a unit operation in a process simulator needs a clear, simple and self-descriptive schema, in which it is possible to implement the equations without wasting too much time understanding the code-structure. For this purpose, the developer should be able to ignore the effective implementation of CAPE-OPEN interfaces, and to focus only on the methods which the programmer has decided to show him. These methods will have to be strongly standardized so that the user can immediately understand how to use them and how they work directly by looking at the name of the method and the returned value. To "hide" CAPE-OPEN standard it is necessary to separate the implementation of interfaces, which must be unique for all the different projects, from the specific part of code of the unit operation developed. By doing so, the user will have at his disposal only two methods of a given class, which will serve as "constructor" of

the unit and as “calculate method”. This corresponds to a simple and rather clear schema: one method to initialize and another to make it run. Lastly, if he needs a more complex and “powerful” structure, he could anyway benefit by the object-oriented model.
The .NET framework allows the developer to use typical and powerful issues of object orientation such as class inheritance and strong typing. Class inheritance allows the development of new classes inheriting code from parent classes. Many times implementation of interface methods is a repetitive work due to infrastructure needs rather than to a specific need. Class inheritance allows the building of a class framework minimizing the work needed to code a fully functional class. Strong Typing is the ability for a compiler to check all types’ compatibility at compile time (“early binding”). This allows avoiding runtime problems such as “Invalid Cast Exception” due to an extensive usage of Variant (VB) or void* (C/C++). Besides this, a better experience in coding is an extended usage of Intellisense in development tools such as Visual Studio .NET that makes coding a faster and more productive experience.

2. The class based framework

The most important interface in the CAPE-OPEN standard is the ICapeIdentification interface. It is the basis for every structure which requires a name and a description (Units, Ports, and Parameters). Other common elements are some types of error which can occur in each class: ECapeUser, ECapeRoot. Each class therefore must contain an object which allows the error-handling.
Each class will then be derived from the class which implements ECapeUser and ECapeRoot, and the classes describing Units, Ports or Parameters will derive from a class which contains also ICapeIdentification implementation.

Port

The concept of “Port” doesn’t describe an actual element. In other words it represents a vast set of very heterogeneous elements, even if with similar characteristics. Therefore, it is not possible to work on a Port if the type it represents isn’t known. It is then an abstract element which provides the structural basis for each of its specializations (Material Port, Information Port, and Energy Port).

Parameters

The analysis is similar to the previous case: the abstraction of Parameter exists, but the actual element is always a specialization (Integer Parameter, Real Parameter, etc.). This distinction is implicit in the ICapeParameterSpec interface, to which the UnitParameter class refers. Therefore, the management of domain and parameter value is already standardized, thanks to this device, just like in the original VB6 project. To hide the CAPE implementation to the user it is anyway convenient to manage the Parameters as different entities.

Carried material

If the Unit operation has Material Ports, these will carry material fluxes, to which it will be accessed by ICapeMaterialObject interface. This interface exposes, for the setting and the return of property, methods which can be distinguished on the basis of complexity of syntax and of unnaturalness of the sort of returned value. It would thus be convenient to encapsulate these methods in other functions and properties

which can hide the effective implementation and can convert the type of returned value.

Unit

The developer of custom components needs to access only the constructor of the class which represents the Unit, and the Calculate method. It is therefore possible to show the programmer only these two methods. The effective implementation of all the other classes could, and should, be hidden. Consequently, the Unit becomes an abstract concept and the Custom Unit then becomes the real entity to work on. A class is defined, which represents conceptually the Unit and implements CAPE-OPEN interfaces. Another class is defined which allows declaring Ports, Parameters and the Calculate method of the specific Unit.

The resulting class-based framework is described in figure 1. All classes are contained in a .NET assembly that needs to be referenced in each unit project to be developed.

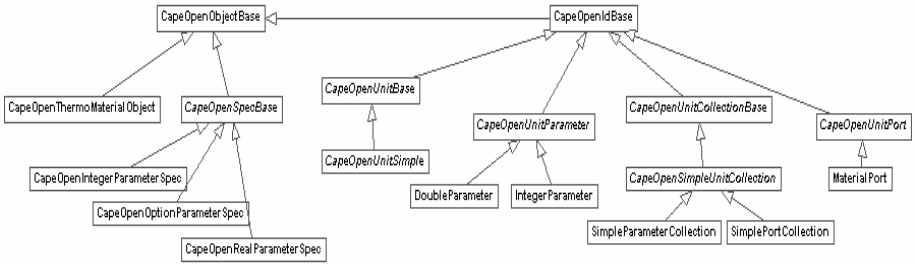

Figure 1: class based framework.

3. Example of application: a dummy test component

One of the main features of the new class based framework is that the component developed by the user is a totally separated project from the infrastructure code. The connection is given by the fact that the class of the new component must inherit from the CapeOpenUnitSimple class in order to allow the process simulator to access the methods of the ICapeUnit interface and those of the interfaces which are connected to it. The user has also to “override” OnInitialize and OnCalculate methods.

Initialization

The creation of ports doesn’t cause many problems even in the original VB6 model, exception made for the fact that, owing to absent-mindedness, a Parameter could be inserted in the collection of Ports and vice versa. The new implementation prevents from this possible error. The creation of a Port, and its insertion in the collection, is obtained by means of the Add method, applied to the “Ports” object, which is of the PortCollection type. In this last’s signature appears the Create static method, which returns a different sort of Port, a Material Port in this case, according to the CapeOpenPortType. If the user tries to insert a Parameter, the error would be pointed out at compilation time and not at runtime as it used to happen before.

The creation of Parameters is very complex in VB. In this version instead the task is definitely simplified. As an example, if one would like to create a Parameter of

the integer type, only the instruction reported in figure 2 are necessary. It is important to note that there is no need to know CAPE-OPEN standards details any more, that the operation now requires much less number of lines of codes than before and that it is no longer required to know the details of the ICapeIntegerParameterSpec interface.

```
protected override void OnInitialize() {
    Name = "DummyUnit";
    Description = "Unit Dummy .NET";
    Ports.Add(CapeOpenUnitPort.Create(CapeOpenPortType.Material,
"InputPort", "Input Port of Dummy", CapeOpenPortDirection.Inlet));
    Ports.Add(CapeOpenUnitPort.Create(CapeOpenPortType.Material,
"OutputPort", "Output Port of Dummy", CapeOpenPortDirection.Outlet));
}
```

Figure 2: code for the implementation of a parameter in the new framework.

Calculate

The first difference that is noticed in the new framework is that the object which represents the material fluxes isn't accessible by ICapeThermoMaterialObject interface but is represented by the CapeOpenThermoMaterialObject class which has all the Properties that allows the simplified calculation of the equilibrium.

The same reasoning is also done for Ports: it is possible to access Ports not by ICapeUnitPort interface, but by CapeOpenUnitPort class (it would be just the same, in this case, to access them by MaterialPort class). The element is returned through index and the order is the same which has been used in the creation of the OnInitialize method. It would be the same to use the GetPortByName method, here applied to the object "Ports".

The tests on direction and on type are the same, but with the difference that the used enumerations are those defined in this new project.

The return of the material flux associated with the Port is obtained in any case by means of the ConnectedObject method. The cast on the returned type is a warranty of consistency among the types.

```
protected override void OnCalculate()
{
    double zTemperature = InputPort.Temperature;
    double zPressure = InputPort.Pressure;
    double zEnthalpy = InputPort.Enthalpy;
    double zTotalFlow = InputPort.TotalFlow;
    double[] zFraction = InputPort.Fraction;

    OutputPort.Temperature = zTemperature;
    OutputPort.Pressure = zPressure;
    OutputPort.Enthalpy = zEnthalpy;
    OutputPort.TotalFlow = zTotalFlow;
    OutputPort.Fraction = zFraction;

    OutputPort.CalculateEquilibrium
       (CapeOpenFlashType.PH);
}
```

Figure 3: implementation of a dummy unit operation with the new framework.

Up to this moment the changes imposed by the new framework are not so relevant to justify the re-implementation of the code. The return of the values of the flux properties in the form of variables makes the difference since they are consistent with the type of quantity they represent. The same applies to the setting of these values. Finally the CalcEquilibrium method is recalled where the possible values of the signature are represented by an enumeration. Figure 3 shows the code necessary for the implementation of the dummy unit operation with the new framework. The code is particularly simple and clean.

4. The CapeOpenToolkit.NET

The code described above is the fundamental part of the development of a fully functional unit, but some infrastructure elements are also necessary. In this project we have developed a complete toolkit to minimize all the complementary activities. The elements of this toolkit are the following.

Unit Wizard

The original CapeOpen Wizard generated a complete Visual Basic 6 project. We have developed a new Wizard that targets the current Microsoft .NET Framework 2.0, generating a compatible Microsoft Visual Studio 2005 project.

Some goodies were added: (i) Load/Save of project generation execution, (ii) Typed parameters and ports, (iii) Multi language targets (C#, Visual Basic .NET) and (iv) COM Registration metadata

Regunitasm.EXE

The .NET unit implementation cannot avoid the necessary COM plumbing. The new project template contains all the information needed for a registration activity, now saved at assembly level. When the generated assembly is installed in the target machine containing the process simulator, this utility can register the unit with a simple command:

```
REGUNITASM <unit.dll> <Unit Type fullname>
```

No Windows COM registry info will be needed to perform this tedious operation.

5. Conclusions

The objective of the development of a new framework, as it was stated at the beginning of this project, is the development of software which allows the creation of a component module to be used in process simulators in a simple and self-evident way. The given solution has actually solved most of the problems which appeared in the previous models and has thoroughly reached the set goals. In summary, the following items evidence the benefits and the significance of the new framework.

1. the re-writing of a component is effectively accessible also to a chemical engineer who has a basic knowledge of informatics;
2. most of errors are pointed out during compilation, thus meaning a faster debug of the code;

3. it is possible to exploit more complex data-structures, which are available in .NET;
4. a reference handbook of few pages would allow the user to access most of the functions he is interested in without concerning about CAPE-OPEN standard details;
5. the inheritance of the classes allows writing base-components which can be re-used many times and which are extendible with further inheritance.

The last point is particularly interesting. Assume, as an example, that one should develop different unit operations, all having a single inlet Port and a single outlet Port with 'some' modification of the material streams inside the unit. The way to proceed is the following. First the user creates a class, made abstract, that basically does what reported in figure 3. After, the user will concentrate on the derived classes, one for each module to be created, each one containing a method, inserted between the returning and the setting of properties, with the implementation of the equations relating input and output streams. By doing this, there is no need to know how Ports and Streams are represented in programming language.

We plan to further develop the framework with the final goal of the complete hiding of the code to the final user in favor of a developer-environment oriented to the process engineer user. This environment will allow the user to write the equations describing the unit operation to be developed without taking care of how the equations are translated in effective code. This operation will be simple also for a process engineer who has no familiarity at all with informatic tools and code.
The last step of the development of the framework will be the implementation of an application developed specifically for the end-user, in which she/he will write the equation directly in a 'natural way' into it. Since it is not possible to separate informatics from engineering, there is the need to provide the final user (a chemical engineer) with a simplified and self-explaining development environment which doesn't require excessive notions of informatics, like Polymath or Matlab. This project has the aim to open a road towards that goal.

Thanks to Marco Carone and Letitia Toma for their contribution in toolkit creation.

References

CO-LaN. *CO Tester Suite.* http://www.co-lan.org/index-10.html.
CO-LaN Methods & Tools Special Interest Group. *.NET Interoperability Guidelines.* CO-LaN, 2006.
CO-LaN. *Migration Support.* http://www.co-lan.org/index-9.html.