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 standards for interfacing process modelling software components developed to be used in 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 (Component Object Model by Microsoft). 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 definitely a 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.
[CITATION OF .NET CHARS]
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.

DEVELOPER EXPERIENCE

The user who has to design a unit will need a clear, simple and self-descriptive schema, in which he can implement the equations without wasting too much time understanding the code-structure. For this purpose, he should be able to ignore the effective implementation of CAPE-OPEN interfaces, and to focus only on the methods which the implementer has decided to show him. These methods will have to be strongly standardized in order that the user can immediately understand how to use them and how they work by means of 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 a single one for all the projects, from the specific part of its own unit. The user will so have at his disposal only two methods of a given class, which will respectively serve as “constructor” of the unit and as Calculate method (a simple and clear schema: one method to initialize and the other to make it run). Lastly, if he needs a more complex and “powerful” structure, he could anyway benefit by the object-oriented model.

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 and not a specific issue needs. 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; something classed “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.

COMMON INTERFACES ANALYSIS

Studying CAPE-OPEN interfaces it will be noticed that ICapeIdentification interface 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 every class): ECapeUser, ECapeRoot. Each class therefore must contain an object which allows the error-handling.
Each class will so 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): it is not possible, in fact, 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).

PARAMETER

The analysis should be the same as before: 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. So, like in the VB version, management of domain and parameter value is already standardized thanks to this device. To hide the CAPE implementation to the user it is anyway convenient to manage the Parameters as different entities

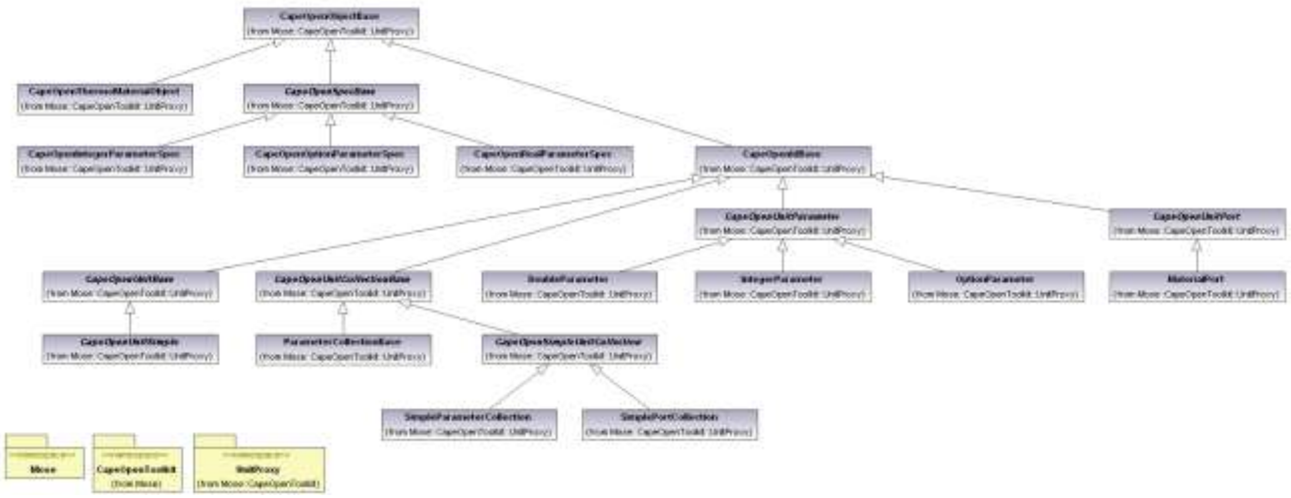
CARRIED MATERIAL

If the Unit has got 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). The Unit so becomes an abstract concept and the Custom Unit then becomes the real entity to work on. There will exist then a class, which represents conceptually the Unit and implements CAPE-OPEN interfaces, and another class which allows declaring Ports, Parameters and the Calculate method of the specific Unit.

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



DEVELOPMENT OF A DUMMY TEST COMPONENT

The component which the user is going to develop is a separated project from the infrastructure code. The connection is represented in the fact that the class of the new component must inherit from the CapeOpenUnitSimple class in order to allow the 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 VB model, exception made for the fact that, owing to absent-mindedness, there could be inserted a Parameter in the collection of Ports and vice versa. The new implementation prevents from this error. The creation of a Port (and its inserting 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 should try 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 problem is definitely simplified. As an example, if there should be the will of creating a Parameter of the integer type, only the following instruction will be needed (notice that with it there is no need of knowing CAPE-OPEN standards and that before the same operation required several code lines together with the familiarity with 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));
}
```

CALCULATE

The first difference to be noticed is that the object which represents the material fluxes isn't accessible by ICapeThermoMaterialObject interface but is represented by the CapeOpenThermoMaterialObject class: it thus has those Properties which allow the simplified calculation of the equilibrium. It is the same for Ports: it is possible to access them 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 (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 returning of the material flux associated with the Port is obtained always by means of the ConnectedObject method (the cast on the returned type is a warranty of coherence among the types).

Up to this moment there haven't been so many changes to justify the re-implementation of the code. Now instead the returning of the values of the flux properties in the form of variables coherent with the sort of quantity they represent gives a more than sufficient justification for this work. The same can be said about the setting of these values.

Finally the CalcEquilibrium method is recalled. In it the possible values of the signature are represented by an enumeration.

```
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);
}
```

CAPEOPENTOOLKIT.NET

The code described above is now the prominent part of the development of a fully functional unit. Some infrastructure items are again necessary: the main difference is that no more coding is required. A complete toolkit was developed to minimize these activities.

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:

- Load/Save of project generation execution
- Typed paramenters and ports
- Multi language targets (C#, Visual Basic .NET)
- COM Registration metadata

REGUNITASM.EXE

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

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

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

CONCLUSIONS

The aim which has been set at the beginning of this project is the development of software which allows the creation of a component used in the chemical-process simulators in a simple and self-evident way, since the currently used instruments don’t make this possible. The given solution has actually solved most of the problems which appeared in the previous models and has thoroughly reached the set goals:

1. the re-writing of a component is effectively accessible also to a chemical engineer who has a basic college knowledge of informatics;
2. most of errors are pointed out during compilation, and this warrants a faster debug of the code;
3. it is possible to exploit more complex data-structures, which are put at disposal by .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;
5. The inheritance of the classes allows writing base-components which can be used over and over again and which are extendible with further inheritance: assume, as an example, that there should be the need for different Units which must have a single inlet Port and a single outlet Port (like the case just mentioned). If the newly created class would be made abstract and a method which had to be implemented in a different way in each of the derived classes would be inserted between the returning and the setting of properties, in these there could be written just the state equations between in and out fluxes without needing to know how Ports and Fluxes are represented in programming language.

A further development of the code will be able to bring to the complete hiding of the code itself to the final user in favor of a develop-environment which should allow realizing the balancing equations without concerning about how they are translated in effective code. This operation could be made even by a chemical engineer who has no familiarity with informatics at all.
This last point must become the goal to achieve in future times. 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 developmental environment which doesn’t require excessive notions of informatics. This project has therefore the aim, on one side, to open a road towards that goal, and on the other side, to provide anyway a environment for the development of components which aren’t based only on flux-balancing but are instead meant to make operations also at a software level.

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.