CAPE-OPEN

Expanding Process Modelling Capability
through Software Interoperability Standards

Errata and Clarifications for

Utilities Common Interface specification

www.colan.org

ARCHIVAL INFORMATION

Filename	Utilities_Errata_1.0.docx
Authors	CO-LaN consortium: M&T SIG
Status	Approved for public release
Date	January 2018
Version	Version 1.024
Number of pages	13
Versioning	0.000 created on Dec 3, 2013 by Bill Barrett (USEPA)
	1.001 Created by Bill Barrett on April 1, 2014
	1.002 Created by Bill Barrett on March5, 2014
	1.003 Created by Bill Barrett on March 19, 2014
	1.004 Created by Bill Barrett on April 1, 2014
	1.005 Created by Bill Barrett on April 1, 2014
	1.006 Edited by Jasper van Baten on April 21, 2014
	1.007 Edited by Bill Barrett on May 20, 2014
	1.011 edited by Bill Barrett on July 24, 2014
	1.012 edited by Michel Pons on July 28, 2014
	1.014 edited by M&T SIG on November 4, 2015
	1.018 edited by M&T SIG on July 7, 2016
	1.021 edited by CTO on December 14, 2017
	1.022 edited by M&T SIG on January 3, 2018
	1.023 modified by M&T SIG on January 10, 2018
	1.024 edited by CTO on January 17, 2018
Additional material	
Web location	
Implementation specifications version	Version 1.0
Comments	

IMPORTANT NOTICES

SUMMARY

The Errata and Clarifications document for the Utilities Common Interface specification 1.0 provides clarification of the distinction between primary and secondary PMCs, as well as about the requirement for primary PMCs to implement the *ICapeUtilities* interface. The document provides a mechanism for the *ICapeUtilities::Edit* method to indicate whether changes were made to a PMC through the use of the S_FALSE (0x01) HRESULT. The document justifies allowing the Simulation Context to be set on a PMC prior to initializing the PMC. The document provides clarification on the lifecycle of a CAPE-OPEN Primary PMC object, indicating the process for creating an instance of the object, setting the Simulation Context, restoring the object from persistence, calling the *ICapeUtilities::Initialize* method and *ICapeUtilities::Terminate* method. The document clarifies operations that can be performed by the PME on the PMC during each of these stages of a PMC's lifecycle.

CLARIFICATIONS

1. Requirement to implement *ICapeUtilities*

***Context*:** implementation of *ICapeUtilities* is necessary for a PMC to expose a Collection of Parameters and to configure the PMC using an *Edit* method.

***Issue*:** the requirement to implement *ICapeUtilities* is not clearly stated within each CAPE-OPEN Business Interface Specification.

***Discussion*:** the summary of the Utilities Common Interface document [0] states that "This interface has to be provided by any PMC primary object. That allows any PME to manage Simulation Context, to collect Parameters of PMC and to edit the PMC." The Introduction to the Utilities Common Interface Specification states that the notion of a primary and secondary PMC is provided by the Method and Tools (M&T) Integrated Guidelines [0]. Section 9.2.1 of the M&T Integrated Guidelines defines primary and secondary PMCs.

Under this model, the SIGs responsible for proposing PMCs are responsible for stating which PMCs are PMC primary objects requiring implementation of *ICapeUtilities*. In general, any category of PMC that requires editing, parameterization, access to the Simulation Context of the PME, or that references any PME object (and therefore must act on *Terminate*), needs to implement *ICapeUtilities*.

The *ICapeUtilities* interface was developed concurrently with the development of the CAPE-OPEN Business Interface specifications. As such, some business specifications either do not indicate the need for the primary PMC objects to implement the *ICapeUtilities* interface, or the interface specification incorporates methods from the *ICapeUtilities* into the interface design. The present document is intended to provide consistency and develop uniformity in the requirement for implementation of *ICapeUtilities*.

Clarification 1: PMC primary objects are PMCs that can be instantiated directly by the PME or indirectly through the use of a manager (e.g. objects implementing *ICapeThermoSystem* or *ICapeThermoPropertyManager*).

Clarification 2: Table 1 lists all PMCs that are currently considered PMC primary objects within the Business Interface Specification that defines them.

***Requirement 1*:** all PMC primary objects must implement the *ICapeUtilities* interface.

An exception to requirement 1 is that some interfaces for PMC primary objects have been designed prior to the definition of *ICapeUtilities* (see discussion above).These interfaces provide services similar to the *ICapeUtilities* interface. These PMC primary objects do not need to implement *ICapeUtilities*. Such PMC primary objects are identified in Table 1. If specifications for these objects are updated in the future, these Primary PMC objects will be required to implement *ICapeUtilities* and the defining interface for the PMC will not include any *ICapeUtilities*-like functionality.

2. *ICapeUtilities.Edit* Return Value

***Context*:** editing of Process Modelling Component (PMC) objects is performed using the CAPE-OPEN *ICapeUtilities::Edit* method. It would be useful to a Process Modelling Environment (PME) to know whether a call to *ICapeUtilities::Edit* method resulted in a change to any aspect of the PMC. This would allow the PME to decide whether to rescan the PMC to update any information presented by the PME's graphical user interface, whether to mark the Unit Operation as not solved, etc.

***Issue*:** the *ICapeUtilities::Edit* method currently does not have a return value to indicate whether a change has occurred to the PMC as a result of the editing.

***Discussion 1*:** use of a return value for the *ICapeUtilities::Edit* method would allow the PMC to communicate whether call to the *ICapeUtilities::Edit* method resulted in a change to the PMC. Two options were considered:

- Boolean value: A return of *TRUE* would indicate that a change had occurred and a return of *FALSE* would indicate that no change had been made.
- An enumerated type, *CapeEditResult*: Two values could be used, (0) for *CAPE_MODIFIED*, and (1) *for CAPE_NOT_MODIFIED*

Modifying the *ICapeUtilities::Edit* method to return the *CapeEditResult* enumerated type provides a clearer indication of the result of the call to the *ICapeUtilities::Edit* method, and is the selected option. In order to make this change, the *ICapeUtilities* Interface Specification document will need to be updated to reflect the change to the return value of the *ICapeUtilities::Edit* method. Modification of the *ICapeUtilities* Interface Specification will be completed as part of the creation of the CAPE-OPEN Binary Interoperability Architecture (COBIA).

***Discussion 2*:** currently, there is a desire to provide the PME with the information regarding whether the PMC was modified by the call to the *ICapeUtilities::Edit* method for existing COM-based CAPE-OPEN implementations. Since the proposed enumerated type return value can be treated as an integer, use of the COM *HRESULT* return value has been identified as a means to return this information. The COM-based function signature of the *ICapeUtilities::Edit* method utilizes the standard COM *HRESULT* return value. This section discusses the implications of the use of the *HRESULT* to provide the *CapeEditResult* return value in existing COM implementations.

The *HRESULT* is a 32-bit signed integer used to indicate whether the function was successful, through the return of a non-negative value, or an error occurred, through the return of a negative value. In standard usage, returning an *HRESULT* value of *S_OK* (0) indicates that an error did not occur during performance of the function, and *S_OK* is typically the only success *HRESULT* used in CAPE-OPEN.

The table below shows the evaluation of whether there would be any side effects of adopting this option. Non-negative *HRESULT* values indicate success conditions, and negative results indicate an error/failure occurred during the method call. COM provides two macros to determine whether a call to a COM method resulted in an error, *SUCCEEDED* and *FAILED. SUCCEEDED* returns a Boolean *TRUE* value if the function call returns a non-negative *HRESULT*, indicating that the method call was successful; and *FAILED* returns a Boolean *TRUE* if the function call returns a negative value, indicating an error occurred.

	CapeEditResult	*HRESULT*	*Numerical value*	*SUCCEEDED return value*	*FAILED return value*
OK and modified	*CAPE_MODIFIED*	*S_OK*	*0*	*TRUE*	*FALSE*
OK and not modified	*CAPE_NOT_MODIFIED*	*S_FALSE*	*1*	*TRUE*	*FALSE*
Error	*Not applicable*	*Any HRESULT error value*	*Any HRESULT error value*	*FALSE*	*TRUE*

***Recommendation 1*:** modify the *ICapeUtilities* Interface Specification as part of COBIA development to incorporate an enumerated type, *CapeEditResult*, valued return for the *ICapeUtilities::Edit* method during the creation of the CAPE-OPEN Object Model.

***Recommendation 2*:** in COM-based CAPE-OPEN implementations, a workaround can be made using the *HRESULT* result value to provide the equivalent of the *CapeEditResult* value. A PMC may return an *S_FALSE HRESULT* value if no changes have been made during the invocation of the *ICapeUtilities::Edit* method. The PME may interpret the *S_FALSE HRESULT* as an indication that the PMC was not modified.

We expect no backward compatibility problems for COM-based PMEs that use the *SUCCEEDED* and *FAILED* macros to test the *ICapeUtilities::Edit* return value as these macros return TRUE for *S_OK* and *S_FALSE*. We also expect no backward compatibility issues for existing COM-based PMCs returning *S_OK*. PMEs that test the *ICapeUtilities::Edit* return value by erroneously comparing it to S_OK, and erroneously conclude that *Edit* has failed on *S_FALSE* returns: in this case, the behaviour of the PME is undefined as the PME will likely consider this an edit failure, and may proceed as if the PMC does not have an editor [see Unit Use Case UC-31-017 Set Unit Specific Data].

***Usage Notes*:**

For .NET-based implementations of the *ICapeUtilities::Edit* method, the *ICapeUtilities::Edit* method signature in the CO-LaN provided Primary Interop Assembly (PIA) has been modified to return an int32 value. The *CapeEditResult::CAPE_MODIFIED* value can be returned by returning zero (0). The *CapeEditResult::CAPE_NOT_MODIFIED* value is returned by returning one (1).

As a result, a .NET PME calling a COM PMC will receive an integer having a value of zero (0) for CAPE_MODIFIED and a value of one (1) for CAPE_NOT_MODIFIED. Likewise, the .NET PMC called by a COM PME will return a value of zero (0) for CAPE_ MODIFIED and one for CAPE_NOT_MODIFIED. Conversion between the enumeration for the COM and the .NET integer is handled by the standard COM/.NET interoperability.

3. Ability to set the Simulation Context prior to calling *ICapeUtilities.Initialize*

***Context*:** PMCs are not considered fully initialized until a call to *ICapeUtilties::Initialize* has been completed. However, the Simulation Context provides logging tools that may be useful to diagnose problems occurring during instantiation, restoration from persistence and initialization process.

***Issue*:** the mechanism for the PMCs to provide information to the PME and Flowsheet User is through the *ICapeDiagnostic* interface available on the Simulation Context set by the PME through the *ICapeUtilities* interface. In order to provide the PMC with access to *ICapeDiagnostic* capabilities, PMEs need to be able to set the value of the *ICapeUtilities::SimulationContext* property after object instantiation, after calling middleware-specific object initialization or restoration from persisted states and prior to calling the *ICapeUtilities::Initialize* method. The benefit of setting a Simulation Context early in the object life cycle is that it would allow use of the CAPE-OPEN logging methods in the *ICapeDiagnostic* interface.

***Discussion*:** the Simulation Context object is a means by which the PME exposes itself to the PMC. The Simulation Context object provides the PMC with access to the PME's diagnostic, Material Template system, and other PME-provided services through support of the *ICapeDiagnostic*, *ICapeThermoMaterialTemplateSystem* and *ICapeCOSEUtilities* interfaces. Of particular interest is access to the diagnostic facilities through the *ICapeDiagnostic* interface, which provides two useful methods, *PopUpMessage* and *LogMessage* that allow PMCs to communicate to users/developers through the PME, providing an avenue to present useful information regarding the state of the PMC during the object creation, restoration, and initialization process.

Developers have asked to be able to access the logging capabilities of the *ICapeDiagnostic* interface immediately upon creation of the object from the underlying middleware (in COM, creation via *CoCreateInstance*). However in COM the persistence machinery (*InitNew* or *Load*) must be invoked prior to all other calls. Any issue to be logged resulting from object restoration problems should therefore be logged during *Initialize*.

***Requirement*:** PMCs that implement *ICapeUtilities* will allow PMEs to set the Simulation Context via the *SetSimulationContext* method prior calling *ICapeUtilities::Initialize*.

4. Object Lifecycle

***Context*:** descriptions of the object lifecycles exist in the Utilities Common Interface Specification, the Unit Operation Interface Specification [0], the Thermodynamic and Physical Properties 1.0 [0] and 1.1 [0] Interface Specifications and the Persistence Common Interface Specification [0]. In addition, the *ICapeUtilities::Terminate* method is not called by some PMEs when the PME is destroying an object. This can lead to circular references and therefore memory leaks, as PMCs are expected to release CAPE-OPEN referenced secondary objects during the *ICapeUtilities::Terminate* method.

***Issue*:** object lifecycle management is not clearly defined in a single location within the specifications.

***Discussion*:** the overall object lifecycle includes both the middleware-specific lifecycle, and the CAPE-OPEN-specific object lifecycle. The overall lifecycle can generally be summarized as creation within the middleware, initialization into CAPE-OPEN operations, CAPE-OPEN usage, end of CAPE-OPEN life, and finally end of middleware life. However, there may be a need to make certain CAPE-OPEN capabilities available immediately, upon object creation in middleware, prior to the initiation of the CAPE-OPEN object lifecycle. As most commercial CAPE-OPEN implementations are developed in the COM middleware, the following section describes the overall lifecycle of a COM-based CAPE-OPEN object.

All COM-based PMCs have a CAPE-OPEN lifecycle within their COM life cycle. This means that COM reference counting rules need to be adhered to. In COM, the reference count is incremented by a call to either *IUnknown::QueryInterface*, or *IUnknown::AddRef*. The reference count is decreased by a call to *IUnknown::Release*. When the reference count reaches zero (0), the object deletes itself. There are three rules within COM related to reference counting [0]:

1. **Call *AddRef* before returning.** All functions that return an interface should call *AddRef* on the pointer before returning. This includes *QueryInterface* and *CreateInstance* functions. This way, the caller does not need to call *AddRef* on the interface received.
2. **Call *Release* when you are done.** When you are finished with an interface, you should call *Release*.
3. **Call *AddRef* after assignment.** Whenever you assign an interface pointer to another interface pointer, you should call *AddRef*. The assignment creates another reference to the interface which eventually must be released.

COM objects are created without having an internal state, and must be initialized to be in a useful state. In COM, object initialization is separated from object creation in order to avoid initializing objects to a default state only to immediately load previously stored data. When a COM object is created in C++ using the *CoCreateInstance* method, a reference is obtained to the requested interface of the created object.

The PME should select desired persistence mechanisms for the PMC based upon the support of the various COM persistence interfaces by the PMC and the persistence needs of the PME. Appropriate calls to the COM persistence interface's *InitNew* method, if supported, should be made to configure the PMC to use the persistence interface. In particular, the *IPersistStorage* interface requires an instance of an *IStorage* object during its lifetime, which is provided by its *InitNew* method. Instead of using *InitNew*, the PMC may be restored from persistence by calling the *Load* method for the persistence interface being used to restore the PMC.

PMC objects are expected to expose the *ICapeIdentification* and, if required, *ICapeUtilities* interfaces. The PME may set the *SimulationContext* property on the PMC and calls *ICapeUtilities::Initialize*. Prior to the call to *ICapeUtilities::Initialize*, other methods on *ICapeUtilities* or any method on any other CAPE-OPEN interface (including *ICapeIdentification*), cannot be expected to function, and may return an error HRESULT if called.

The *ICapeUtilities::Terminate* method provides a mechanism for the PMC to release all CAPE-OPEN related resources. All external references must be released at this point, including Simulation Context, Material Objects connected to Ports and the active Material Object on the Property Package. In the case of a primary PMC that contains secondary PMCs, the primary PMC needs to manage the life cycle of any contained PMCs. For example, a Unit Operation PMC must delete its Port and Parameter Collections. After an object is terminated using the *ICapeUtilities::Terminate* method, no CAPE-OPEN method call can be made on the object.

All references to the object can then be released using middleware-defined low-level methods used to control object life cycle. In COM, this means a call to the *IUnknown::Release* method for all acquired COM interfaces on the PMC object. These include persistence interfaces, *ICapeUtilities*, *ICapeIdentification* and the interface returned by *CoCreateInstance*.

The lifecycle for COM-based CAPE-OPEN PMCs is as follows:

1. PME creates PMC (*e.g.,* call *CoCreateInstance*).
2. PME determines the persistence interface to be used, discussed above.
3. PME calls the *InitNew* method, if included, on the desired persistence interface or Restore object from persistence using the *Load()* method on the desired persistence interface (see note below about VB6).
4. PME sets the Simulation Context using *ICapeUtilities::put_SimulationContext*
5. PME calls *ICapeUtilities::Initialize*
    a. Parameter Collection is now available to the PME via *ICapeUtilities::Parameters.*
    b. For Unit Operation PMCs, the Port Collection is available, and Ports are available for connection of stream objects.
    c. The PMC can now be edited by calling *ICapeUtilities::Edit.*
6. PME places the PMC into a PMC Collection
    a. PME ensures that the PMC has a unique *ICapeIdentification::ComponentName*
7. PME uses PMC in simulation, including, as appropriate:
    a. Calls *ICapeUtilities::Edit*,
    b. Obtains Parameter Collection via *ICapeUtilities::GetParameters,*
    c. Exercises the PMC
        i. For Unit Operations:
            1. Obtains Port Collection via *ICapeUnit.GetPorts.*
            2. Calls *ICapeUnit::Validate* and *ICapeUnit::Calculate.*
            3. Accesses reports.
        ii. For other PMCs, exercises the PMC as appropriate
    d. Persists the object to the selected persistence interface.
8. PME terminates the PMC
    a. PME releases all references to PMC secondary objects, including:
        i. Release references to individual Parameters
        ii. Release References to the Parameter Collection
        iii. For Unit Operations:
            1. Disconnect all Unit Ports.
            2. Release references to individual Ports
            3. Release references to the Port Collection
        iv. Release any other references to PMC secondary objects
    b. PME calls *ICapeUtilities::Terminate*
        i. PMC releases all external references

        ii. PMC releases the Simulation Context.
9. Release all interface references to the PMC primary object, including
    a. *ICapeUtilities* interface
    b. *ICapeIdentification* interface.
    c. Any persistence interfaces.
    d. PMC’s interface.

Note: During object creation, Visual Basic 6 (VB6) calls the *IPersistStreamInit.InitNew()* method, regardless of whether the object is created using the *New* method or the *CreateObject* method. As such, there appears to be no method to create the object without calling the *IPersistStreamInit.InitNew()* method, violating the *IPersistStreamInit* interface’s requirement that *InitNew()* and *Load()* methods not be called on the same instance of an object. PMCs should not raise an error condition if both these methods are called.

Table 1. List of Current PMC primary objects defined in Business Interface Specification documents.

Primary PMC Type	**Requires Implementation of *ICapeUtilities***	**Registered as category**
Unit Operation interface specification 1.0		
Unit Operation	Yes	CapeUnitOperation_CATID {678c09a5-7d66-11d2-a67d-00105a42887f}
Thermodynamic and Physical Properties interface specification 1.0		
Property Package	Yes	CapeThermoPropertyPackage_CATID {678c09a4-7d66-11d2-a67d-00105a42887f} Or created by a Thermo System
Calculation Routine	Yes	CapeExternalThermoRoutine_CATID {678c09a2-7d66-11d2-a67d-00105a42887f}
Equilibrium Server	Yes	CapeThermoEquilibriumServer_CATID {678c09a6-7d66-11d2-a67d-00105a42887f}
Thermo System	Yes	CapeThermoSystem_CATID {678c09a3-7d66-11d2-a67d-00105a42887f}
Thermodynamic and Physical Properties interface specification 1.1		
Physical Property Calculator	Yes	CapePhysicalPropertyCalculator_CATID {CF51E385-0110-4ED8-ACB7-B50CFDE6908E}
Equilibrium Calculator	Yes	CapeThermoEquilibriumCalculator_CATID {CF51E386-0110-4ED8-ACB7-B50CFDE6908E}
Property Package	Yes	CapeThermoPropertyPackage_CATID {CF51E384-0110-4ED8-ACB7-B50CFDE6908E} Or created by a Property Package Manager
Property Package Manager	Yes	CapeThermoPropertyPackageManager_CATID {CF51E383-0110-4ED8-ACB7-B50CFDE6908E}
Chemical Reaction Package interface specification 1.1 (under development)		
Reactions Package Manager	Yes	CAPEOPENReactionsPackageManager_CATID {678c09aa-0100-11d2-a67d-00105a42887f}
Reactions Package	Yes	CAPEOPENReactionsPackage_CATID {678c09ab-0100-11d2-a67d-00105a42887f} Or created by a Reactions Package Manager

Primary PMC Type	**Requires Implementation of *ICapeUtilities***	**Registered as category**
Numerics interface specification		
Solver Package	No	CapeSolversPackage_CATID {79DD785E-27E5-4939-B040-B1E45B1F2C64}
Optimization interface specification		
MINLP Solver Package	No	CapeMINLPSolverPackage_CATID {678c09ac-7d66-11d2-a67d-00105a42887f}
Planning and Scheduling interface specification		
Planning and Scheduling Package	No	CapePSPPackage_CATID {3EFFA2BD-D9E7-4e55-B515-AD3E3623AAD5}
Sequential Modular Specific Tools interface specification		
Sequential Modular Specific Tools	No	CapeSMSTPackage_CATID {678c09ab-7d66-11d2-a67d-00105a42887f}
Flowsheet Monitoring interface specification		
Flowsheet Monitoring Object	Yes	CapeFlowsheetMonitoringComponent_CATID {7BA1AF89-B2E4-493d-BD80-2970BF4CBE99}
Physical Property Data Base interface specification		
Physical Property Data Base	No	CapePPDBService_CATID {678c09aa-7d66-11d2-a67d-00105a42887f}

5. References

CAPE-OPEN Interface specifications: Utilities Common Interface

CAPE-OPEN Methods and Tools integrated guidelines

CAPE-OPEN Interface Specifications: Unit Operation Specification

CAPE-OPEN Interface Specifications: Thermodynamic and Physical Properties Version 1.0

CAPE-OPEN Interface Specifications: Thermodynamic and Physical Properties Version 1.1

CAPE-OPEN Interface Specifications: Persistence Common Interfaces

Rogerson, D. (1997). Inside COM. Redmond, Washington, Microsoft Press