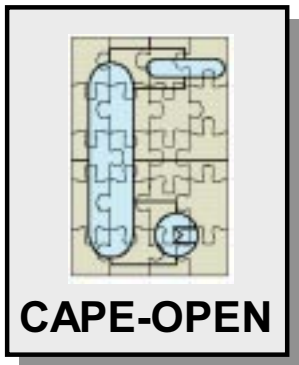


# **CAPE-OPEN Methods and Tools Guidelines**



**The Methods and Tools Group**

*CO-MGT-M&T Version 1 Januar 2000*

# IMPORTANT NOTICES

## **Disclaimer of Warranty**

CAPE-OPEN documents and publications include software in the form of *sample code*. Any such software described or provided by CAPE-OPEN --- in whatever form --- is provided "as-is" without warranty of any kind. CAPE-OPEN and its partners and suppliers disclaim any warranties including without limitation an implied warrant or fitness for a particular purpose. The entire risk arising out of the use or performance of any sample code --- or any other software described by the CAPE-OPEN project --- remains with you.

**Copyright © 1999 CAPE-OPEN and project partners and/or suppliers.** All rights are reserved unless specifically stated otherwise.

CAPE-OPEN is a collaborative research project established under BE 3512 "Industrial and Materials Technologies" (Brite-EuRam III), reference BRPR-CT96-0293.

## **Trademark Usage**

Many of the designations used by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in CAPE-OPEN publications, and the authors are aware of a trademark claim, the designations have been printed in caps or initial caps.

Microsoft, Microsoft Word, Visual Basic, Visual Basic for Applications, Internet Explorer, Windows and Windows NT are registered trademarks and ActiveX is a trademark of Microsoft Corporation.

Netscape Navigator is a registered trademark of Netscape Corporation.

Adobe Acrobat is a registered trademark of Adobe Corporation.

Visio is a registered trademark of Visio Corporation.

## CAPE-OPEN Archival Information

<b>Reference</b>	CO-M&T
Coordinated by	IFP
Date	30 September 1999
Number of Pages	27
Version	Version 1.00
Filename	CAPE-OPEN Methods and Tools Guidelines
<b>Developmental Editor(s)</b>	Bertrand Braunschweig, IFP
<b>Contributor(s)</b>	Michael White, Aspentech Juan Carlos Rodriguez, DuPont Sergi Sama, Hyprotech Lars Von Wedel; RWTH.LPT Mike Williams, QuantiSci Peter Edwards, DuPont Stefan Zlatintsis, RWTH.I5 Curt Arnold, Hyprotech Andre Terroux, SimSci Ralf Bogusch, RWTH.LPT Andy Lui, Aspentech Joe de Almeida, SimSci



## Summary

This document gathers the two main Methods and Tools (M&T) recommendation documents of CAPE-OPEN:

- The initial text "Methods and Tools within CAPE-OPEN Project", an appendix section of the Conceptual Design Document (CDD2) released in 1997; this appendix itself summarized a 100-pages document authored by the M&T group;
- The document "Interfaces Guidelines and Review", an internal project document published in November 1997 following a technical session of the M&T group. The "Review" part was removed from the text.

The first document (chapter 1) presents the conclusions of the Methods and Tools Group, in charge of making the choices on methods and software tools for the CAPE-OPEN project and the CAPE-OPEN standard. The main recommendations are to use the UML notation and to implement the open interfaces both for OLE/COM and CORBA.

The second document (chapter 2) provides guidelines for developing CAPE-OPEN interface specifications. It addresses important software analysis and procedural issues and makes specific recommendations in order that work package teams can create uniform release documents.

## CAPE-OPEN Document Roadmap

This document is intended primarily for software engineers and process simulation users, who are interested in understanding and applying the CAPE-OPEN development process.

All other readers need not go beyond **page 3**.

## Acknowledgements

The authors of this document wish to acknowledge the contribution of the other participants to the CAPE-OPEN project. Without their help, thoughts and technical expertise we would not have been able to produce this document. We would also like to thank the companies involved for supporting this project and providing the significant resources needed to carry out the work.

## Contents

<b>1. METHODS AND TOOLS WITHIN CAPE-OPEN PROJECT.....</b>	<b>1</b>
1.1. MAIN TECHNICAL CHOICES.....	1
1.2. NOTATION AND INTERFACE DEFINITION LANGUAGES FOR THE CAPE-OPEN STANDARD .....	1
1.3. METHODS AND TOOLS FOR THE CAPE-OPEN PROJECT .....	2
1.4. WORK PROCESS .....	2
<b>2. SOFTWARE ENGINEERING STANDARDS FOR CAPE-OPEN.....</b>	<b>11</b>
2.1. INTRODUCTION.....	11
2.2. NAMING .....	12
2.3. ARGUMENT ORDERING.....	15
2.4. DATA TYPES .....	15
2.5. PROPERTIES AS ATTRIBUTES .....	17
2.6. REGISTRY .....	18
2.7. ARCHITECTURAL ASPECTS OF INTERFACES .....	18
2.8. RUNNING COMPONENTS IN-PROCESS, OUT-OF-PROCESS (LOCAL AND REMOTE) .....	21
2.9. IDENTIFICATION .....	21
2.10. ERROR HANDLING.....	21
2.11. CAPABILITIES FOR WHICH NO COMMON INTERFACES WILL BE PROVIDED .....	22
2.12. VERSION CONTROL .....	23
2.13. THE REPOSITORY.....	24
2.14. APPENDIX: SOME RATIONALE.....	25

## Figures and tables

FIGURE 1: SUMMARY OF WORK PROCESS .....	3
FIGURE 2: USE CASE EXAMPLE.....	4
FIGURE 3: SEQUENCE MODEL .....	5
FIGURE 4: INTERFACE MODEL.....	6
FIGURE 5: COMPONENT MODEL .....	6
FIGURE 6: ITERATIVE PROCESS .....	9
FIGURE 7: EARLY AND LATE BINDING.....	26
TABLE 1 :WORK PROCESS PHASES.....	3
TABLE 2: METHODS NAMING .....	13
TABLE 3: CO DATA TYPES .....	16

# 1. Methods and Tools within CAPE-OPEN Project

## 1.1. Main Technical Choices

The following technical decisions are supported by work done during the OO-CAPE and OS-CAPE projects and by individual achievements of many partners involved in the development of modern software for process simulation.

1. The standard interfaces should be defined and expressed using an **object-oriented** approach. The OO approach is currently the best technical solution for developing interface standards. It also encompasses the « conventional » procedural approach.
2. The standard interfaces assume that a process simulator is made of several **components**.
3. The standard interfaces should use existing **middleware**, namely ActiveX/COM and CORBA. More specifically, the standard interfaces should be expressed **in both forms** in order to be future-proof.
4. The standard interfaces should be applicable **to several hardware platforms and operating systems**.
5. There should be a distinction between **project work** and **the standard**. The life-span of the standard is expected to be much greater than the duration of the initial CAPE-OPEN project itself. Choices made for the project - because it has limited resources and duration - should not compromise the quality and scope of the standard.
6. The standard interfaces should allow the encapsulation of **legacy code**.

## 1.2. Notation and Interface Definition Languages for the CAPE-OPEN Standard

### 1.2.1. Notation

We adopt the Unified Modelling Language (UML) for the CAPE-OPEN set of object models. UML unifies the popular OMT, Booch and Jacobson methods for object-oriented projects, and it is becoming a de facto standard with high acceptance from the OMG. UML is seen as the way forward for CAPE-OPEN. Appendix A2 provides an introduction to the UML notation.

### 1.2.2. Interface Specifications

We will express the interface specifications **both for ActiveX/COM and CORBA**. This means that the CAPE-OPEN interface specification documents are expected to contain two

parts<sup>1</sup>, one describing the ActiveX specification, one describing the CORBA specification. We will further reference the interface specifications as **CO/Ax** and **CO/CORBA**.

The CO/Ax specifications will be expressed in MIDL, the Microsoft Interface Definitions Language. The CO/CORBA specifications will be expressed in IDL, OMG's Interface Definition Language. We will evaluate the **bridges between CO/Ax and CO/CORBA** as a part of the PATH work package.

### 1.3. Methods and Tools for the CAPE-OPEN Project

**We do not recommend a single modelling tool** for developing the object models. The available tools are not mature enough. As a consequence, the UML models will be prepared with different tools (Rational Rose, Select, P+, Visio etc.) until we can recommend the use of a single one. As another consequence, the *CO/Ax* and *CO/CORBA* specifications will not systematically be automatically generated from the UML models. We will **reassess the tools** after OMG's acceptance of UML and when good UML tools will be available. We will, in the mean time, establish an **electronic repository** for all UML models developed in the project

### 1.4. Work Process

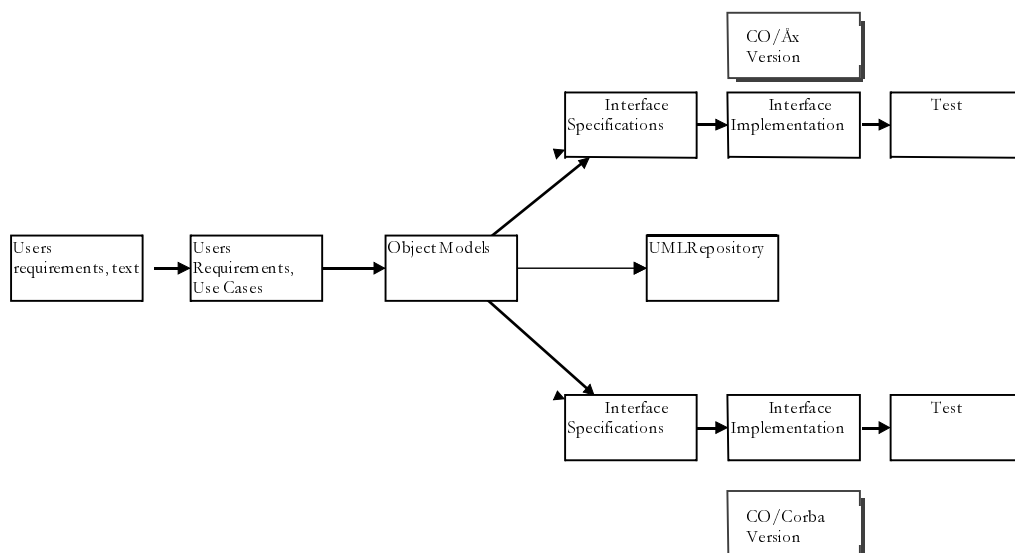
The work process that we follow for each sub-task of the three CAPE-OPEN technical work packages expected to deliver standard interface specifications and prototypes is presented in Table 1.

---

<sup>1</sup> In addition to the formal models in UML

Phase	Step	Goal	Means
ANALYSIS	Users requirements, text	Requirements in textual format	MS-Word
ANALYSIS	Users Requirements, Use Cases	Use Case models	Any drawing package or UML tool, Visio
DESIGN	Design Models	Sequence, Interface, Component Models using UML	Any drawing package or UML tool
DESIGN	UML Repository	Implement UML models in repository	Tool such as Microsoft Repository or Rose4.0
SPECS	CO/Ax Interface Specifications	Interface specifications in Microsoft's IDL	
SPECS	CO/CORBA Interface Specifications	Interface specifications in CORBA's IDL	
IMPLEMENT	CO/Ax Interface Implementation	Prototype MIDL implementation	Visual C++, Visual Basic Encapsulated FORTRAN
IMPLEMENT	CO/CORBA Interface Implementation	Prototype IDL implementation	C++ compiler, Encapsulated FORTRAN
TEST	Standalone Testing	Tested component	

**Table 1 :Work Process Phases**



**Figure 1: Summary of Work Process**

### 1.4.1. Analysis

The goal of the Analysis phase is to produce a structured set of users requirements in the form of a textual description and of a set of Use Cases.

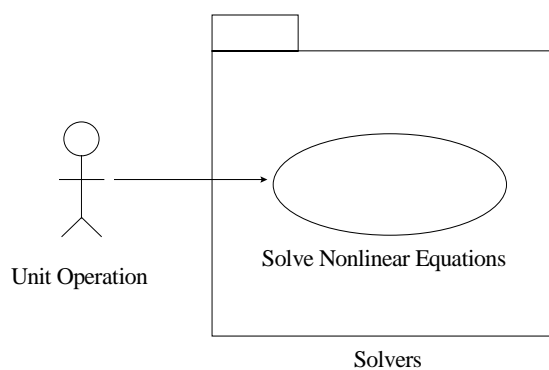
### 1.4.2. Users Requirements, Text

Express the users requirements in written form. This should be obtained by consensus of the Work package team. No tools are used at this stage. The result is an MS-Word document which presents and justifies the requirements. For the CAPE-OPEN Unit Operation interface, an example could read as follows (UO means Unit Operation):

... In this phase, the simulator evaluates each unit in turn. In the case of CAPE-OPEN UOs, it invokes the UO's "calc" method. The UO in turn invokes methods of the simulator, thermo and numerics objects, as required, during the course of its calculation....The basic requirement is for the external UO to be able to return to the host simulator either "OK", "failed", or "caution". The condition "caution" could apply if the external UO has an internal convergence loop that has not fully converged. This may not matter if, for example, the UO is in a flowsheet recycle loop and UO convergence is achieved by the time the flowsheet loop has converged. It would also be valuable to allow an option in the interface for an explanatory text string to be passed from the external UO to the host in the event of an error occurring...

### 1.4.3. Users Requirements, Use Cases

Build Use Case Models from the Users Requirements, using the UML notation. The Use Case models express the core requirements and provide a basis for testing a proposed design. These models should be obtained by consensus of the Work package team. No tools are required (i.e. the drawing capacities of MS-Word suffice) although the use of UML diagramming tools is encouraged. The result is an MS-Word document including the models. The following example is a Use Case for a Unit Operation requiring to solve its nonlinear equations:



#### Use case description:

A Unit Operation requires that the Solvers package solve a set of nonlinear equations. When the solution is complete the Unit Operation checks if the solution has converged. If the solution has converged then the Unit Operation uses the solution. If the solution has not converged then the Unit Operation must handle the exception.

**Exceptions:** Solution may not have converged

Figure 2: Use Case Example

#### 1.4.4. Design

The goal of the Design phase is to produce a set of design models using the UML notation suitable for the requirements expressed in the Analysis phase.

#### 1.4.5. Design Models using UML

Build Sequence, Interface, Component models from the Analysis set of documents and following the UML notation. These models are essential for the design of interface objects and will be used in a later stage for preparing the interface specifications. No tools are required (i.e. the drawing capacities of MS-Word suffice) although the use of UML diagramming tools is encouraged. The result is an MS-Word document including the models. The following example extends the Use Case « Solve Nonlinear Equations ».

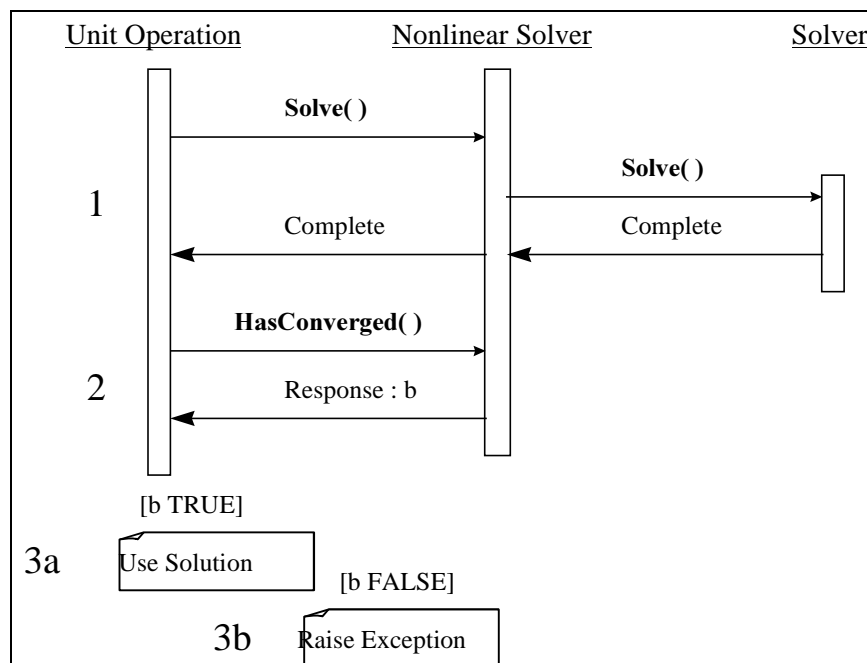


Figure 3: Sequence Model

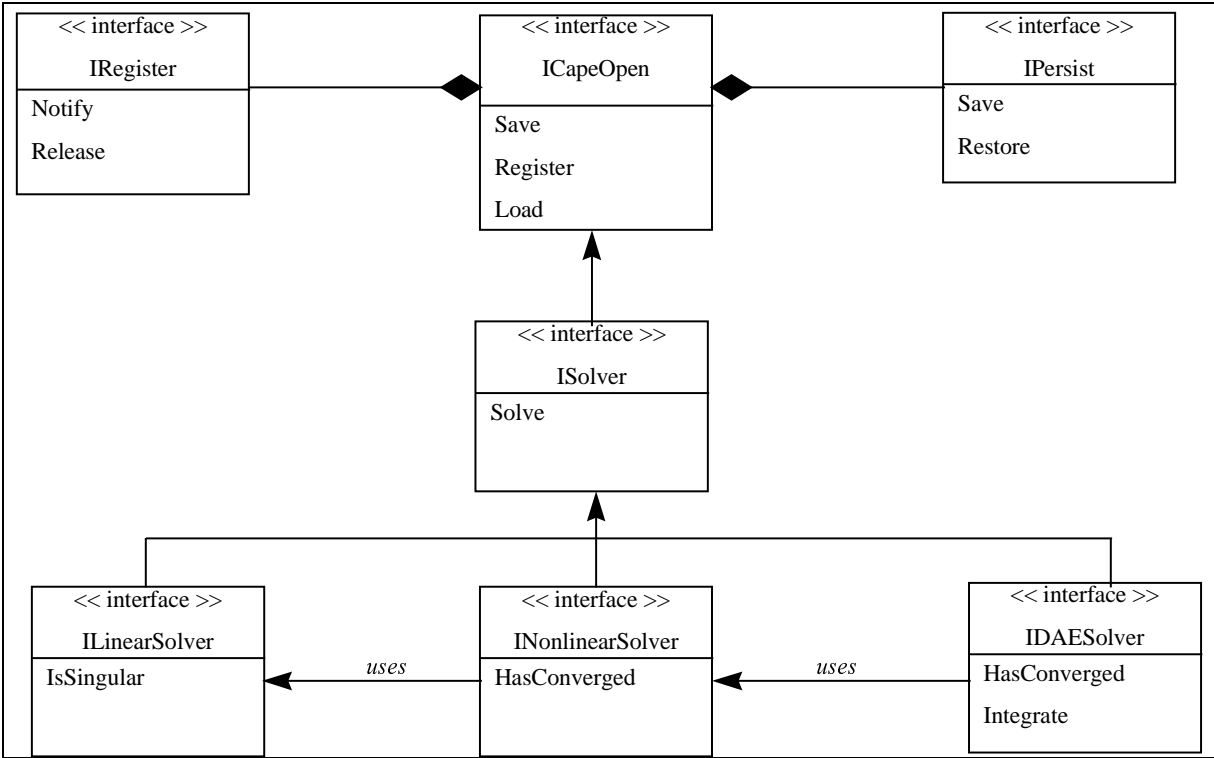


Figure 4: Interface Model

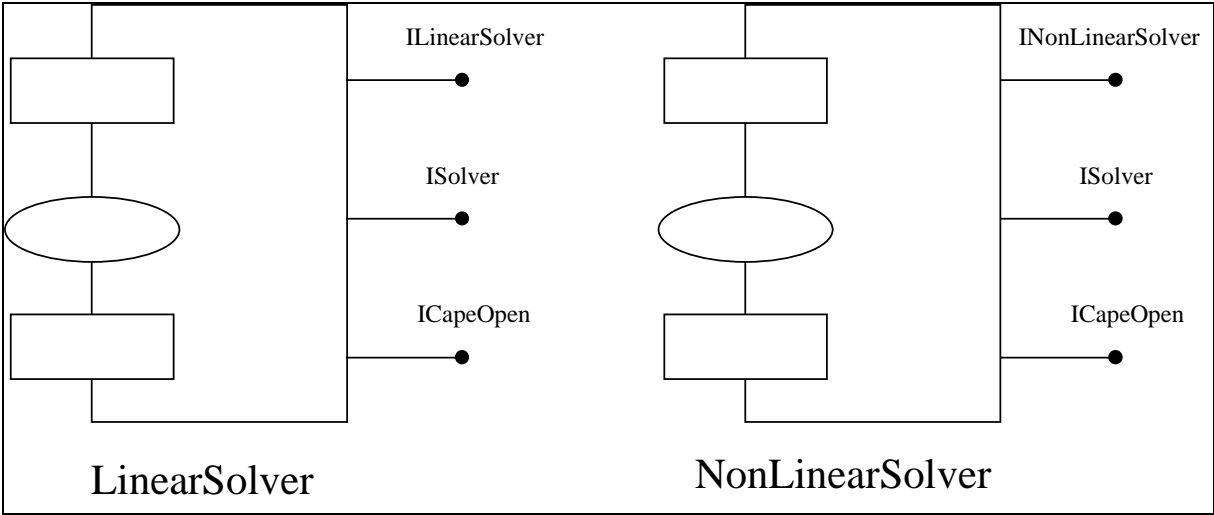


Figure 5: Component Model

#### 1.4.6. Implement UML Models in UML Repository

This phase is parallel to the mainstream activity of CAPE-OPEN. All UML models are implemented in an electronic repository which serves several purposes:

1. supply a single repository for all analysis and design models accessible to everyone;
2. prove that the UML model is consistent by introducing it in an environment with consistency checking capacities;
3. offer hands-on experience with UML tools;
4. prepare the automated generation of interface specifications.

The results of this phase are an electronic version of the UML models..

#### 1.4.7. Specifications

The goal of the Specifications phase is to produce the CAPE-OPEN Interface Specifications in MIDL (*CO/Ax*) and IDL (*CO/CORBA*) from the design models established in the Design phase.

#### 1.4.8. CO/Ax Interface Specifications in Microsoft's IDL

MIDL specifications for interface objects will be built from the design models developed in the design phase. The MIDL specifications contain:

1. object overview describing the functionalities and interfaces supported by the object;
2. interface descriptions that list all methods available within the interface;
3. detailed descriptions of the methods with input and output parameters and error codes.

No tools are recommended at this stage: a text editor suffices. However, automated generation of some of the MIDL specifications could be envisaged later on during the project, depending on the reassessment of UML compatible tools.

#### 1.4.9. CO/CORBA Interface Specifications in CORBA's IDL

IDL specifications for interface objects will be built from the design models developed in the design phase. The IDL specifications contain:

1. object overview describing the attributes and methods provided by the object;
2. detailed descriptions that list all methods available within the interface;

No tools are recommended at this stage; a text editor suffices. However, automated generation of some of the IDL specifications could be envisaged later on during the project, depending on the reassessment of UML compatible tools.

#### **1.4.10. Implementation**

The goal of the implementation phase is to produce prototype ActiveX /COM- and CORBA-compliant components using the interface specifications developed in the Specifications phase. Many of these components will encapsulate legacy code.

#### **1.4.11. CO/Ax Prototype MIDL Implementation and Encapsulation of Existing Code**

Prototype COM-compliant implementations will be produced by generating the interface code with the Microsoft IDL (MIDL) compiler and encapsulating other pieces of code within the interface. The generated interface code comprises:

1. a set of header files defining the interfaces;
2. code for proxies and stubs needed for local and remote methods invocation;
3. code for filling the COM object library.

Legacy code in FORTRAN or other language will have to be wrapped.

The use of the MIDL compiler is mandatory for this phase.

The result of this phase is a binary executable expected to work with ActiveX/COM and ready to be tested.

#### **1.4.12. CO/CORBA Prototype IDL Implementation and Encapsulation of Existing Code**

Prototype CORBA-compliant implementations will be produced by generating the interface code with an IDL compiler and encapsulating other pieces of code within the interface. The generation of interface code comprises

1. class declarations for the interface
2. additional code for client-server communication
3. skeletons to be used for the implementation of the interface

Legacy code in FORTRAN or other language will have to be wrapped. The use of an IDL compiler is mandatory for this phase. The result of this phase is a binary executable expected to work with an ORB (Object Request Broker) and ready to be tested.

#### 1.4.13. Test

The goal of the Test phase is to do a quick standalone test of the prototype components before submitting them to the VALidation team. The tests are expected to be done by the developers and should not need external review unless specific problems arise.

The result of this phase is a set of tested prototype components ready to be delivered to the VALidation team for integration in the validation environment. Both implementations will need such internal testing.

#### 1.4.14. An Iterative Process

We encourage an iterative approach where the different models and implementations are subject to progressive refinements like in the following spiral development process for standard interfaces to solvers (continuous arrows indicate strong dependencies, dashed arrows indicate that the destination node can take advantage of experience gained in the origin node):

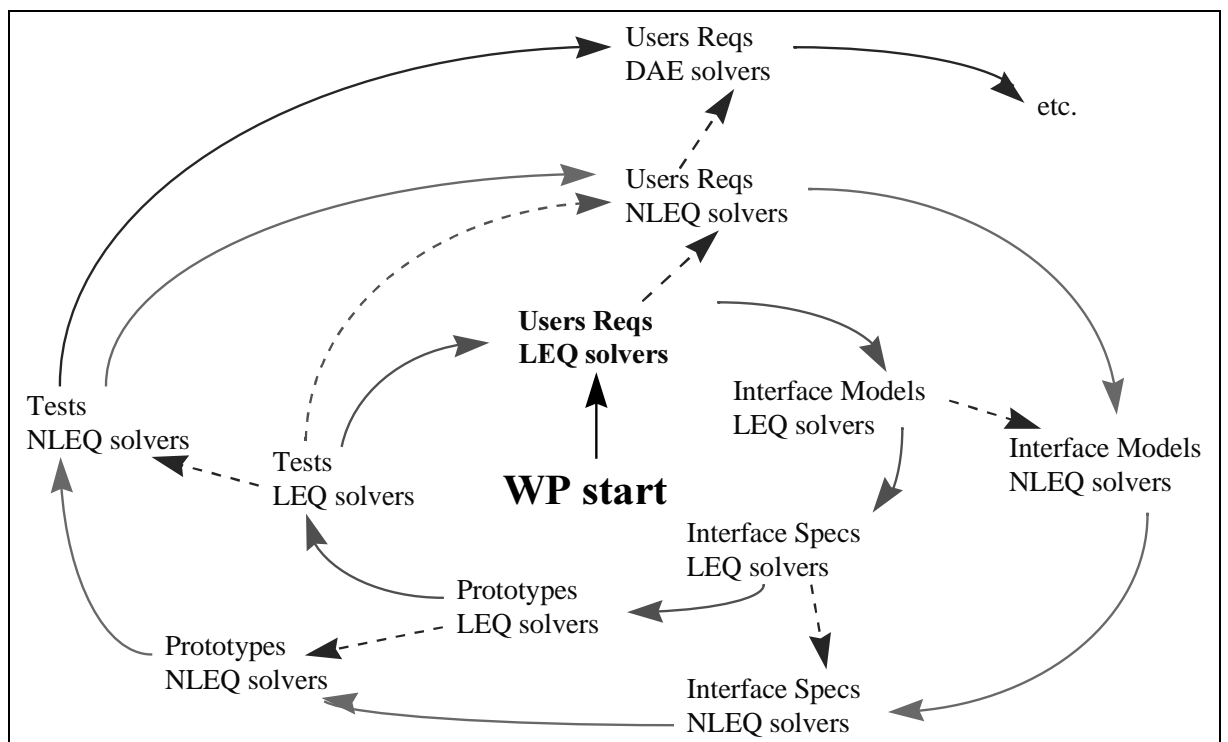


Figure 6: Iterative Process



## 2. Software Engineering Standards for CAPE-OPEN

### 2.1. Introduction

This document provides guidelines for developing CAPE-OPEN interface specifications. It addresses important software analysis and procedural issues and makes specific recommendations so that work package teams can create uniform release documents.

The areas considered in this document are as follows, together with the section number:

- (i) Naming : how names will be given to interfaces, methods, properties and arguments (2.2) ;
- (ii) Argument Ordering : order of arguments in interface methods (2.3);
- (iii) Data Types to be used in arguments of interface methods (2.4);
- (iv) Properties : how standard and non-standard properties are accessed (2.5);
- (v) Registry : how software components are categorized within the CAPE-OPEN section of the system registry (2.6);
- (vi) Architectural Aspects of Interfaces : how the UML interface diagrams are mapped into middleware specifications (2.7);
- (vii) Running components in-process, out-of-process (local and remote) (2.8);
- (viii) Identification : the minimal CAPE-OPEN standard interface definition (2.9);
- (ix) Error Handling (2.10);
- (x) Capabilities for which no common interfaces will be provided : persistence, graphical user interface (GUI), events handling (2.11);
- (xi) Version Control : how versions and releases of interfaces will be managed in CAPE-OPEN (2.12);
- (xii) The CAPE-OPEN Project repository : how versions and releases of interfaces will be stored for CAPE-OPEN (2.13).

In the sections that follow these guidelines, there is some review material which was produced by the Methods & Tools group. This review took the form of an overview rather than a detailed UML review of the documents produced to date. This was due to the recent nature of the currently available documents. This review material should guide the work teams to examine and critically review their own work, together with advice provided by the authors of this document.

## 2.2. Naming

For CAPE-OPEN interface specifications, the following guide will be used in order to provide naming conventions for the actual interfaces, for the methods which belong to interfaces, and for the arguments which belong to methods.

English is the base language to be used. Names should contain a clear indication of the purpose of the interface, method or argument and use mixed case words not allowing the underscore character ( `_` ) as a separator. There is no maximum length for names.

Care should be taken in the use of abbreviated words. The meaning of shortened words may not be obvious to non-native English speakers, or to people that are not involved in the software development process.

In naming interfaces, it should be noted that Microsoft IDL and CORBA IDL are case sensitive. However, CORBA IDL does not allow different names that differ only in case.

In this document the IDL used by Microsoft and the IDL used by CORBA are frequently encountered. They will be referred to as MIDL and CIDL in order to simplify the text of this document. When the term IDL is used alone, it implies a generic CAPE-OPEN interface language (IDL), rather than a particular implementation.

### 2.2.1. Interfaces

The CAPE-OPEN interfaces themselves will be prefixed as:

```
ICapeWorkPackageID
```

where `I` implies interface, `Cape` refers to the domain, and `WorkPackageID` is one of: `Unit`; `Thermo`; `Numeric`; or `Utility`. The first three of these terms refer to CAPE-OPEN work packages, while `Utility` refers to a base interface that all CAPE-OPEN work packages may utilise. Reasons for identifying the work package are for clarity and in order to scope the interface correctly. Following this prefix is the function name (e.g. `LinearSolver`)

As an example, for the numerical work package we might have an interface named

```
ICapeNumericLinearSolver
```

And following the same rules, for the Unit work package, we might have:

```
ICapeUnitSMUnitOperation
```

*Note:*

*Within CIDL we want to allow modules to be used to scope the interface definitions, e.g. `Cape::Numeric::ILinearSolver`. Type definitions can be used to resemble the naming convention as above in addition to supporting module scoping. In the case of a Unit Operation the interface name would be `ICapeUnitUnitOperation`, it should however be more readable if the UnitOperations interface will be named in other way, such as e.g. `SMUnitOperation` & `EOUnitOperation` for example.*

### 2.2.2. Attributes and Methods

Within an interface there will be methods to perform defined services of the interface. These methods can be classified as follows:

- (i) Properties/attributes access methods
- (ii) Object creation and object release and/or destruction
- (iii) Object enhancement or extension
- (iv) Specialist domain behaviours (such as pre-conditioning an iterative solver)

Methods used to provide standard base services of the CAPE-OPEN interfaces should use names that map naturally to their middleware counterparts if this is possible. For each of these groups, the methods names should follow a standard convention to aid someone who wishes to use the methods in the interface or in the future to extend the interface. The proposed naming convention for each of these is as follows:

<b>TYPE OF METHOD</b>	<b>METHOD NAME PREFIX</b>
Data obtaining	(none)*
Data providing	(none)*
Data inquiring	Query
Object creation	Create
Object release	Release
Object destruction	Destroy
Object enhancement	Add
Object restrict	Remove
Object duplication	Clone
Specialist behaviour	[none]

**Table 2: Methods Naming**

---

\* These are covered in the section on Properties later in the document.

The inclusion of the Add and Remove behaviours arise from reviewing the Use Cases produced so far in CAPE-OPEN. They are generally anticipated to be used in Interfaces which have a container (collection) like responsibility. So as an example in Visual Basic, we might have:

```
Dim element as new MyClassOfObjects
Dim MyCollection as new CollectionInterface
Dim elementName as String

elementName=»CapeOpenElement»
MyCollection.Add element, elementName
MyCollection.Remove elementName
```

where `CollectionInterface` is some class supporting the collection-like interface we have defined to include the Add and Remove methods.

So for example, to obtain the name of a unit operation (pre supposing some means of identification) we could have a method (attribute) such as `Name`. Other examples could be `AddPort`, `CreateLinearSolver`, `QueryDensity` and so on.

Prefixes for getting and setting properties are not included, because it is not common practice in COM and CORBA. Object release is used in COM and in CORBA it is supported as well, but not across address spaces. Object destroy is needed on the CORBA side since CORBA does not provide any other method to shut down an object by releasing it from the client.

The various work package teams have identified the need for object duplication, therefore we have defined a method of name `«Clone»`. The implications of providing object duplication methods are being studied by the Methods and Tools group.

*Note:*

*At this time, there are unresolved issues on the exact semantics and implementation of clone. It is anticipated that clone will relate to a shallow rather than deep copy.*

### 2.2.3. Arguments

Interface method arguments should be named so that they define clearly their purpose. In the full IDL produced, the type of the argument is also needed. This type information must reflect upon the types CAPE-OPEN supports in its interfaces as described in a following section. Attributes should begin with a lower case letter and multiple words should capitalise the initial letter of the second and subsequent words.

Examples are:

```
pressure, deltaPressure, numTraysInColumn
```

## 2.3. Argument Ordering

In methods within interfaces, the arguments to those methods should always have a consistent ordering to aid clear reading from the user. Read arguments should appear first and be marked with the [in] IDL attribute, followed by read-write arguments marked with the [in,out] attribute, then write-only arguments marked with the [out] attribute. Finally in MIDL the function return value marked with the [out,retval] attribute whereas in CIDL it is implied by the return type. For the generic CAPE-OPEN IDL the return value should be marked as [return]. So in MIDL we would have

```
HRESULT Methodname([in] type readarg1,...,
[in,out] type rwarg1,..., [out] type warg1,...,[out,retval]
type* retval)
```

*Note: In MIDL some other argument qualifiers, such as [optional], [defaultval].*

In CIDL we would have

```
type Methodname (in type readarg1, ...,
inout type rwarg1, ...,out type warg1,...);
```

and for methods that may raise exceptions (see error handling section):

```
type Methodname (in type readarg1, ...,
inout type rwarg1, ...,out type warg1,...)
raises (Exception1, ...);
```

**Reference:** «*Inside OLE*» Second Ed. Brockschmidt, K. Microsoft press. p. 77. (1995) and *OLE Automation Programmer's Reference*» Microsoft press. Chap. 7. (1996)

## 2.4. Data Types

The CAPE-OPEN project is adopting a standard set of data types that are handled by the CAPE-OPEN interfaces. These types will be independent of the component implementation language, and middleware, but they must be capable of being easily mapped to the middleware or implementation language types that are used. Developers of prototypes will define the mappings from these CAPE-OPEN types to middleware data types.

The most common set of data types to appear in interfaces are presented in the table below (the C++ mapping is given as an implementation example):

CAPE-OPEN	C++	ActiveX	CORBA
CapeLong	long	long	long
CapeDouble	double	double	double
CapeString	std::string	BSTR	string
CapeArrayT <sup>2</sup>	std::vector<T>	SAFEARRAY(T)	sequence<T>
CapeBoolean	bool	VARIANT_BOOL	boolean
CapeDate	std::string	DATE	string
CapeError	std::exception	HRESULT	exception
CapeVariant	void	VARIANT	any
CapeInterface	Interface*	VARIANT or IDispatch	Interface

**Table 3: CO Data Types**

The defined CAPE-OPEN data types will be used in IDL files with appropriate definitions for the specific middleware types. The `CapeInterface` type is used when passing other interfaces through argument lists of methods. So for example in Unit we may have a method on the `ICapeUnitSMUnitOperations` interface named `GetPorts`, which could return the `ICapeUnitPorts` interface and then this would be represented generically in IDL by

```
interface ICapeUnitSMUnitOperations
{
    ...
    GetPorts([return] CapeInterface portsInterface)
    ...
}
```

---

<sup>2</sup> Here T refers to any other CAPE-OPEN type, e.g. an object (`CapeUnitSMUnitOperation`) or a basic type such as `LONG`.

## 2.5. Properties as Attributes

Getting standard properties : `CODataType PropertyName()`

Setting standard properties : `void PropertyName(CODataType value)`

Getting non-standard properties :

`CODataType GetProperty(CapeString propertyName)`

Setting non-standard properties :

`void Property(CapeString propertyName, CODataType value)`

We therefore need to define standard properties in work packages (such as temperature in Thermo, or materialOfConstruction, etc. in Unit) and decide whether to factor the interfaces any further taking into account such standard properties.

*Technical Note: In the case of Unit Operations it is likely that most of the properties would be of the non-standard category. In addition, COM provides mechanisms for getting and setting public properties of a component, just by making use of the IDispatch interface which all our COM components should support. This could provide the functionality expected from GetProperty() plus some other additional features such as Setting properties or Querying them. Below is a code snippet showing this mechanism;*

```
BSTR SzMember=SysAllocString(»MyProp»);

HRESULT=pdisp->GetIDsOfNames(IID_NULL,
&szMember, 1, LOCAL_SYSTEM_DEFAULT, &dispid);

HRESULT=pdisp->Invoke(dispid, IID_NULL,
LOCAL_SYSTEM_DEFAULT, DISPATCH_PROPERTYGET,
&dispparamsNoArgs, pvarResult,NULL,NULL);
```

*The example above shows how to query an object property (GetIDsOfNames), for getting its dispid (dispatch identifier) (in case the property can not be found HRESULT will reveal that), and for obtaining its value (Invoke + DISPATCH\_PROPERTYGET). COM also provides other means for, at run time, obtain the addresses of property functions, so that the subsequent calls would be very fast.*

*The intention of this C++ example (extracted from «OLE Automation Programmer's Reference») is to reveal that COM supports part of the functionality CAPE-OPEN is expecting to provide, thus it could be interesting to make use of that in the maximum possible extent*

*Additionally, it is possible to obtain the overall list of properties and methods a component provides, by using similar sort of COM mechanisms.*

## 2.6. Registry

The middleware environment provides registration. In CORBA this feature is implemented with the Naming and Trader Services which may be used. COM/ActiveX will use the Windows Registry.

There will be a naming hierarchy of CAPE-OPEN components. The CAPE-OPEN interfaces are logically grouped in these categories.

```

CAPE
- UTILITY
- UNIT
- THERMO
  i.e. PROPERTYSYSTEM
  i.e. PROPERTYPACKAGE
  etc
- NUMERIC
  i.e. LINEARSOLVER
  i.e. NONLINEARSOLVER
  etc

```

These categories could be further refined by the work packages if they deem it valuable. This classification of the components into categories assists applications which can offer to users only those external components which claim to be of the right category. Obviously, further checks will be made to establish that the components do actually support the CAPE-OPEN standard interfaces. Note that this categorisation does not imply that any extra interfaces need to be developed and that also components can be in more than one category (or subcategory).

## 2.7. Architectural Aspects of Interfaces

Work packages will produce interface specifications that include the interface diagram and a generic CAPE-OPEN IDL. This keeps the CAPE-OPEN interfaces independent of the middleware implementation in MIDL or CIDL. The specification releases will also provide both MIDL and CIDL because both COM and CORBA prototypes are being implemented. The following is given as a guide to the production of specifications for the MIDL and CIDL versions.

### COM

The analysis version of the interface diagram will be prepared showing inheritance with a traditional inheritance notation. For the MIDL specification, the WP should write the MIDL assuming an implementation that is not done with custom interface inheritance. The actual design/implementation work can then decide whether to handle such inheritance with delegation/aggregation, containment techniques or whether to use some other alternative.

All COM interfaces will be dual interfaces, directly inherited from IDispatch. This is a common recommended COM practice that allows users of scripting languages, such as Visual Basic for Applications (VBA), to access properties and methods.

*Technical Note on early and late binding:*

*Dual interfaces allow clients of the component to make use of early and late binding. VTBL Binding, or more commonly, early-binding, is achieved when the client uses the server type library to, at compile time, get the addresses of the different interface functions. Thus, at run time, those functions are directly accessed through their respective addresses, without using IDispatch. A Visual Basic example is shown below*

```
` VTBL Binding is used here
Dim MyObject as new MyClassOfObject
MyObject.MyProperty = 25.0
```

*Late binding requires the use of IDispatch behaviour. Clients know methods and properties of an object by their names, nevertheless, interfaces derived from IDispatch (dispinterfaces) recognises those functions by their identifiers (DispIDs), that are internally stored. When a client uses late binding for calling a server method, the process involves two steps: 1) the client uses IDispatch::GetIDsOfNames for getting the DispID of the method (or property), 2) the client uses IDispatch::Invoke, passing in the DispID as an argument, for getting/setting the value of a property, or invoking a function.*

```
` Late Binding is used here
Dim MyObject as Object
Dim ProgID as String
Set MyObject = CreateObject(ProgID)
MyObject.MyProperty = 25.0
```

*Note: If it seems appropriate or necessary, the design/implementation team will go back to the analysis team to consult on alternative implementations.*

## **CORBA**

It is recommended that for CIDL, the analysis version of the interface diagram will also show an inheritance relationship with the traditional UML notation. However, in this instance, the interface diagram can map the inheritance relationships directly into the CIDL.

The objective of this approach is to use the different strengths of COM and CORBA implementations in the CAPE-OPEN prototyping work.

### **2.7.1. Factoring of Interfaces in COM**

One of the biggest issues to be considered when designing interfaces is factoring. Factoring is the process by which you decide how many interfaces to design, how many methods each of the interfaces have, and how many parameters each of the methods has. An entire book could be written on strategies for factoring interfaces, and there is much literature available on the topic of object-oriented analysis and design that is applicable. However, there are some basic rules you can use as you design your interfaces. These rules are described in the following sections.

### 2.7.2. Number of methods per interface

Experience has shown that interfaces with fewer methods are better. Interfaces with many methods that are intended to be implemented by a large number of objects usually end up having most of the methods return E\_NOTIMPL.

Fewer methods, however, means more interfaces. The greater the number of interfaces, the greater the number of times a client might be forced to call QueryInterface just to execute a simple task. The general rule is if two sets of functions are independent, that is, you expect either to be implemented without the other, the sets of functions should be contained in different interfaces. In most cases, if you are tempted to have a «capability flag» to indicate whether some functions are implemented, you should separate interfaces and take advantage of QueryInterface instead.

Also, try to eliminate options no one will want to use or implement. Often, interface designers try to think up every conceivable use for their interface and thus add additional methods to satisfy these «potential» users. Do not fall into this trap. Instead, focus on your primary users and design the interface so that it fits their needs. If a customer needs additional, special functionality, you can provide that functionality in another interface.

### 2.7.3. Number of parameters per method

When factoring your design, think about «round trips.» Each call to an interface method involves at least one «round trip,» potentially across a process or machine boundary. Therefore, it is «cheaper» to send everything needed to execute a call with one method than to have to call two methods with half as many parameters. However, it is sometimes possible to reduce the amount of data marshalled by doing just the opposite: have one «setup» method and then let users call the various «worker» methods without having to supply the «setup» information each time.

Also, try to limit the number of parameters a method contains. Having to call a method that takes more than five or six parameters is bothersome to many programmers (and you may start reaching the bounds of what the programmer's compiler can handle).

### 2.7.4. Note on Preparing UML Interface diagrams

Interface diagrams used by CAPE-OPEN are an extension of the official UML. However, they are consistent with the thrust of UML and have been found to be a useful tool for moving from use cases to interface specifications. They are required as part of the interface specification documentation. The following comments are provided for guidance of work package teams.

The aggregation relation within the interface diagrams seemed to confuse some project members. The intention behind drawing an interface aggregation is not to specify in MIDL an interface that consists of some other interface, but to show the actual aggregation relationship between the objects that are modelled. MIDL definitions will not use aggregation relationships or custom interface inheritance implicitly, whereas the CIDL can.

## 2.8. Running components in-process, out-of-process (local and remote)

There should be no implications for the design of the interfaces at the design stage as regards the memory space in which the component implementing the interface should run. It should be noted here that in-process server components are much more efficient than out-of-process servers. Thus, whenever the overhead imposed by the calling represents a high percentage of the overall calculation time (e.g. in a simple Unit Operation, or for flashing a Material), an in-process server would be preferred.

## 2.9. Identification

All CAPE-OPEN compliant components will provide an ICapeIdentification interface. This interface exposes descriptive information for each component as an attribute. Thus in CAPE-OPEN generic IDL form this interface is defined as:

```
ICapeIdentification
{
    CapeString componentName;
    CapeString componentDescription;
}
```

## 2.10. Error Handling

There are different mechanisms to handle errors within both middleware architectures. COM uses HRESULT return types, whereas CORBA uses an exception mechanism similar to C++. For CAPE-OPEN at least two types of errors should be distinguished:

### 1. Errors.

Report error when the contract between caller and callee was violated in a manner that the calculation could not be finished. Those errors require user interaction or notification.

### 2. Warnings.

Report warning when the calculation is performed, but some problem occurs that may influence the result and should be reported to the user, e.g. violating the valid range for some physical property correlation.

The authors of interfaces will define the contracts for each methods and property access. A violation of the contract, such as passing a Word document object to a physical property calculation, should result in an exception or the return of an error HRESULT, as appropriate for the platform. An exception or error return would typically result in a break in the program flow in the calling routine. Handling an error or exception is typically much more expensive than a successful function return so methods should be defined to avoid using exceptions and error result to communicate conditions that can do not justify a break in program flow. For example, if it is anticipated that the caller would be able to take corrective action for an extrapolation beyond the recommended range for a correlation, extrapolation should be communicated by a different mechanism. Standard error codes or exceptions should be used in preference to defining equivalent CAPE-OPEN specific error codes or exceptions. The documentation of each method should list anticipated error codes or exceptions.

If it is desired to allow a routine to return human-readable diagnostic messages to the calling routine, this method should be defined by including an argument for passing an object implementing ICapeUtilityMessageLog to the method. The caller should be allowed to indicate that it is not interested in collecting any warnings from the called routine. ICapeUtilityMessageLog will have one method, AddMessage that will allow the calling routine to specify a message code , a textual description of the message and source, a URL of a help or html file and a help context. A proposed form of the call in MIDL is:

```
HRESULT AddMessage([in] HRESULT msgCode,[in] BSTR
description, [in] BSTR source,[in] BSTR
url,[in,defaultval=0] long context);
```

If an error or exception occurs, the called routine should not attempt to call AddMessage. The called routine should not call AddMessage to indicate normal completion.

In CIDL we have:

```
void AddMessage (in long msgCode,in string
description,in string source,in string url,
in long context)raises (MessageLogFullException);
```

## 2.11. Capabilities for which no common interfaces will be provided

### 2.11.1. Persistence

Persistence interfaces will be provided by each of the interfaces where appropriate. COM or CORBA Persistent Interfaces will be used as a standard, and according to Microsoft and CORBA guidelines. CORBA defines means outside interface definitions to ensure persistence. Usually, persistence is not modelled within an interface but deferred to distinct concepts (the persistence layer) for more flexibility. The result of using these persistent interfaces causes the underlying state of the implementation to be saved. Not every component needs to support a persistent interface, only where dictated by a Use Case.

### 2.11.2. Graphical UI

It is not mandatory that a user interface be provided for each component. In many cases, the client of the component will be able to build and display a graphic user interface for the component. In the COM architecture this can be accomplished by common OLE techniques. The preferred approach within CORBA is to have a server without an interface and a separate interface on the client side because:

- This provides more flexibility for using the server's services
- Less data traffic in the remote case
- Implementing a user interface on the server side would make the use of this interface impossible if a remote client does supports a different UI system

*Note: In those cases in which a component provides its own GUI, a triggering mechanism for the client to launch a component user interface should represent a minimum effort for being included as part of the component interface, and would potentially facilitate the increased use of that component*

## Events

«Events» is a common Microsoft way of providing bi-directional communication between client and server components (that, therefore, no longer need to be considered pure client or servers). In this sense, events refer to «connectable objects», e.g. objects that support both incoming and outgoing interfaces. An outgoing interface is a connection point in the server that may be used by the client. By using these connection points, the server connects with a certain part of the client (called an events sink). Events sources appear in MIDL by special keywords.

As a matter of example: when, in a windowing environment, the user clicks on a button, this action (event), performed on a specific component, is notified to the client (the application that uses the button) that handles the event by taking appropriate actions (i.e. displays a message box).

Tasks, such as, controlling the time a given component has for performing its calculations can be accomplished by other means, as described in the appendix

Events are not relevant to CAPE-OPEN at this time.

## 2.12. Version Control

### Versioning

The following practice will be used. Version numbers will consist of three digits separated by dots, e.g. 0.5.8. The version numbers will be increased as follows:

- last digit: the last digit will be increased according to the development releases by the organisation responsible for the prototype and for describing the actual interface implementation. For a release, such as 0.5.8, the particular organisation will use its own release scheme for the actual software, e.g. Version 0.5.8 Build 3.
- middle digit: this version number is the responsibility of the work package leader. Each new release on this level will be given to validation as described below, in the beginning it will be stand-alone validation, towards the end of the project, and it will be integration testing.
- first digit: increasing the first digit is in the responsibility of the whole project, and it is assigned to the project leader. That means, it cannot be changed, unless integration testing was successful.

When a release from a work package is published, it must include the following as a minimum:

1. UML descriptions (sequence diagrams, interface diagrams, etc.)

2. Documentation (e.g. help files). The documentation could be provided in HTML format and includes the generic CAPE-OPEN IDL form of the interfaces.
3. CIDL/MIDL definitions for COM AND CORBA.
4. Binary of any simple component prototype implementing the interface
5. Source code for a simple test harness or other supporting code

All these items should be clearly identified with the version number of the release to ensure consistency. The documentation should mention the releases of other prototypes that the release is designed to operate with for backward compatibility.

### **2.13. The repository**

As part of the CAPE-OPEN process, a separate electronic repository of CAPE-OPEN releases will be kept at QuantiSci Limited on their public BSCW server, with access to all project members. This repository will store the released information, including Use Cases, UML diagrams, Interface Specification documents, and IDL files and help documentation. These will be made navigable by utilising HTML wherever possible.

## 2.14. Appendix: Some rationale

### 2.14.1. Data Types

The data types to be used are shown in the Table shown in page 8.

The reason for not including the OPEN part of CAPE-OPEN is that these standards will live on past the project and therefore the OPEN part is redundant. Also, CO is used by Microsoft (actually as Co, but it is easy to forget the capital letter) to indicate COM function calls.

Cloning . . . need some comment here

Argument Ordering. Although differences between the MIDL and CIDL are small, the actual representation of return values has not been specified. It was considered whether or not to write them down as a special argument role (such as [return]), which would invent a new language and several problems with it. This has to be further discussed.

There was a long discussion on the usage of inheritance within the UML interface diagrams. The current outcome can be summarised as follows:

We want to use each middleware's advantages to the best extent possible, but still keep the UML diagrams for the analysis phase consistent with common object-oriented modelling techniques. We propose to map these UML descriptions to whatever fits best into each technology. That means that the COM implementations will use aggregation/containment to represent the inheritance relations. The CORBA developers will be able to directly map the interface definitions to their IDL and can thus make direct use of the inheritance relationship.

The COM developers mentioned the fragile base class problem, which has to be investigated. The CORBA development will be continued to explore this problem.

Defining CAPE-OPEN data types. Int and short are not included because all CAPE-OPEN will be capable of handling 32bit integers (long) as a single word anyway. The same argumentation applies to float, which has also been left out. When wrapping legacy code, the developer is responsible for converting to the appropriate types.

As opposed to COM, the CORBA strings are not capable of handling UNICODE, yet.

An issue was raised to define a type that can refer to any CAPE-OPEN interface, such as CO\_INTERFACE, but no decision has been made. A possible choice would be to make it refer to IDispatch in COM and to CORBA::Object in the CORBA world.

### 2.14.2. Notes on Interruptions and Progress Monitoring

The implementations of some methods can take substantial time and may need to provide a method to report their progress and to check for an interruption request. The interface designer should accomplish this by passing an object in the argument list to the method that acts as a progress monitor. This object should support the IProgressNotify interface (on COM platforms) and the ICapeProgressNotify interface. The IProgressNotify interface has

one method `OnProgress` that takes a numeric indication of the completeness and returns a value that indicates whether the calculation should continue.

### 2.14.3. Notes on Early and Late Binding: Relevance for CAPE-OPEN

An example that can illustrate, in a very simple manner, how early and late binding condition how properties and methods are accessed is the way in which the new release of Visual Basic (VB5) helps the user in declaring and using objects in his applications.

VB5 has recently added a new feature «**Auto List Members**». When a user has declared an object reference and is writing code to get access that object's properties or member functions, VB5 automatically displays them in a list. This allows the programmer to speed up writing code, since VB5 automatically completes the name of the member function or property. When the programmer starts writing the arguments of that member function he also gets a displayed pop up window showing the prototype of this function with the argument names and types.

In the following picture the programmer has declared the object `ps` as a `ProcessStream`. When he types the dot, VB5 displays the list of properties and member functions of the object `ProcessStream`.

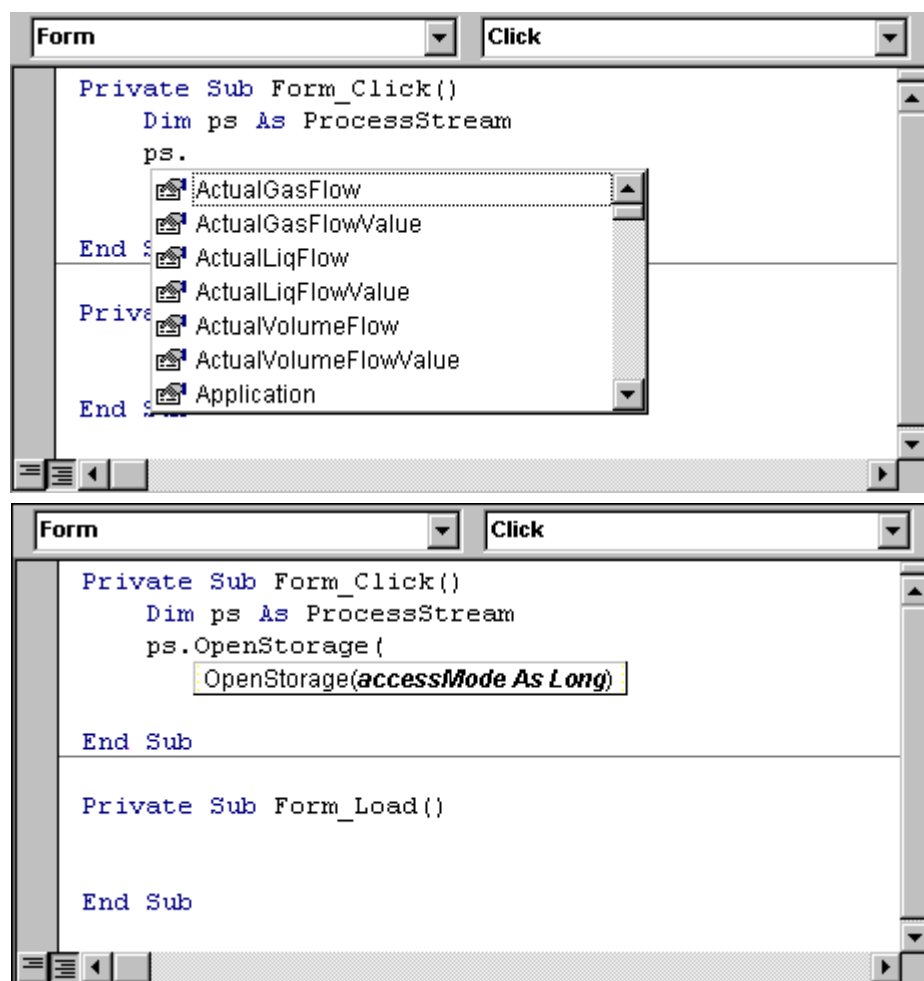


Figure 7: Early and late binding

In the second picture, VB5 shows the prototype of `ProcessStream`'s member function `OpenStorage`, so that, the programmer can pass the correct parameter type.

The above declaration (e.g. `Dim ps As ProcessStream`) forces VB5 to treat «ps» as a `ProcessStream` at compile time. This is what is known in OLE/COM technology as «Early Binding». If the type of the object is not known at compile time, it can still be declared as «Object», meaning that object's behaviour will be determined at run time. This is known as «Late Binding». One of the most remarkable differences between Early Binding and Late Binding is that in the case in which the server is an In-Process server calls are very fast.

```
Dim ps As Object
Debug.Print ps.ActualGasFlowValue
```

By using the above declaration (Late Binding) VB5 does not know anything about «ps» object's property `ActualGasFlowValue` or about its type, at compile time. Since VB objects are derived from OLE/COM `IDispatch`, they are self-described. At run time, VB5 will use their self-describing functionality to access their values and/or behaviour.

Finally, in the case of functions having as parameters library-defined constant values, VB helps to set the values since it displays a list with all possible values.

Of course, Automation functionality is language independent, so this discussion is not restricted to VB.