CAPE-OPEN

Delivering the power of component software
and open standard interfaces
in Computer-Aided Process Engineering

Open Interface Specification:

Persistence Common Interface

CO LaN

www.colan.org

ARCHIVAL INFORMATION

Filename	Persistence Common Interface.doc
Authors	CO-LaN consortium
Status	Public
Date	August 2003
Version	version 2
Number of pages	30
Versioning	version 2, reviewed by Jean-Pierre Belaud, August 2003
	version 1, Methods & Tools group, November 2001
Additional material	
Web location	www.colan.org
Implementation specifications version	CAPE-OPENv1-0-0.idl (CORBA) CAPE-OPENv1-0-0.zip and CAPE-OPENv1-0-0.tlb (COM)
Comments	

IMPORTANT NOTICES

SUMMARY

Most simulation environments allow the possibility to store at any moment the state of a simulation case, in order to be able to restore it at any time in the future. In the CAPE-OPEN distributed environment, where different pieces of the simulation may be implemented by different vendors, there must be a standard mechanism to provide this feature.

ACKNOWLEDGEMENTS

CONTENTS

LIST OF FIGURES

1. Introduction

Most simulation environments allow the possibility to store at any moment the state of a simulation case, in order to be able to restore it at any time in the future. In the CAPE-OPEN distributed environment, where different pieces of the simulation may be implemented by different vendors, there must be a standard mechanism to provide this feature.

The UNIT interfaces provided two methods for storing and restoring the state of each Unit Operation inserted into a flowsheet. Upon this experience, it became evident that:

- Units must store their state in order to keep the configuration set by the user. This configuration may have been performed through the Simulator mechanisms to update the unit's public parameters, or through the custom Graphical User Interface that the Units may provide.

- Any type of CAPE-OPEN PMC might benefit of this functionality. PMCs such as Property Packages, should be allowed to be configured from the simulation environment, in order to allow the user to deploy all the PMC functionalities from the same entry point, the simulator.

- Originally, the COM mechanisms for implementing persistence were not regarded satisfactory at all and no analogue was available in the CORBA world. As it is becoming evident that an integration of CORBA and COM components is far more difficult than expected, this requirement can be weakened. Hence, the mechanism proposes to use the standard COM implementation on Windows-based machines, whereas a similar model for CORBA will be proposed.

2. Requirements

2.1 Textual requirements

Based on the introduction chapter, the requirements to be fulfilled consist of:

- A common mechanism that any CAPE-OPEN PMC may use in order to persist its state.
- The PMC and the Simulator Executive may be located on different machines. Persistence based on storage of local files may not be feasible due to security limitations.
- A practical implementation mechanism must be chosen. The one chosen by the original Unit specification presented major limitations, especially when the PMC were implemented in high level languages such as VB. Although the standard allowed two alternative mechanisms, both presented shortcomings:
  - *The simulator passes the Unit the name of the file where the state must be persisted.* This implementation was not convenient at all, since it created a different file for each Unit Operation instance. Apart from this undesirable multiplicity of files for a single case, moving them to a different storage device or path was unfeasible, since the file location became fixed.
  - *The simulator passes the Unit a storage interface where the Unit must write its data.* Unfortunately, VB did not support the access to this type of interfaces.

2.2 Use cases

2.2.1 Actors

- ❑ **Flowsheet User.** The person who uses an existing flowsheet. This person will put new data into the flowsheet, rather than change the structure of the flowsheet.
- ❑ **Simulator Executive.** The part of a simulator whose job it is to create, or load, a previously stored flowsheet, solve it and display the results.
- ❑ **CAPE-OPEN PMC.** Any type of PMC inserted in a simulation flowsheet.

Use Cases Categories

- ❑ **General Purpose Use Cases:** Since persistence applies to any type of PMC, these use cases are applicable to Unit Operations, Property Package, Solvers, ...
- ❑ **Specific Use Cases:** The specific Use Cases for each particular type of PMC should be specified in each Business Interface specification. This document only shows two template Uses Cases to be customized on each particular case.

Use Cases Priorities

- ❑ **High.** Essential functionality. Functionality without which usability or performance might be seriously compromised

- **Low.** Desirable functionality that will improve performance. If this Use Case is not met, usability or acceptance can decrease.

2.2.2 List of Use Cases

- UC-001: Save Flowsheet
- UC-002: Save PMC
- UC-003: Retrieve Flowsheet
- UC-004: Restore PMC

2.2.3 Use Cases maps

2.2.4 Use Cases

UC-001 SAVE FLOWSHEET

Actors: <Flowsheet user>

Priority: <High>

Classification: <General Purpose Use Cases>

Pre-conditions:

<A PMC has been inserted into a flowsheet. A particular case would be UNIT's [Add Unit to Flowsheet] Use Case>

Flow of events:

Basic Path:

The Flowsheet User requests the Simulator Executive to save the Flowsheet. The Simulator Executive saves its native information in its own native format (this is outside CAPE-OPEN).

The Simulator Executive asks the CAPE-OPEN PMC to save its data in a persistence structure provided by the former. The CAPE-OPEN PMC uses the [Save PMC] Use Case to save its data. Each instance of each CAPE-OPEN PMC is provided an independent persistence structure, so that each PMC instance can be individually retrieve its data in the future. The Simulator Executive will integrate the persistence structure in the native simulator's mechanism for storing simulation cases.

Note 1: In the case of Units, Connectivity data must be persisted by the simulator, including port names.

Post-conditions:

<PMC's state data has been successfully stored with the simulation case>

Errors:

<Failure saving>

Uses:

[Save PMC]

Extends:

UC-002 Save PMC

Actors: <Simulator Executive>

Priority: <High>

Classification: <Specific Use Cases>

Pre-conditions:

The PMC supports persisting its state

Flow of events:

Basic Path:

The PMC information, including Specific Data and Results if any, is saved on the persistence structure provided by the Simulator Executive.

Post-conditions:

<Save succeeded>

Errors:

<Fails to save>

Uses:

Extends:

UC-003 Retrieve Flowsheet

Actors: <Flowsheet User>

Priority: <High>

Classification: <General Purpose Use Cases>

Pre-conditions:

<The simulation case to be retrieved exists>

Flow of events:

Basic Path:

The Flowsheet User asks the Simulator Executive to retrieve a previously stored flowsheet. The Simulator Executive retrieves the native Flowsheet data, such as stream connections, in its usual way. For each CAPE-OPEN PMC in the flowsheet, it recovers the PMC type together with the persistence structure at which the PMC data was stored. A new instance of this PMC type is created. It requests each CAPE-OPEN PMC to connect to the simulator's CAPE-OPEN objects, such as connecting their ports to the simulators streams in the case of unit. If the creation is successful, it asks the PMC to retrieve its data from its corresponding persistence structure. It does this using the [Restore PMC] Use Case. If the Flowsheet Unit fails to restore, the Flowsheet User is notified.

Post-conditions:

<PMC has been appropriately created and initialised>

<PMC has been connected to the appropriate simulator objects>

<PMC has recovered its state>

Errors:

<Failed to retrieve the flowsheet>

<Failed to restore the PMC data>

<Failure to connect to the simulator objects>

Uses:

[Add PMC to Flowsheet]

[Specify PMC Connections to Simulator Objects]

[Restore PMC]

Extends:

UC-004 Restore PMC

Actors: <Simulator Executive>

Priority: <High>

Classification: <Specific Use Cases>

Pre-conditions:

<[Add PMC to Flowsheet] has been used and successfully passed. A particular case would be UNIT's [Add Unit to Flowsheet] Use Case >

Flow of events:

Basic Path:

The PMC restores its specific data from the specified persistence structure.

Post-conditions:

Errors:

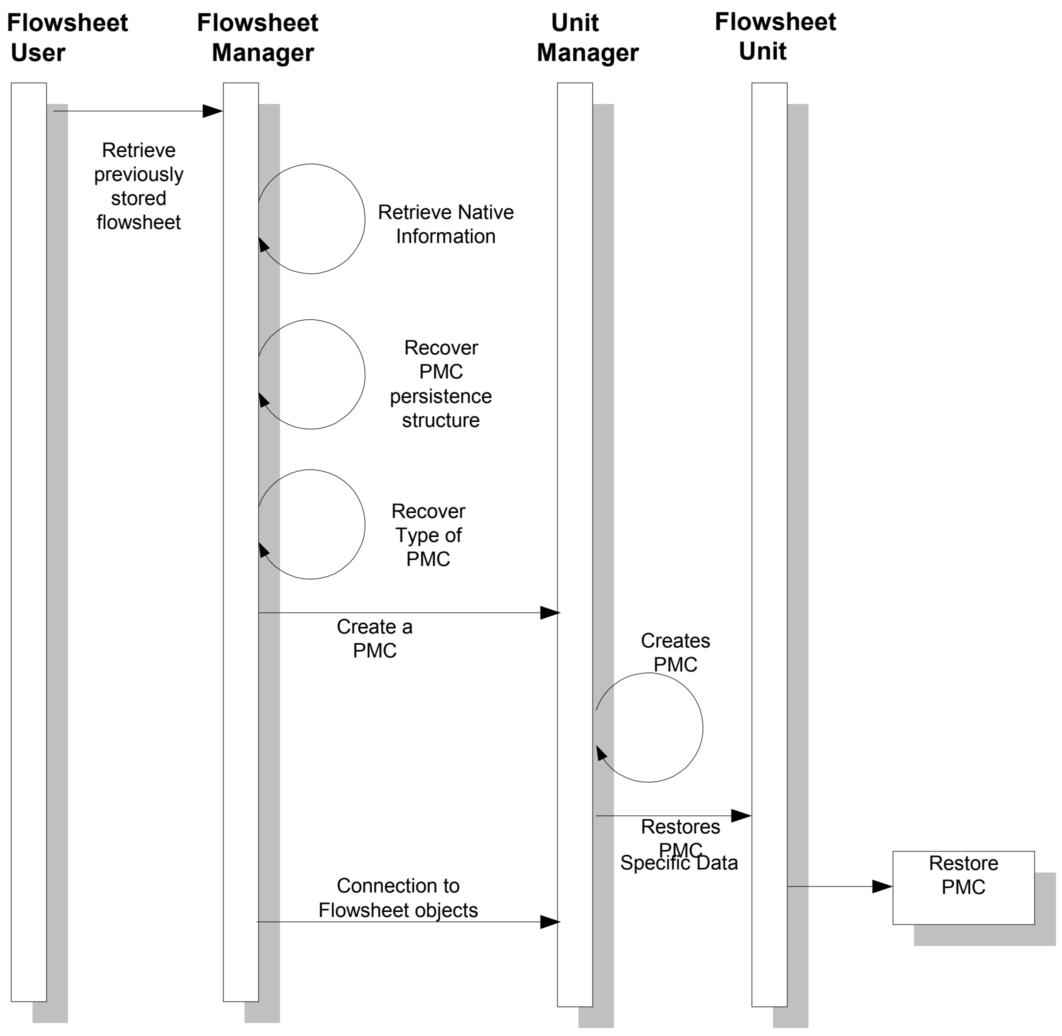
<Bad data>

Uses:

Extends:

2.3 Sequence diagrams

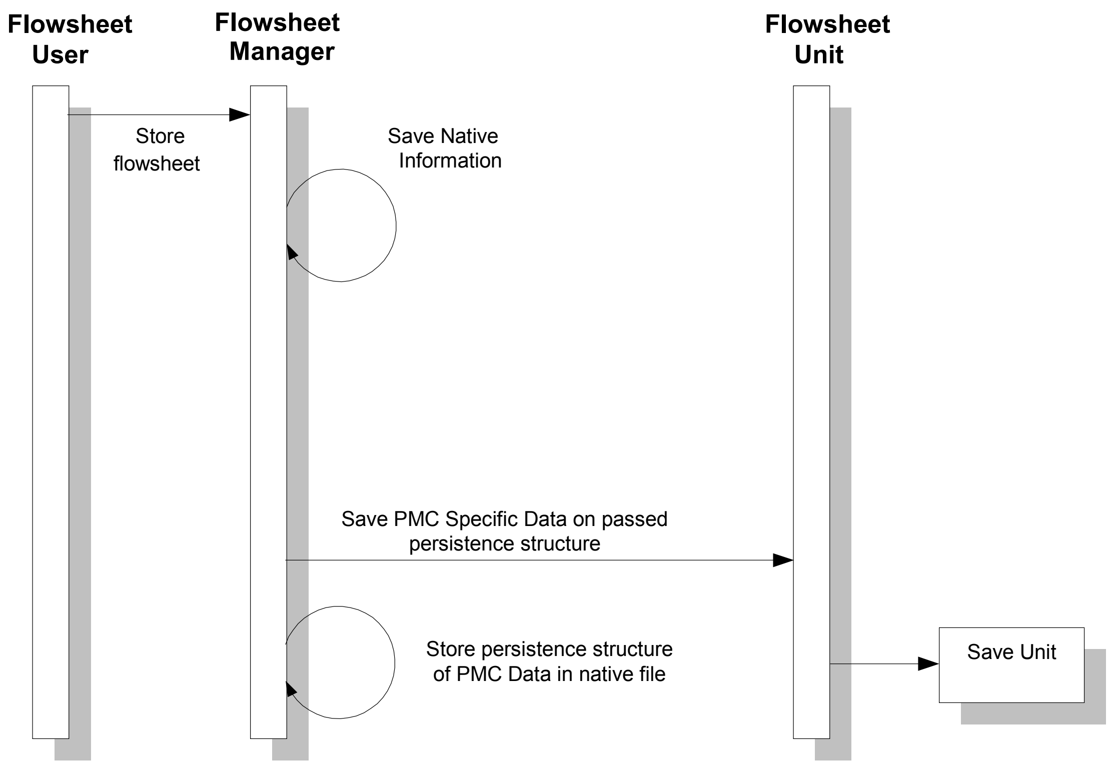
SQ-001 RETRIEVE FLOWSHEET



Flowsheet User
Flowsheet Manager
Unit Manager
Flowsheet Unit
Retrieve previously stored flowsheet
Retrieve Native Information
Recover PMC persistence structure
Recover Type of PMC
Create a PMC
Creates PMC
Restores PMC Specific Data
Restore PMC
Connection to Flowsheet objects

Figure 1 Retrieve flowsheet

SQ-002 SAVE FLOWSHEET



Flowsheet User
Flowsheet Manager
Flowsheet Unit
Store flowsheet
Save Native Information
Save PMC Specific Data on passed persistence structure
Store persistence structure of PMC Data in native file
Save Unit

Figure 2 Save flowsheet

3. Analysis and Design

3.1 Overview

3.2 Sequence diagrams

3.3 Interface diagrams

IN- 001 INTERFACE DIAGRAM

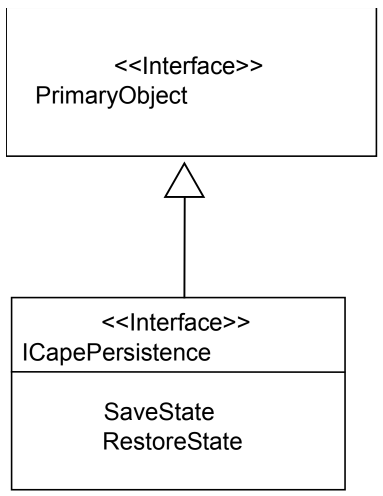


Figure 3 Interface diagram

3.4 State diagrams

<u>ST- 001 PERSISTENCE STATE DIAGRAM</u>

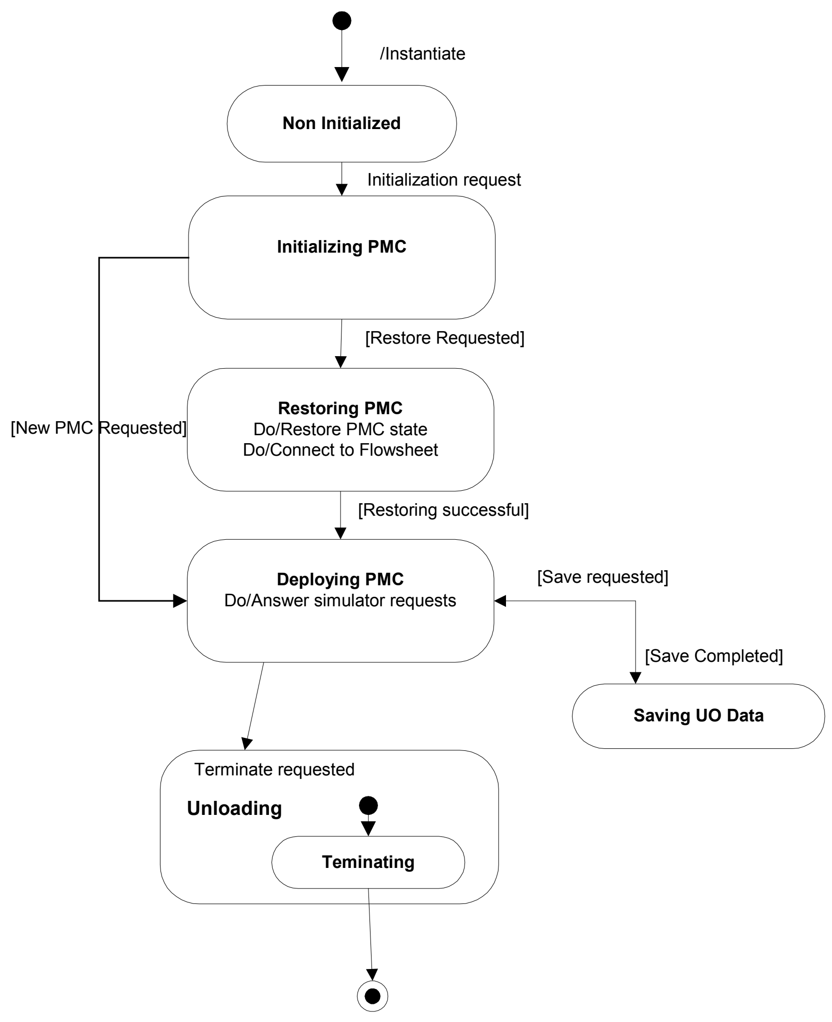


Figure 4 Persitence state diagram

3.5 Other diagrams

CO-001 COMPONENT DIAGRAM

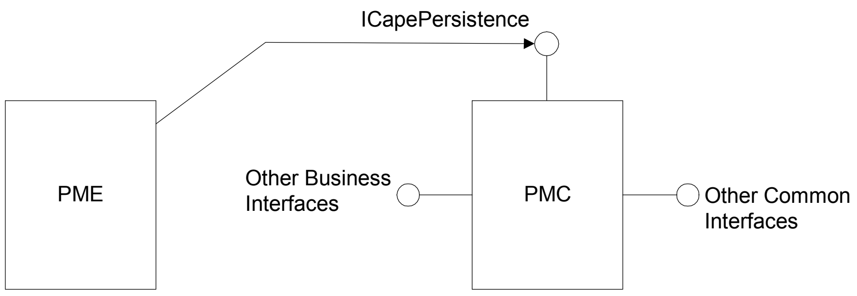


Figure 5 Persitence component diagram

3.6 Interfaces descriptions

3.7 Scenarios

4. Interface Specifications

4.1 COM Persistence Mechanisms

Based on the existing requirements, it was decided that rather that defining CAPE-OPEN specific persistence interfaces and data structure, it was best to reuse the COM Persistent native mechanisms. See below an overview of this specification. Since methods *InitNew* and *Load* are considered to be part of the COM middleware life-cycle, they will always be the first methods called on each PMC (obviously, after their object constructors). This means that even *Initialize* method (belonging to common or business interfaces) will be called after invoking the methods *InitNew* and *Load*

4.1.1 COM Persistence Interfaces

Objects that have a persistent state of any kind must implement at least one **IPersist*** interface, and preferably multiple interfaces, in order to provide the container with the most flexible choice of how it wishes to save a control's state.

If a control has any persistent state whatsoever, it must, as a minimum, implement either **IPersistStream** or **IPersistStreamInit** (the two are mutually exclusive and shouldn't be implemented together for the most part). The latter is used when a control wishes to know when it is created new as opposed to reloaded from an existing persistent state (**IPersistStream** does not have the created new capability). The existence of either interface indicates that the control can save and load its persistent state into a stream, that is, an instance of **IStream**.

Beyond these two stream-based interfaces, the **IPersist*** interfaces listed in the following table can be optionally provided in order to support persistence to locations other than an expandable **IStream**.

{PRIVATE}Interface	**Usage**
IPersistMemory	The object can save and load its state into a fixed-length sequential byte array (in memory).
IPersistStorage	The object can save and load its state into an IStorage instance. Controls that wish to be marked Insertable as other compound document objects (for insertion into non-control aware containers) must support this interface.
IPersistPropertyBag	The object can save and load its state as individual properties written to IPropertyBag which the container implements. This is used for Save As Text functionality in some containers.
IPersistMoniker	The object can save and load its state to a location named by a moniker. The control calls IMoniker::BindToStorage to retrieve the storage interface it requires, such as IStorage, IStream, ILockBytes, IDataObject, etc.

With the exception of **IPersistStream[Init]::GetSizeMax** and **IPersistMemory::GetSizeMax**, all methods of each interface must be fully implemented.

See below the specification of IPersistStreamInit. More information on any of these interfaces may be found at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/cmi_n2p_02b1.asp

4.1.2 IPersistStreamInit

The **IPersistStreamInit** interface is defined as a replacement for IPersistStream in order to add an initialization method, **InitNew**. This interface is not derived from **IPersistStream**; it is mutually exclusive with **IPersistStream**. An object chooses to support only one of the two interfaces, based on whether it requires the **InitNew** method. Otherwise, the signatures and semantics of the other methods are the same as the corresponding methods of **IPersistStream**, except as described below.

Methods

{PRIVATE}IPersist **Method**	**Description**
GetClassID	Returns the class identifier (CLSID) for the component object.

{PRIVATE}IPersistStreamInit Methods	Description
IsDirty	Checks the object for changes since it was last saved.
Load	Initializes an object from the stream where it was previously saved.
Save	Saves an object into the specified stream and indicates whether the object should reset its dirty flag.
GetSizeMax	Return the size in bytes of the stream needed to save the object.
InitNew	Initializes an object to a default state.

4.2 CORBA Persistence Mechanisms

CORBA is a middleware facing towards large-scale systems integration such as in Enterprise Application Integration purposes. Banks run transaction systems using CORBA systems which connect hundreds of computers all over the world, for example. This is unlike COM which actually aims at the integration of smaller components (such as controls) on a single desktop computer. Although the underlying technical basics of the two technologies are very similar, the differences become obvious when considering the infrastructures developed for them: CORBA services standardized by the OMG always focus on large-scale systems integration and are often very complex systems, always taking distribution issues into account whereever possible.

This is more true than anywhere else for persistence services which defines access to persistent data in distributed storage via a two-phase transactional protocol. This is clearly not the focus of process modeling components so a different mechanism is required here. Although CORBA components can transparently maintain a persistent state without interacting with their clients, this leads to the same difficulties as the original unit specification, where a number of data storages exists but no single file containing the complete model/simulation specification.

Hence, a mechanism similar to the COM approach presented above is required here. In the CORBA services world, such an approach is provided by the COS property service specification. It defines two interfaces representing a property set (with a similar meaning to a property bag). The first is called PropertySet and does not specify any information about properties whereas the PropertySetDef interface allows additional information (modes) such as read-only properties to be specified. These interfaces are defined by the OMG and implementations for these services are available e.g. as part of the free ORB TAO (Schmidt, 2001) so that implementers of CORBA-based simulation systems can reuse them for developing persistence features of their systems.

However, we need a linkage between a PMC and a property set which is defined analog to the COM interface. The interface will be called *PropertySetPersist* and reside in the package *CapeOpen::Base::Persistence*. Interfaces which shall be persistence-capable A method *InitProperties* tells the PMC to initialize its properties internally. This method returns a boolean which is true if the PMC actually supports persistence and false otherwise. The method *LoadProperties* is given a property set from which the persistent properties can be loaded into the PMC. Finally, *SaveProperties* is given a property set into which persistent parameters shall be stored by the PMC. The PMC is free to return an object reference to a different property package as the one being passed. The PME will have to detect this and make sure to clean up the original property set object.

Some notes on the use of PropertySetDef. The PME can pass a PropertySetDef to a PMC using either LoadProperties or SaveProperties because PropertySetDef actually subclasses from PropertySet. In such a case, the PMC is strongly advised to respect the additional information that can be obtained from PropertySetDef. In the implementation of a PME supporting persistence, the implementer has to serialize/unserialize the information in the property bag during save and restore phases as needed. This specification does not make any assumptions how this is done internally. Also, the specification does not impose whether the PME actually implements the PropertySet interfaces on its own or uses an external implementation to do it.

```
// put the CORBA IDL here
#include <Cos/CosPropertyService.idl>

module CapeOpen
{
   module Base
   {
      module Persistence
      {
         interface PropertySetPersist
         {

           exception PropertySetPersistException
           { };

            boolean
            InitProperties()
               raises (PropertySetPersistException);

            void
            LoadProperties(in CosPropertyService::PropertySet prop_set)
               raises (PropertySetPersistException);

            void
            SaveProperties(inout CosPropertyService::PropertySet prop_set)
               raises (PropertySetPersistException);

         };
      };
   }

...
   module Unit
   {
      interface UnitOperation : CapeOpen::Base::Persistence::PropertySetPersist
      {

      };
   };
};
```

5. Notes on the interface specifications

6. Prototypes implementation

6.1 PMC implemented in VB

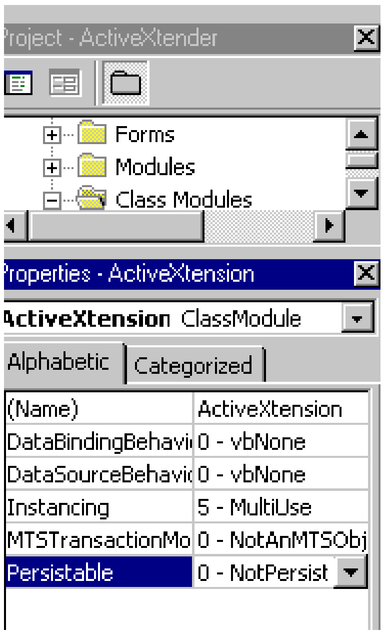
The VB way for a class module to implement the COM persistence interfaces is configuring the class as persistable. To do that:

(i) Select the corresponding "Class Modules" in the "Project Explorer" window.

(ii) Open the "Properties Window" (View menu)

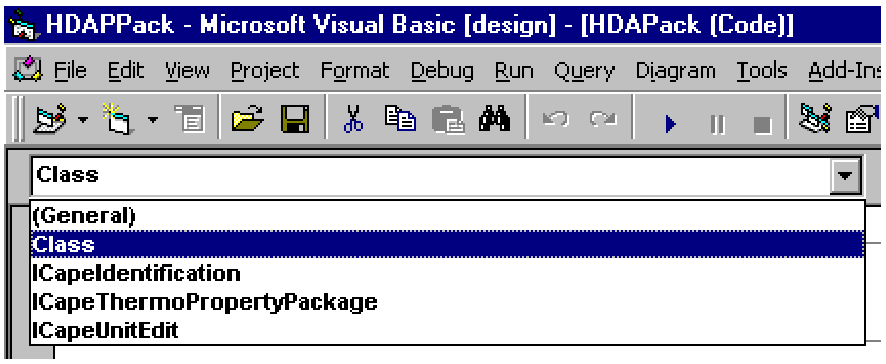
(iii) Select the "Persistable" property.

(iv) Change its value to "1 – Persistable"

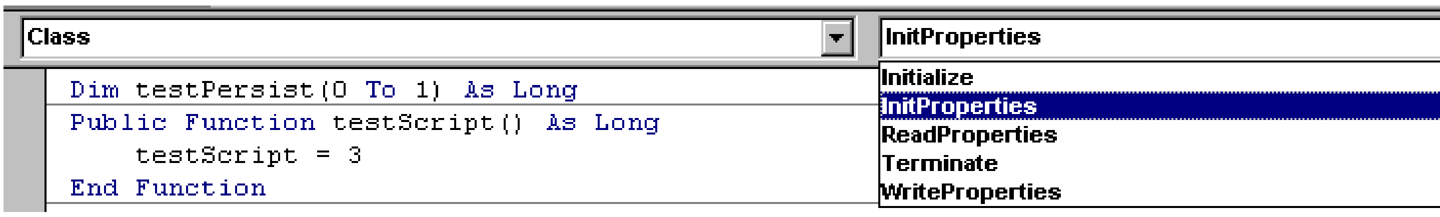


(Name)	ActiveXtension
DataBindingBehavi	0 - vbNone
DataSourceBehavi	0 - vbNone
Instancing	5 - MultiUse
MTSTransactionMo	0 - NotAnMTSObj
Persistable	0 - NotPersist

1. Click on the interface drop-down list. Select the class entry.



2. If your "Class Module" has been set to persistable, you will see three new procedure names: InitiProperties, ReadProperties and WriteProperties.


Class | InitProperties
```
Dim testPersist(0 To 1) As Long
Public Function testScript() As Long
    testScript = 3
End Function
```

3. Implement these 3 procedures:

```
Private m_persistableVariable as String
'Called after class_initialize when client uses CAPE-OPEN Persistence to create a new
instance.
'Be aware that only clients that support Persistence will call it
Private Sub Class_InitProperties()
End Sub

'called after class_initialize when client uses COMPersistence to restore a saved
instance.
Private Sub Class_ReadProperties(PropBag As PropertyBag)
 m_persistableVariable = PropBag.ReadProperty("notes")
End Sub

'called when client uses COMPersistence to persist an instance.
Private Sub Class_WriteProperties(PropBag As PropertyBag)
      Call PropBag.WriteProperty("notes", m_persistableVariable)
End Sub
```

4. Be aware that property bags allow storing any variant type except for arrays. For arrays, you may use these functions:

```
Public Function isArray(v as Variant) As Boolean
    isArray = VarType(v) And vbArray
End Function

'returns 0 if it is not an array
Public Function dimensions(v) As Integer
On Error GoTo ONE_DIM
    If isArray(v) Then
        Dim i&
        i = LBound(v, 2)
        dimensions = 2
    Else
        dimensions = 0
    End If
    Exit Function
ONE_DIM:
    dimensions = 1
End Function

'Saves variant 'm_value' into PropertyBag 'PropBag' identifying it with name 'varID'
Public Sub Save(PropBag As PropertyBag, m_varID as String, m_value as Variant)
    Dim d&, vt&
    Dim v
    d = dimensions(m_value)
    vt = VarType(m_value)
    Call PropBag.WriteProperty(hideBlanks(m_varID) & "_vt", vt)
    Call PropBag.WriteProperty(hideBlanks(m_varID) & "_Dim", d)

    If d = 1 Then
        Call PropBag.WriteProperty(hideBlanks(m_varID) & "_lb", LBound(m_value))
        Call PropBag.WriteProperty(hideBlanks(m_varID) & "_ub", UBound(m_value))

        Dim i&
        For i = LBound(m_value) To UBound(m_value)
            Call PropBag.WriteProperty(hideBlanks(m_varID) & "_" & i, m_value(i))
        Next
    ElseIf d = 0 Then
        Call PropBag.WriteProperty(hideBlanks(m_varID), m_value)
    Else
```

```
        Call MsgBox("CapePublicVariable does not support persisting 2dimensional arrays")
    End If

End Sub

‘Restores from PropertyBag ‘PropBag’ a variant identified with name ‘varID’ and places
value into ‘m_value’
Public Sub Load(PropBag As PropertyBag, m_varID as String, m_value as Variant)
    Dim d&, vt&
    Dim v
    Dim ub&, lb&

    vt = PropBag.ReadProperty(hideBlanks(m_varID) & "_vt")
    d = PropBag.ReadProperty(hideBlanks(m_varID) & "_Dim")

    If d = 1 Then
        lb = PropBag.ReadProperty(hideBlanks(m_varID) & "_lb")
        ub = PropBag.ReadProperty(hideBlanks(m_varID) & "_ub")
        Select Case (vt - vbArray)
            Case vbDouble:
                ReDim m_value(lb To ub) As Double
            Case vbInteger:
                ReDim m_value(lb, ub) As Long
            Case vbString:
                ReDim m_value(lb, ub) As String
            Case vbVariant:
                ReDim m_value(lb, ub) As Variant
            Case Else
                Call MsgBox( "CapePublicVariable does not support persisting vartype =" &
(vt - vbArray))
        End Select
        Dim i&
        For i = lb To ub
            m_value(i) = PropBag.ReadProperty(hideBlanks(m_varID) & "_" & i)
        Next
    ElseIf d = 0 Then
        m_value = PropBag.ReadProperty(hideBlanks(m_varID))
    Else
        Call MsgBox( "CapePublicVariable does not support persisting 2dimensional
arrays")
    End If
End Sub
```

6.2 PME implemented in C++

```
typedef enum {
    NO_PS = 0,      //no persistence
    IPSTG,          //IPersistStoragePtr
    IPSTR,          //IPersistStreamPtr
    IPSTI           //IPersistStreamPtrInit
  } COMPersistType;
IDispatch* m_CAPEOPEN_PMC;
IStoragePtr openStorage();
_bstr_t persistStreamName();


STDMETHODIMP CCCAPEGUI::getPersistType(COMPersistType* persistenceSupported) {
HRESULT hr(S_OK);
IDispatch* comp;
  comp = m_CAPEOPEN_PMC;
  try {
    IPersistStoragePtr ipstg;
    if (ipstg = comp) {
      *persistenceSupported = IPSTG;
    } else {
      IPersistStreamInitPtr ipsti;
      if (ipsti = comp) {
        *persistenceSupported = IPSTI;
      } else {
        IPersistStreamPtr ipstr;
        if (ipstr = comp) {
          *persistenceSupported = IPSTR;
        } else {
          *persistenceSupported = NO_PS;
        }
      }
    }
  }catch(...) {
    //error
  }
  return S_OK;
}


STDMETHODIMP  CCCAPEGUI::saveStorage() {
HRESULT hr(S_OK);
  try{
    IStoragePtr istg;
    IStreamPtr stream;

    istg = openStorage();
    _bstr_t name;
    name = persistStreamName();
    switch (m_persistType) {
      case IPSTG:{
        IPersistStoragePtr ipstg;
  ipstg = m_CAPEOPEN_PMC;
        ASSERT(hr, ipstg->Save(istg, true));
        ipstg->SaveCompleted(istg);
        break;

      }case IPSTI:{
        IPersistStreamInitPtr ipsti;
  ipsti = m_CAPEOPEN_PMC;
        ASSERT(hr,  istg->CreateStream(name, STGM_DIRECT | STGM_CREATE | STGM_READWRITE |
STGM_SHARE_EXCLUSIVE, 0, 0, &stream));
        ASSERT(hr,  ipsti->Save(stream, true));
        ASSERT(hr,  istg->Commit(STGC_DEFAULT));
  break;
```

```
      }case IPSTR:{
        IPersistStreamPtr ipst;
  ipst = m_CAPEOPEN_PMC;
  ASSERT(hr,  istg->CreateStream(name, STGM_DIRECT | STGM_CREATE | STGM_READWRITE |
STGM_SHARE_EXCLUSIVE, 0, 0, &stream));
        ASSERT(hr,  ipst->Save(stream, true));
        ASSERT(hr,  istg->Commit(STGC_DEFAULT));
        break;
      }
    }
    return hr;
  }catch(...) {
    //error
  }
}


STDMETHODIMP  CCCAPEGUI::loadStorage(bool isRecalling) {
HRESULT hr(S_OK);
  try{
    if (m_persistType==NO_PS) return hr;
    IStoragePtr istg;

    istg = openStorage();
    if (!isRecalling)
        return initStorage(istg);

    _bstr_t name;
    IStreamPtr stream;

    name = persistStreamName();

    switch (m_persistType) {
      case IPSTG:{
        IPersistStoragePtr ipstg;
  ipstg = m_CAPEOPEN_PMC;
        ASSERT(hr, ipstg->Load(istg));
        break;
      }case IPSTI:{
        IPersistStreamInitPtr ipsti;
  ipsti = m_CAPEOPEN_PMC;
        ASSERT(istg->OpenStream(name, NULL, STGM_DIRECT | STGM_READ |
STGM_SHARE_EXCLUSIVE, 0,  &stream));
        if (hr  == STG_E_FILENOTFOUND) return S_OK;
        ASSERT(hr, ipsti->Load(stream));
        break;
      }case IPSTR:{
        IPersistStreamPtr ipst;
  ipst = m_CAPEOPEN_PMC;

         ASSERT(istg->OpenStream(name, NULL, STGM_DIRECT | STGM_READ |
STGM_SHARE_EXCLUSIVE, 0,  &stream));
        if (hr  == STG_E_FILENOTFOUND) return S_OK;
        ASSERT(hr, ipst->Load(stream));
        break;
      }
    }
    return hr;
  }catch(...) {
    //error
  }
}

HRESULT CCCAPEGUI::initStorage(IStorage* istg) {
  HRESULT hr(S_OK);
  switch (m_persistType) {
    case IPSTI:{
      IPersistStreamInitPtr ipsti;
Ipsti = m_CAPEOPEN_PMC;
ipsti->InitNew();
      break;
    }case IPSTG:{
      IPersistStoragePtr ipstg;
       Ipstg= m_CAPEOPEN_PMC;
       ipstg->InitNew(istg);
      break;
    }
  }
  return hr;
}
```

7. Specific Glossary Terms

8. Bibliography

- OMG, 2000: Property Service Specification. Available online at: http://www.omg.org/technology/ documents/formal/property_service.htm
- Schmidt, D.C.: Real-time CORBA with TAO (The ACE ORB). See online at http://www.cs.wustl.edu/ ~schmidt/TAO.html

9. Appendices