CAPE-OPEN

Expanding Process Modelling Capability
Through Software Interoperability Standards

Cape Open Interface Specification:

Persistence Common Interface

www.colan.org

ARCHIVAL INFORMATION

Title:	Open Interface Specification: Persistence Common Interface
Owner	M&T SIG
Location	-
Document Unique Identifier:	A5149DF0-95D1-11EF-B559-0800200C9A66
Distribution	Internal
Status	Draft
Document Version Number	1.2.0.5
Date Created	2024-07-23
Revision Date	2026-06-08

Version History

Note: Provides summary information regarding released versions. Most recent version appended at the bottom of the table.

Document Version Number	RFC Date	Release Date	Comments
1.2.0.0		2024-07-02	M&T SIG Meeting
1.2.0.2		2024-07-23	M&T SIG Meeting
1.2.0.3		2024-10-29	M&T SIG Meeting
1.2.0.4		2025-12-12	M&T SIG Meeting
1.2.0.5		2026-06-08	M&T SIG Meeting

IMPORTANT NOTICES

Disclaimer of Warranty

CO-LaN documents and publications include software in the form of *sample code.* Any such software described or provided by CO-LaN --- in whatever form --- is provided "as-is" without warranty of any kind. CO-LaN 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 Laboratories Network --- remains with you.

CO-LaN is a not-for-profit organization established under French law of 1901.

Trademark Usage

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

SUMMARY

This document describes a Common Interface: the Persistence Common Interface. The Common Interfaces are interfaces and implementation models for handling concepts that may be required by any CAPE-OPEN Object. This Persistence Common interface specification is part of version 1.2 of the CAPE-OPEN standards.

An interface is defined on a Persistable Object (PO), typically a Process Modelling Component (PMC) that indicates that the component is a PO and is capable of being persisted and restored from persistence by the Persistence Host (PH), typically the PME. Persistence of a PO occurs when the PH requests the PO to save itself by invoking the *ICapePersistence::Save* method. The PH provides the PO with access to a data store used by the PO to store internal information and the PH indicates whether the PO should clear its dirty status. The PH can request that the PO be restored by invoking the *ICapePersist::Load* method, providing a data store that contains information previously written by the PO. By implementing the *ICapePersistReader* or *ICapePersistWriter* interface, the data store enables the PO to write data to or read data from the store.

The scenarios envisioned for the use of these persistence interfaces include storing and retrieving instances of POs as part of saving a flowsheet to a file and copying of POs applications such as cut and paste copying, or duplication of the PO on different threads.

An appendix is included that provides mechanisms for transitioning COM persisted CAPE-OPEN v1.x objects to persistence using CAPE-OPEN persistence.

ACKNOWLEDGEMENTS

This CAPE-OPEN Interface Specification Document has been developed by the Methods & Tools Special Interest Group within the common interface specifications. The following are the main contributors:

Bill Barrett	US Environmental Protection Agency
Jasper van Baten	AmsterCHEM
Michael Hlavinka	Bryn Research & Engineering, LLC
Kyle Abrahams	CO-LaN

CONTENTS

LIST OF FIGURES

1. Introduction

Most simulation environments allow the state of a simulation case to be stored at any moment so it can be restored at any time in the future. This ability allows for sharing of simulation files amongst different members of a design team, return to a prior state following design modifications, or archiving at various stages of the design process.

A key feature of CAPE-OPEN is that different components used in the simulation may be implemented by different vendors. To ensure that a simulator can store all of the pieces of the simulated process, there must be a standard mechanism to store and restore Process Modelling Components (PMCs) within a Flowsheet. Persistence is not limited to file storage of a flowsheet but applies to any scenario in which the state of an object can be stored or restored, for example transferring the state of an object between applications or threads.

Persistence allows the Persistence Host (PH) to save and restore the current state of a Persistable Object (PO) whenever the need arises. Uses of persistence include saving the state of POs to a data store that is either permanent storage, such as a data file, or ephemeral for local transfer, such as copying/pasting or duplication on different threads. The data are written into and read from the data store. Persistence may also be used as part of the management of multi-threaded applications to move a PO from the control of one thread to another.

CAPE-OPEN did not define Persistence Interfaces as part of the Persistence Common Interface Specification, Version 2 document published in 2003. Instead, at that time, CAPE-OPEN adopted the use of platform-specific persistence mechanisms on either the Microsoft Component Object Model (COM) or Common Object Request Broker Architecture (CORBA) platform. With the development of the CAPE-OPEN Binary Interop Architecture (COBIA), a cross-platform persistence mechanism is required to enable platform-independent persistence of CAPE-OPEN Components. This document defines this set of platform-independent CAPE-OPEN Persistence Interfaces for use across CAPE-OPEN components.

1.1 Applicability of Persistence Interfaces

The interfaces specified in this document are available in COBIA-based CAPE-OPEN version 1.2. The interfaces are designed with a future CAPE-OPEN version 2.0 standard in mind, and in that version will also become part of the COM interface set.

CAPE-OPEN versions prior to 1.2, including COM based CAPE-OPEN 1.1, use middleware-specific persistence interfaces. This document provides guidance on updating from platform specific persistence, as well as transitioning from platform-based persistence mechanisms to CAPE-OPEN persistence. COMBIA provides interoperability between COBIA based CAPE-OPEN 1.2, and COM based CAPE-OPEN 1.1; in doing so it provides transparent interoperability between the interfaces specified in this document, and COM based persistence.

Applications that used to be written using COM and the CAPE-OPEN 1.1 implementation, and would like to transition to COBIA and the CAPE-OPEN 1.2 implementation, are offered the route of transition as outlined in Appendix A - Transitioning between COM-based Persistence and CAPE-OPEN Persistence.

1.2 Document Scope

This document defines the CAPE-OPEN Persistence Common Interface, designed to standardize the mechanisms by which CAPE-OPEN components persist and restore internal state.

After this introduction, the document is organized with a section listing the requirements leading to the design proposed, followed by the description of several Use Cases as well as Sequence and State Diagrams for the interfaces defined. Next a section describes each interface and their methods in details.

The scope of this specification covers the following key aspects:

- Persistence Mechanisms:

The document specifies a unified persistence framework for Persistable Objects (POs) — typically process modeling components (PMCs)—and the Persistence Host (PH) — typically a PME — that manages the storage medium. It details how a PO saves its complete internal state (including configuration data, parameter values, and other modifiable attributes) to a structured data store and later restores this state in a consistent, lossless manner.

It is possible for POs to maintain a history/archive of past states through the persistence process but doing so would mean that the PO needs to maintain and storage that archive itself through the persistence mechanism.

- Core Interface Definitions:

The persistence mechanism is built around three core interfaces:

ICapePersist: Exposed by POs to signal support for persistence and to initiate save/load operations.

ICapePersistWriter: Provides a standardized API for writing the PO's internal state into a hierarchical tree structure of key/value pairs.

ICapePersistReader: Enables retrieval of stored data, ensuring that a PO can accurately reconstitute its state.

- Structured Data Storage:

The specification mandates that all data to be persisted is organized in a tree structure where each node—represented by a CAPE-OPEN object—contains unique key/value pairs. This approach simplifies error handling, supports lossless storage, and ensures that the entire state of the PO is captured in one operation (no partial persistence).

- Operational Guidelines:

The document includes a full set of textual requirements, use cases, sequence and state diagrams, and interface definitions that detail:

  - The process by which the PH supplies a writer or reader object to the PO,
  - The complete lifecycle of saving a PO's state (including deep copy operations) and subsequently restoring that state,
  - The error handling, memory management, and data type support (including real numbers, integers, Booleans, strings, enumerations, and their arrays) necessary to guarantee interoperability across diverse implementations.

- Transitional Guidance for Legacy Systems: Outlined in Appendix A - Transitioning between COM-based Persistence and CAPE-OPEN Persistence

2. Requirements

This section lists the functionalities required to implement persistence of POs and gathers textual requirements for the PO and PH implementing and consuming the CAPE-OPEN Persistence. The relationships between the PO and PH are also listed and described.

2.1 Textual requirements

Textual requirements are listed hereafter and referenced by mentioning the software component to which each requirement applies and its number in the global list of requirements. The requirements styled REQ-PO- refer to requirements on the PO, typically a primary PMC. Requirements styled REQ-PH- refer to requirements on the PH, the object requesting persistence of the PO, typically the PME.

REQ-PMC-01: a PMC must support persistence if it can be configured

Rationale: if the Flowsheet User can change the configuration of the PMC, the PMC must support persistence and therefore become a PO. This implies that any PMC that exposes public parameters must persist itself. Note that this requirement also holds for PMCs that only expose output parameters. Also, PMCs that support *ICapeUtilities::Edit*, and configuration changes can be made during *Edit*, must support persistence. This requirement does not hold for PMCs that implement *Edit* but no configuration changes are possible during *Edit*.

Note that a PMC that does not have parameters or allow for configuration changes through *ICapeUtilites::Edit*, does not need to support persistence, however, the name is still modifiable (as per the *ICapeIdentification* common interface specification[1] general requirements). If the PME makes changes to the component name, the PME is responsible for persisting the component name.

It is possible that a PMC has an internal state which is not manipulated through public parameters or *Edit*. For example, the Unit Operation may save its last solution as an initial guess for the next calculation. Therefore, a PMC may always implement persistence even if it is not required, in accordance with **REQ-PMC-01**.

REQ-PH-02: the PH must store the information needed to create the PO.

Rationale: prior to restoration of a PO, the PH needs to be able to create the PO. So, the PH needs to maintain and store the information needed to create an instance of the PO to be restored. POs may either be created directly by the PH through middleware or using a Manager Object.

For POs that can be created directly by middleware, the PH needs to store the information required by middleware to create the PO.

Creation of any PO that is managed by a Manager Object is performed by the Manager Object. In order to restore a managed PO, the PH will need to request that the Manager Object instantiates the appropriate PO. Therefore, the PH needs to store the identifier of the managed PO known to the Manager Object to recreate the PO. This requirement is fulfilled internally by the PH.

REQ-PO-03: Manager Objects are not persistable.

Rationale: Manager Objects are configurable PMCs. As such they could be considered for persistence. However, their configuration falls outside of the scope of the Flowsheet and therefore Manager Objects do not need to be persisted as part of the Flowsheet. Maintaining the private configuration of any Manager Object is the responsibility of the Manager Object itself.

The requirement is fulfilled by mechanisms internal to the Manager Object.

REQ-PO-04: any object that can be persisted must advertise that it is a PO**.**

Rationale: persistence allows a PH, which is typically a PME to save and restore the configuration of a PO, which is typically a PMC within a Flowsheet. PMCs are not required to support persistence unless the PMC has a state that it desires to be configured by the Flowsheet User through the PME using *ICapeUtilities::Edit* method or by exposing public Parameters. Therefore, a PH must be able to detect that any given PMC is a PO.

In accordance with the objective of simplification, a single interface is managing persistence of a PO. The presence of a single interface needs to be checked by a PH to determine that the object is a PO. Exercising persistence/depersistence is handled through a single interface on the PO, another aspect of simplification.

This requirement is fulfilled by the presence of the *ICapePersist* interface on the PO.

REQ-PH-05: the PH provides for the storage of the PO's internal data.

Rationale: persistence and restoration of the internal state of a PO is incorporated in the process used by the PH to persist and restore Flowsheets. Persistence and restoration of the complete Flowsheet by a PH is outside of the scope of this specification. Therefore, the storage used by the PO to persist its internal state must belong to the PH. This storage used for the POs internal data is referred to as the data store. There is no requirement regarding how the PH implements the data store.

Fulfillment of this requirement is the responsibility of the PH.

REQ-Design-06: it must be possible for the PH to store persisted data in a human readable form.

Rationale: the ability to view persisted data in a human-readable format is more transparent and provides a means to inspect archived data. Providing human readable storage is optional, and a responsibility of the PH.

This requirement is fulfilled by design of the *ICapePersistWriter* and *ICapePersistReader* interfaces.

REQ-PH-07: the data store is exposed to the PO as the root node of a tree data structure dedicated to that PO.

Rationale: requiring use of the *node* argument in *ICapePersist::Save* and the *reader* argument in *ICapePersist::Load* ensures a uniform, easy-to-understand pattern for data-store interaction. It also creates distinct node for each PO's persisted data, safeguarding against collisions between multiple POs.

REQ-Design-08: each node is represented as a CAPE-OPEN object.

Rationale: The object that represents a node of the data store is accessed through CAPE-OPEN interfaces (*ICapePersistReader* or *ICapePersistWriter*). Only the PH may instantiate a node, and a node can only be used for persistence.

This requirement is fulfilled by the *ICapePersistWriter* and *ICapeReader* interfaces.

REQ-PH-09: Each node of a reader or writer object is a CAPE-OPEN object, and is required to implement the *ICapeIdentification* interface.

Rationale: The *ICapeIdentification* implementation holds for all CAPE-OPEN objects. The node path relative to root note is useful for debugging and logging purposes.

REQ-PH-10: when the PH invokes the *ICapePersist::Save* method, the PH provides the PO with a root node containing zero child nodes and zero key/value pairs to the PO.

Rationale: this provides consistency across implementations and reduces complexity of the storage operation by eliminating the need to handle updating previously stored data. The requirement is fulfilled through the *rootNode* argument of the *ICapePersist::Save* method.

REQ-PH-11: during the invocation of the *ICapePersist::Save* method, data is placed into a node using the *ICapePersistWriter* interface.

Rationale: the specification does not specify the structure of the data store, rather the methods to place data into it.

This requirement is fulfilled by the *ICapePersistWriter* interface.

REQ-PH-12: during the invocation of the *ICapePersist::Save* method, the PO can create and acquire child nodes on a node to build the tree structure.

Rationale: each PO must be able to create a tree structure suitable for storage of its internal information.

This requirement is fulfilled by the *ICapePersistWriter::AddNode* method.

REQ-PH-13: when the PH invokes the *ICapePersist::Load* method, the PH provides the saved data to the PO in the form of the root node containing the previously-saved child nodes and key/value pairs.

Rationale: the implementation of the data store is not defined within this specification, but the data must be presented to the PO as it was placed into the data store during the *ICapePersist::Save* method.

The requirement is fulfilled through the *rootNode* argument of the *ICapePersist::Load* method.

REQ-PH-14: during the *ICapePersist::Load* method, data is retrieved from a node using *ICapePersistReader* interface.

Rationale: the specification does not specify the structure of the data store, rather the method for retrieving data from it.

This requirement is fulfilled by methods of the *ICapePersistReader* interface.

REQ-PH-15: during the *ICapePersist::Load* method, the PO can access child nodes of a node to traverse the tree structure.

Rationale: the PO needs to be able to traverse the tree structure to recover its internal information from storage.

This requirement is fulfilled by the *ICapePersistReader::GetNode* method.

REQ-PH-16: each node provides a collection of key-value pairs.

Rationale: a dictionary data structure within each node was selected for placement of PO data into the data store at each level of the data store.

This requirement is fulfilled by accessing each value by its name.

REQ-PO-17: child node and key names must be unique *CapeString* values that comply with the same rules as the component names in the Identification Common interface specification.

Rationale: values and nodes are accessed by name; therefore, names should be unique within each node, in a case-insensitive manner. Enforcing uniqueness prevents naming collisions and lets implementations use a single data structure to hold all entries.

Fulfillment of this requirement is the responsibility of the PO within the *ICapePersist::Save* method.

REQ-PH-18: the PH must accept any non-empty *CapeString* value as the key or node names.

Rationale: no limitation is placed upon the PO regarding the naming of keys or nodes aside from those in REQ-PH-16.

This requirement is fulfilled by methods of the *ICapePersistWriter* interface.

REQ-PO-19: references to nodes need to be released within the Load or Save method.

Rationale: the PO can only interact with the data store during Save and Load operations. This requirement is fulfilled by middleware.

Fulfillment of this requirement is the responsibility of the PO.

REQ-PH-20: the data store needs to support storage or retrieval of any data.

Rationale: the internal state to be persisted includes all modifiable data of the PO. Therefore, the data types used to store internal data should not be a limitation to the persistence process.

This requirement is fulfilled by the definition of CAPE-OPEN Persisted Data Types which are supported by the methods of *ICapePersistWriter* and *ICapePersistReader*. These Data Types include real numbers, integer numbers, Boolean values, character strings, enumerations, arrays of real numbers, arrays of integer numbers, arrays of Boolean values, arrays of strings, arrays of enumerations, arrays of bytes. This requirement is fulfilled by methods of the *ICapePersistWriter* interface.

REQ-PH-21: the data store must provide lossless storage and retrieval of data.

Rationale: the PO needs to be able to restore its previous state from the data that was placed into and retrieved from the data store.

For example, the *CapeReal* data type is a binary floating-point value that may not be exactly represented as a text-based decimal. The conversion to and from a *CapeReal* value to a text-based decimal value needs to retain the exact numeric value of the real number that was stored.

In another example, a string may be saved by the PO encoded as UTF-16 but could be placed in the data store with UTF-8 encoding. In that case it must be returned to the PO in the original encoding (in this case UTF-16) string when retrieved.

Fulfillment of this requirement is the responsibility of the PH.

REQ-PO-22: the PO must be able to inform the PH if the PO internal data needs to be stored.

Rationale: the PH may wish to not update the stored representation of the PO if the state of the PO has not been modified since the last time the PO was stored. The PO needs a mechanism to inform the PH of whether storage of the PO is required.

This requirement is fulfilled by the *IsDirty* property of the PO exposed through the *ICapePersist* interface.

REQ-PH-23: the PH must instruct the PO whether to clear its dirty flag.

Rationale: the PO's dirty flag is used to indicate to the PH whether the PO has been modified since the PH placed the stored data for the PO into permanent storage. The PO does not know if the storage performed on request of the PH is permanent or not. Since the PH initiates the request, the PH knows whether the request for a PO to store its data will place the data into permanent storage.

For example, the PH may serialize the PO into temporary storage for purposes such as copying the PO from one thread to another in multi-threaded applications. In this case, the stored data has not been saved to permanent storage, so the current state of the PO cannot be restored in the future from a permanent storage.

This requirement is fulfilled by the *clearDirty* argument of the *Save* operation of the *ICapePersist* interface. Through this argument, when the PH requests the PO to save its internal data, the PH specifies to the PO if the dirty flag must be cleared.

REQ-PO-24: the PO must place all information into the data store required to restore its persisted state.

Rationale: the PO must store its internal state so that it can be restored to an identical state. The internal state to be persisted includes all modifiable data of the PO, for example:

- Name and description (if modifiable) of the PO **(See REQ-PMC-01)**,
- State of all Parameters (such as their current values)
- Parameter specifications (such as default values, upper and lower bounds, and options list) that are unmodifiable through CAPE-OPEN interfaces but modifiable through internal configuration mechanisms (typically for Unit Operations via *ICapeUtilities::Edit*)
- Any configuration data private to the PO, such as the local setup of a Property Package Manager.

There is no need to persist data that can be reconstructed internally by the PO to the same state it was in at the time the PO was saved. However, the PO needs to persist anything it cannot re-create on demand. For example, reports that can be immediately re-created do not need to be persisted however reports that cannot be regenerated should be persisted. Context dependent objects are not saved, for example the active Material Object that is set using *ICapeThermoMaterialContext::SetActiveMaterial*.

REQ-PH-25: the PH must place all the information necessary to restore the persisted state of a PO in the data store

Rationale: the PH needs to save the object type of the PO to instantiate the appropriate PO object. It is recommended that in COM and COBIA, this is the class id (CLSID). For POs that are created through a Property Package Manager or Reaction Package Manager, the PH needs to store the name of the managed PO to recreate the PO. After the PO has been created, the PH can call load on the managed POs.

Port connections are not saved by the PO but are restored by the PH (after initialization of the PO).

The PH should not merely save and restore a PO's parameter values (via the *ICapeParameter* interface) as a substitute for persistence of the PO itself.

REQ-PH-26: The PH is responsible for reimposing all external conditions of the POs after depersisting

Rationale: The PME is responsible for resetting the Simulation Context of the PMCs.

Port connections are not saved by the PMC but are restored by the PME (after initialization of the PMC).

REQ-PH-27: The key name "COBIA" in the root node is reserved, and cannot be used by the PH.

Rationale: See Appendix A - Transitioning between COM-based Persistence and CAPE-OPEN Persistence; the key name "COBIA" is reserved for transitioning and recognizing objects stored by COMBIA, and only COMBIA may use this key.

2.2 Use Cases

This section contains the set of Use Cases that explain how to make use of the interfaces defined in this document.

Use Cases Priorities

All Use Cases are given a high priority. Hence all Use Cases must be fulfilled.

- **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.

2.2.1 Actors

- **PH -** The Persistence Host is any object that can request that a Persistable Object store or retrieve its state using the defined persistence mechanism. The Persistence Host is typically a Process Modeling Environment (PME).
- **PO -** Any component that exposes the ability to persist itself using the CAPE-OPEN persistence mechanisms. POs are typically Primary PMC objects that implement the *ICapeUtilities* interface. Additionally, the CAPE-OPEN persistence mechanisms could also be used by the PME to persist software components internal to the PME.

2.2.2 List of Use Cases

This section presents a list of use cases that together define the full spectrum of persistence operations within the CAPE-OPEN framework. The use cases detail the interactions and responsibilities of the two primary actors—the Persistence Host (PH) and the Persistable Object (PO)—as well as additional roles (such as the Property Package Manager) when applicable. They describe how the PH and PO work in tandem to save, restore, and duplicate a PO's complete internal state using standardized persistence interfaces.

2.2.3 Use Cases Map

Use Cases Map as well as individual descriptions of each Use Case have been produced by assigning high level actors (i.e. Persistence Host and Persistence Object).

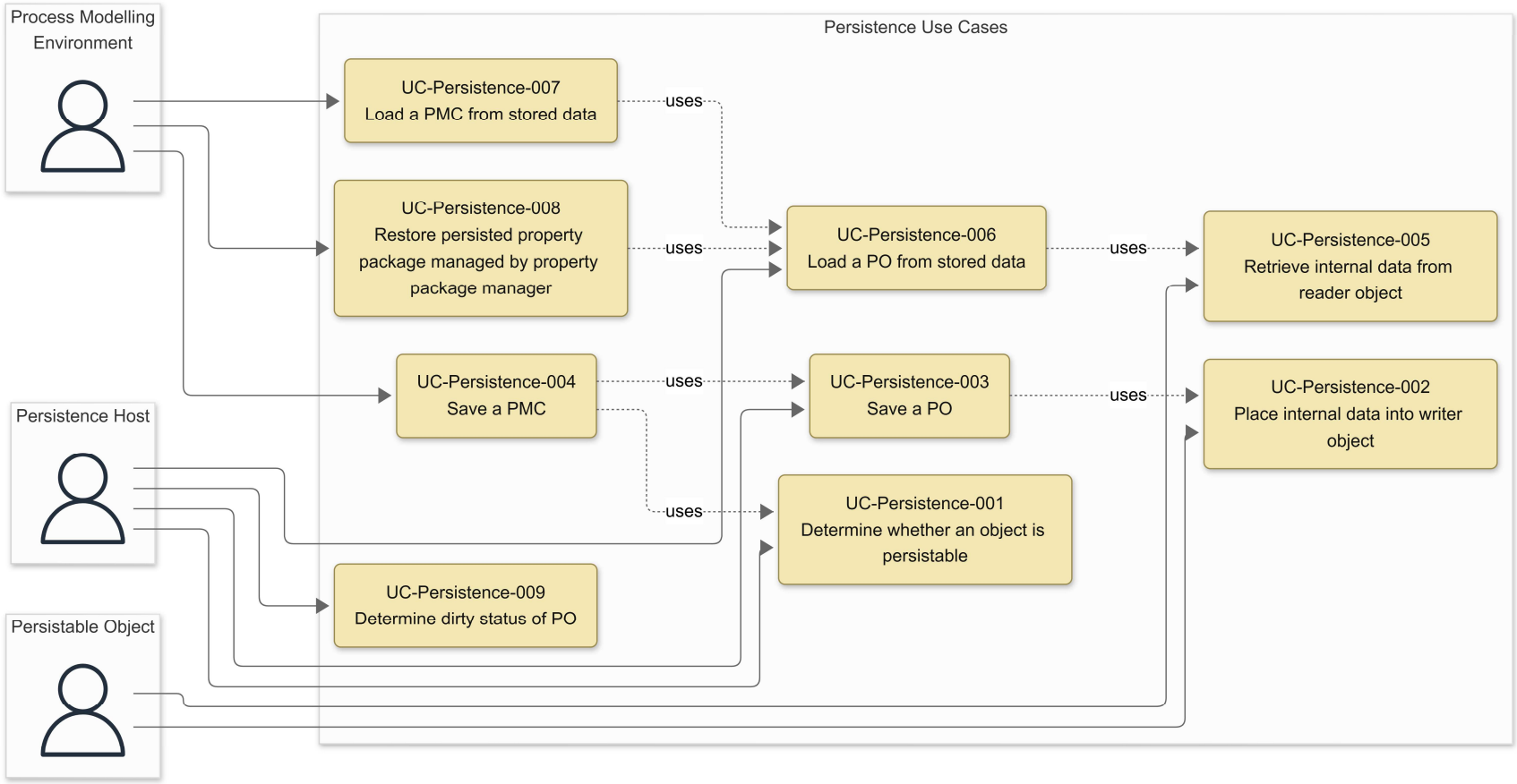


Figure 1. Persistence Use Case Map

2.2.4 Use Cases

UC-Persistence-001 DETERMINE WHETHER AN OBJECT IS PERSISTABLE.

Actor: <**Error! Reference source not found.**>

Priority: <**High**>

Status: <This Use Case is fulfilled by middleware mechanisms>

Pre-conditions: <Target object is initialized>

Flow of events:

PH queries the target object using middleware mechanisms to determine whether it implements the *ICapePersist* interface. If the target object implements the *ICapePersist* interface, then the target object is a PO.

Post-conditions: <PH knows whether the target object is a PO>

Errors: <None>

Uses: <None>

Extends: <None>

UC-Persistence-002 PLACE INTERNAL DATA INTO WRITER OBJECT.

Actor: <Persistable Object (PO)>

Priority: <High>

Status: <This Use Case is fulfilled by the *ICapePersist::Save method*>

Pre-conditions: <The PO has a reference to a *writer* object that implements the *ICapePersistWriter* interface>;
<The PO has been informed of whether to clear its dirty flag using the *clearDirty* argument>

Flow of events:

The PO considers the *writer* object to be a node. The PO can execute the following actions on a node any number of times:

- The PO may place data values by name into a node by invoking the type-specific methods of the *ICapePersistWriter* interface on the *writer* object.

- The PO may request creation of, and acquire the reference to a child node of a *writer* object by invoking the *ICapePersistWriter::AddNode* method on the *writer* object.

PO updates its *dirty flag* to conform to status consistent with the *clearDirty* argument.

Post-conditions: <Internal data of PO has been persisted>
<Dirty status of PO conforms to the *clearDirty* argument>

Errors: <None>

Uses: <None>

Extends: <None>

UC-Persistence-003 SAVE A PO.

Actor: <**Persistence Host (PH)**>

Priority: <**High**>

Status: <This Use Case is fulfilled by invoking the *ICapePersist::Save* method>

Pre-conditions: <PO is initialized>
<PH has a writer object>

Flow of events:

Basic Path:

The PH provides a root node that implements *ICapePersistWriter* to the PO for use in storing the PO's internal Data.

The PH determines whether the PO should clear its dirty flag.

The PH requests that the PO saves its internal data by invoking UC-Persistence-002, passing a root node to the PO for use in saving its internal data and indicating whether the PO should clear its dirty flag.

Finally, the PH stores the information needed to recreate the PO or its manager, as well as the data stored to the *ICapePersistWriter*, using machinery internal to the PH.

Post-conditions: <The PO's internal data is stored within the data store>

Errors: <Out of storage space>

Uses: < UC-Persistence-002>

Extends: <None>

UC-Persistence-004 SAVE A PMC.

Actor: <**Process Modelling Environment (PME)**>

Priority: <**High**>

Status: <This Use Case is fulfilled by the machinery internal to the PME>

Pre-conditions: <PMC is initialized>

Flow of events:

Basic Path:

The PME determines and stores sufficient information to recreate the PMC, or if the PMC is managed by a Manager Object, sufficient information to recreate the PMC using its manager. In the special case of a Property Package managed by a Property Package Manager, the name that was used to create the Property Package must be stored.

It is recommended that this information is stored as the CLSID of the PMC or its Manager. Alternatively, one could store the ProgID (if available), or version independent ProgID of the PMC or its Manager. Both these optional pieces of information point back to the CLSID. The link from ProgID and version independent ProgID and version ProgID to CLSID may change between different versions of the software that serves the PMC.

Determine if the PMC is a PO by using UC-Persistence-001 (PMCs managed by *ICapeManager* are always POs). If the PMC is not a PO, save sufficient information to restore the state, such as the name and description if they are modifiable, and the Use Case ends here.

The PME invokes and UC-Persistence-003

Post-conditions: <The PME can restore an interchangeable instance of the PMC by invoking **Error! Reference source not found.**>

Errors: <Out of storage space>

Uses: < UC-Persistence-001, UC-Persistence-003>

Extends: <None>

UC-Persistence-005 RETRIEVE INTERNAL DATA FROM READER OBJECT.

Actor: <Persistable Object (PO)>

Priority: <High>

Status: <This Use Case is fulfilled by the *ICapePersist::Load* method>

Pre-conditions: <PO is initialized>;
<The PO has a reference to a *reader* object that implements the *ICapePersistReader* interface>;

Flow of events:

The PO considers the *reader* object to be a node populated with data it previously stored. The PO can execute the following actions on a node any number of times:

- The PO may retrieve data values by name from a node by invoking the type-specific methods of the *ICapePersistReader* interface on the *reader* object.
- The PO may acquire the reference to a child node of a node by invoking the *ICapePersistReader::GetNode* method on the *reader* object.

PO clears its *dirty flag*.

Post-conditions: <Internal data of PO has been restored>
<Dirty status of PO is cleared>

Errors: <PO is unable to persist its internal data in the data store provided>

Uses: <None>

Extends: <None>

UC-Persistence-006 LOAD A PO FROM STORED DATA.

Actor: <Persistence Host (PH)>

Priority: <High>

Status: <This Use Case is fulfilled by invoking the *ICapePersist::Load* method>

Pre-conditions: < The PO instantiated > <The PH obtained a reader object by performing UC-Persistence-005>

Flow of events:

The PH retrieves the information needed to recreate the PO or its manager, and places the data stored into a *ICapePersistReader*, using machinery internal to the PH.

The PH invokes the *ICapePersist::Load* method on the PO passing a *reader* object to the PO for use in retrieving its internal data using UC-Persistence-005.

Post-conditions: <The PO's internal data has been restored to a prior state with data from the data store>

Errors:

Uses: < UC-Persistence-005>

Extends: <None>

UC-Persistence-007 LOAD A PMC FROM STORED DATA.

For Property Packages that belong to a Property Package Manager, refer to use case UC-Persistence-008 instead.

Actor: <PME>

Priority: <High>

Status: <This Use Case is fulfilled machinery internal to the PME>

Pre-conditions: <The data that was previously stored for the PMC using UC-Persistence-004 is available>

Flow of events:

The PME retrieves from local or persistent storage the information required to create an instance of the PMC as part of UC-Persistence-004 or the Manager for the PMC. It is recommended that this information is represented by the class ID (CLSID)

If the PMC is managed by a Manager Object[2]:

- Using the retrieved information stored in UC-Persistence-004, the PME creates an instance of the Manager using either middleware protocols or the appropriate Manager Object.
- Create an instance of the PMC by invoking *ICapeManager::CreateForLoad()*

Otherwise:

- Using the retrieved information stored in UC-Persistence-004, the PME creates an instance of the PMC using either middleware protocols or the appropriate Manager Object.

If the PME supports the simulation context, set the Simulation Context.

If data was persisted as part of UC-Persistence-004, the PMC is a PO and its data is restored using UC-Persistence-006, the PME invokes *ICapeUtilities::Initialize* on the PMC.

If the name and/or description were stored in UC-Persistence-004, the PME imposes the name and the description on the PMC using *ICapeIdentification*.

Post-conditions: <The PMC is restored to its previous state and initialized> <The *Dirty* flag on the PMC is false>

Errors:

Uses: < UC-Persistence-006>

Extends: <None>

UC-Persistence-008 RESTORE AN INSTANCE OF PERSISTED PROPERTY PACKAGE, THAT IS MANAGED BY A PROPERTY PACKAGE MANAGER

Actor: <PME>

Priority: <High>

Status: <This Use Case is fulfilled by middleware functionalities available to the PME or by functionalities made available by a Property Package Manager>

Pre-conditions: <The data that was previously stored for the Property Package using UC-Persistence-004 is available>

Flow of events:

The PME retrieves from local or persistent storage the information required to create an instance of the Property Package Manager. It is recommended that this information is represented by the class ID (CLSID)

The PME creates an instance of the Property Package Manager using the middleware protocols.

The PME calls *Initialize* on the Property Package Manager.

Create an instance of the Property Package by invoking *ICapeThermoPropertyPackageManager::GetPropertyPackage()* passing the name that was stored inUC-Persistence-004.

If the PME supports the simulation context, set the Simulation Context.

If data was persisted as part of UC-Persistence-004, the PMC is a PO and its data is restored using UC-Persistence-006, the PME invokes *ICapeUtilities::Initialize* on the PMC.

If the name and/or description were stored in UC-Persistence-004, the PME imposes the name and the description on the PMC using *ICapeIdentification*.

Post-conditions: <The PMC is restored to its previous state and initialized> <The *Dirty* flag on the PMC is false>

Errors:

Uses: <UC-Persistence-006>

Extends: <None>

UC-Persistence-009 DETERMINE DIRTY STATUS OF PO.

Actor: <Persistence Host (PH)>

Priority: <Low>

Status: <This Use Case is fulfilled by the *ICapePersist* interface>

Pre-conditions: <The PO is Initialized>

Flow of events:

Basic Path:

Using *ICapePersist::IsDirty*, the PH queries the PO to determine the Dirty Status of the PO.

Post-conditions: <The PH knows the dirty status of the PO>

Errors: <None>

Uses: <None>

Extends: <None>

3. Analysis and Design

3.1 Overview

Originally, CAPE-OPEN utilized COM persistence interfaces. The use of more than one single agreed upon COM Persistence interface added significant complexity to CAPE-OPEN development. As a result, it was determined that CAPE-OPEN should develop its own Persistence Interface set, based on CAPE-OPEN specific data types. While this specification was primarily designed to save a flowsheet to permanent (disk) storage, it has been made general enough to allow its use in any instance where creating a duplication of a CAPE-OPEN object could be required. For this reason, the definition of a Persistable Object was expended beyond PMCs, and the Persistence Host (PH) could be any object that needs to create a duplicate copy of a CAPE-OPEN object, not just a PME.

In developing the CAPE-OPEN Persistence Interfaces, it was realized that Persistence could be used for more than just storing the PMC objects on a Flowsheet to a data file. For instance, one potential use is multiple threaded applications where persistence could be used to copy an object from one thread to another. For this reason, the Persistence Interfaces included here are focused on storing and retrieving the internal data of an object into any storage media, included application memory. Therefore, storage of that data onto operating system disk files, or other long-term storage is outside of the scope of the Persistence Interfaces. The task of deciding where to allocate memory is left to the Persistence Host.

The terminology used in this specification defines the object container hosting the objects as a Persistence Host. A PME is one example of a Persistence Host. It is possible to identify PMCs that contain Persistable Objects, and these Persistence Interfaces can be used to store and retrieve data from these secondary objects. A Persistable Object is any object that implements the *ICapePersist* Interface and can be stored or loaded from a data store.

The data store object is defined in terms of the functionality of a writer, placing the data to be stored into storage, and a reader, retrieving the stored data from storage. Each of these roles are encapsulated in separate interfaces, *ICapePersistWriter*, and *ICapePersistReader*. The writer provides the ability to store data using CAPE-OPEN datatypes in a dictionary-like structure. Each stored piece of data is placed into and retrieved from a key using a key name by the object being persisted. The specifics of the implementation of the writer and reader objects is left to the Persistence Host.

Compared to COM, the CAPE-OPEN Persistence Interface simplifies the persistence process by requiring the Persistable Object to support only one persistence interface instead of many. There are five possible COM persistence interfaces that may be implemented, and the Persistence Host needs to determine which of these interfaces it must use to ensure proper persistence of the Persistable Object. Furthermore, the CAPE-OPEN persistence mechanism allows standardization around modern serialization mechanisms utilized by various platforms such as .NET and Python, that store data in human readable formats such as XML or JSON.

CAPE-OPEN version 1.0 and 1.1 objects utilize Microsoft Component Object Model (COM) persistence mechanisms. The COMBIA layer includes the functionality to persist and depersist COM-based CAPE-OPEN objects through theses interfaces. Native COBIA implementations will utilize the interfaces described in this specification. In this way, PMEs that implement COBIA do not need to access legacy COM PMCs directly but can rely on COMBIA to handle these legacy components. Transitioning of CAPE-OPEN version 1.0 and 1.1 COM-based CAPE-OPEN objects to CAPE-OPEN version 1.2 persistence is presented in Appendix A.

3.2 Sequence diagrams

SQ-001: SAVE PERSISTABLE OBJECT

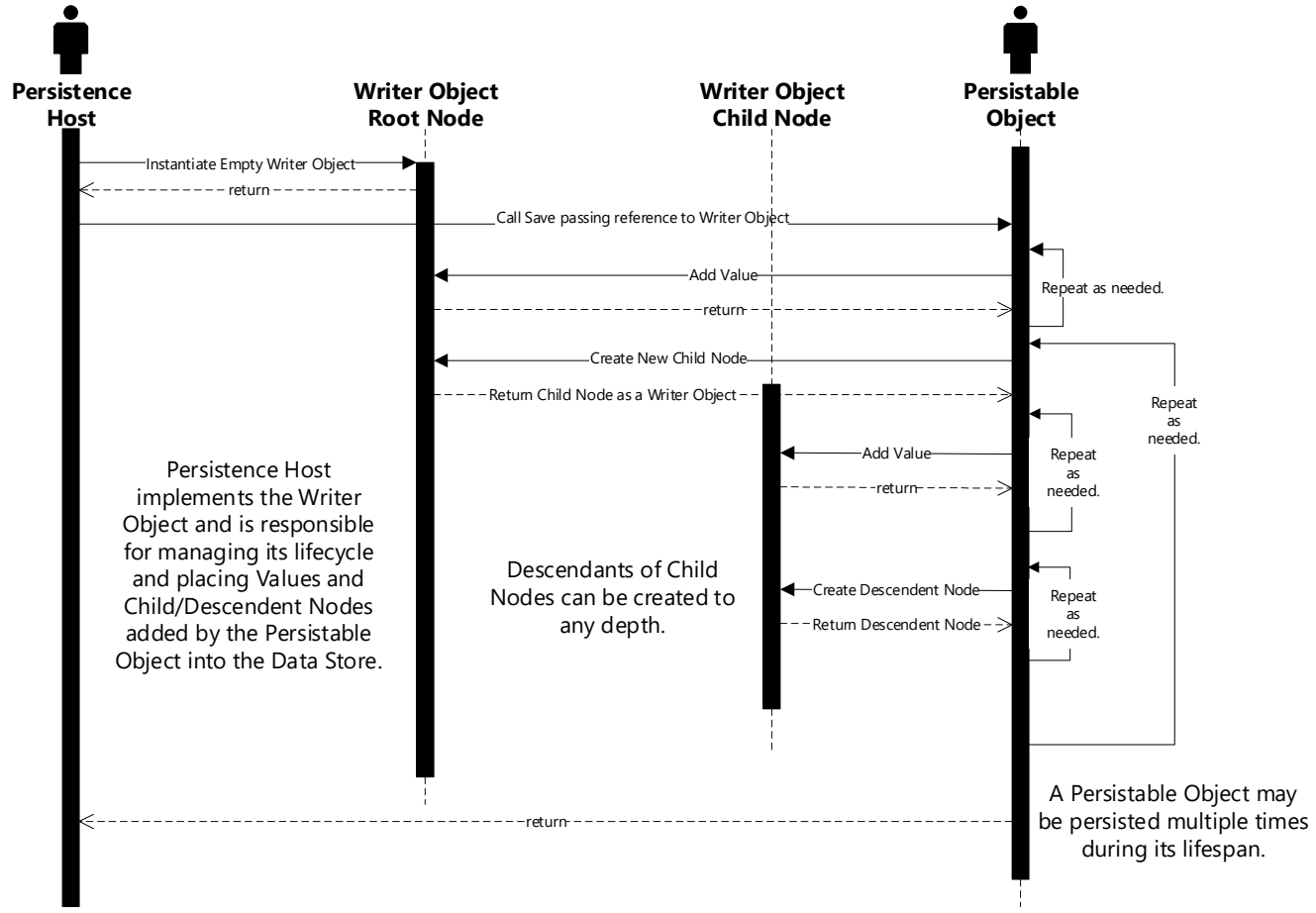


Figure 2 Save Persistable Object

SQ-002: RESTORE PERSISTABLE OBJECT

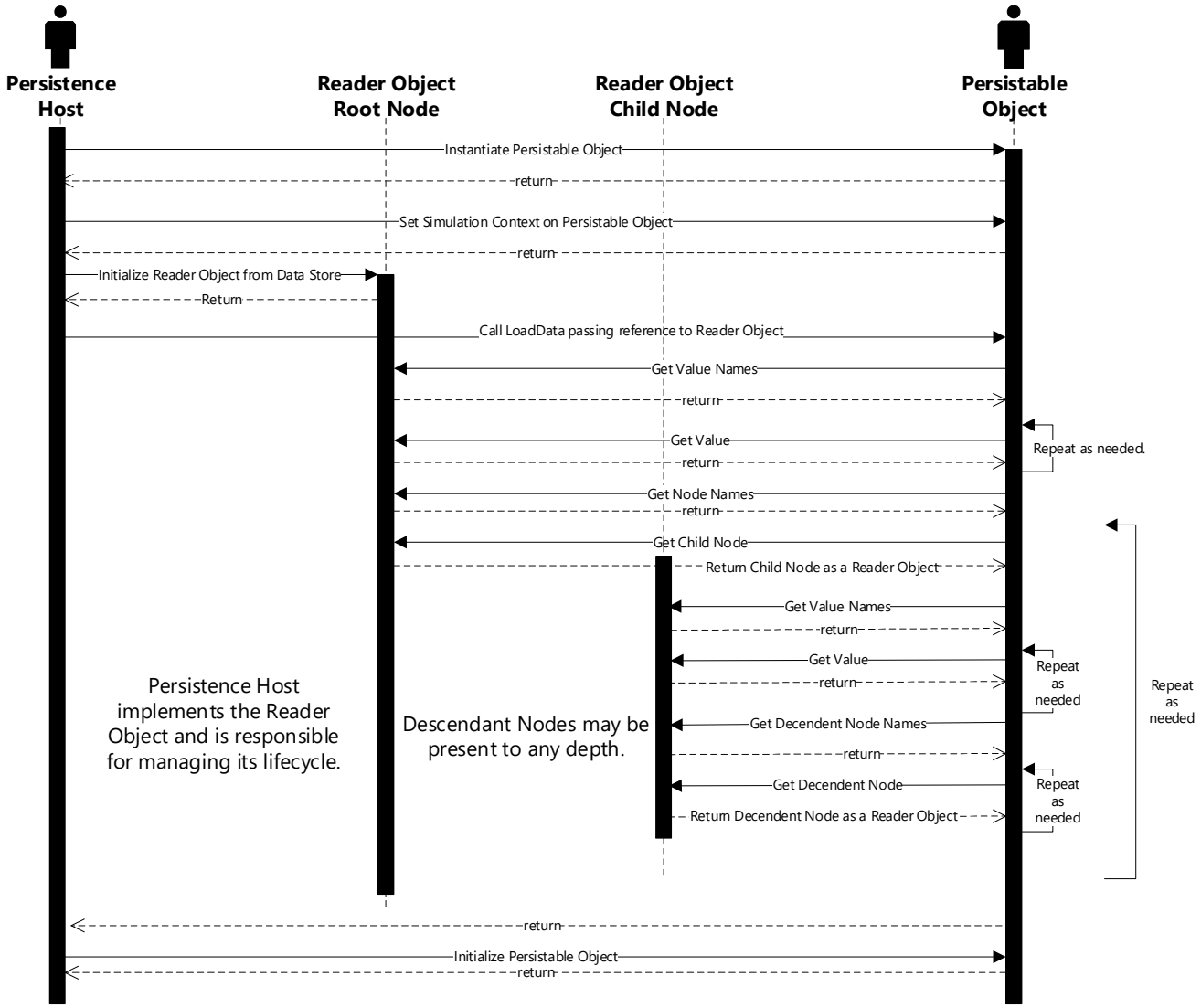


Figure 3. Restore Persistable Object

3.3 Interface diagrams

Figure **4** illustrates the core persistence interfaces:

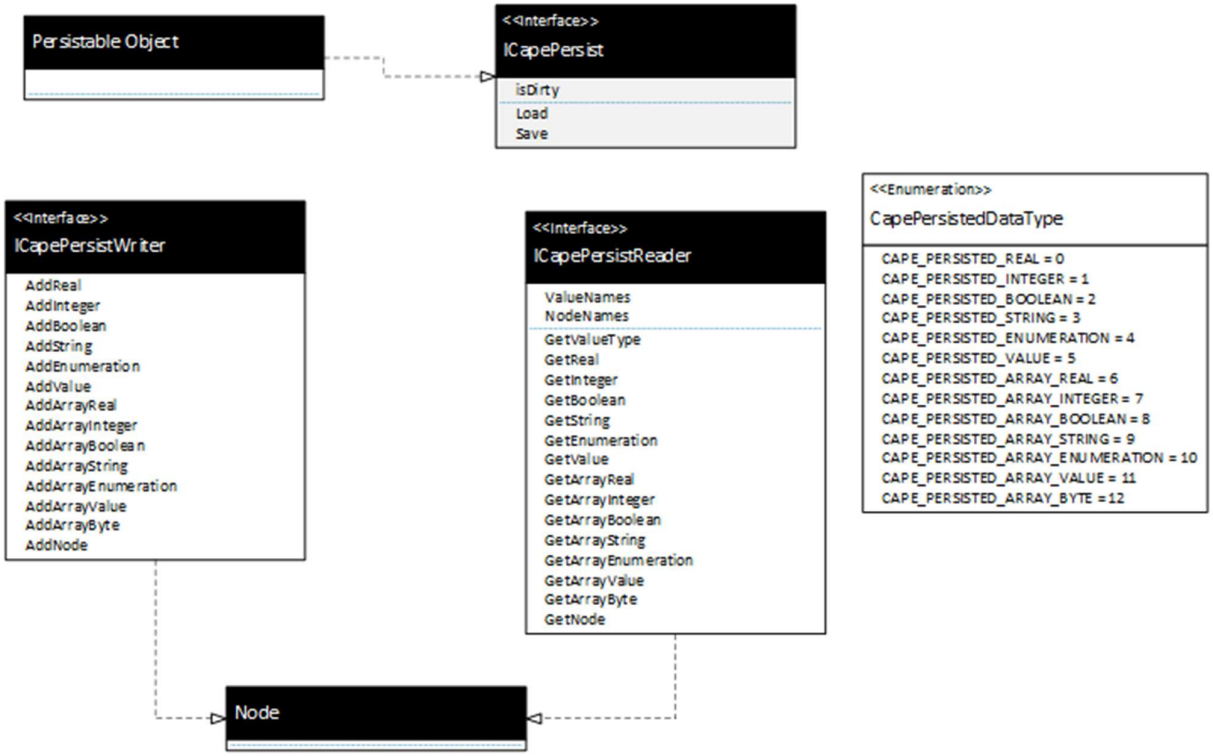


Figure 4. **Interface diagram**

3.4 State diagrams

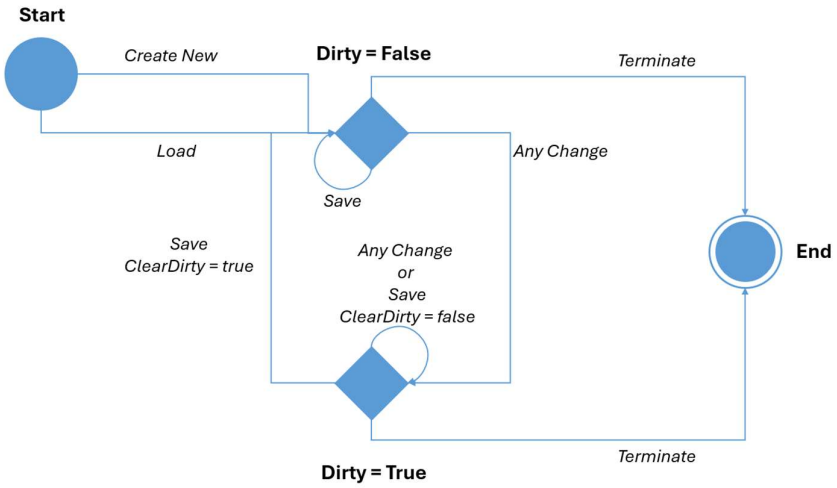
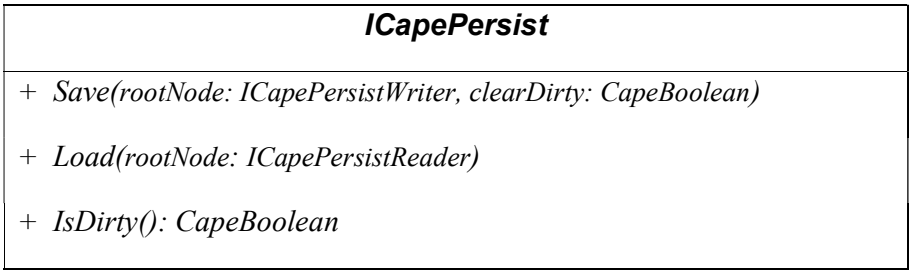


Figure 5. ***Dirty*** **State diagram**

3.5 Interfaces descriptions

3.5.1 ICapePersist

The *ICapePersist* interface is the basic interface implemented by a PO and contains the following methods:



Interface Name	*ICapePersist*
Method Name	Save
Returns	None

Description

Saves the PO to the specified root node. Clears the *dirty* flag for the PO based upon the value of *clearDirty*.

Arguments

Name	Type	Description
[in] *rootNode*	*ICapePersistWriter*	The writer of the root node used to store the object's data. Cannot be null.
[in] *clearDirty*	*CapeBoolean*	Clears the dirty flag after persistence if true.

Errors

- Any errors raised by the writer object

Notes

The *dirty* flag is maintained by the PO to inform the PH of whether the PO has been modified since the last time that *ICapePersist::Save* was invoked on the PO with the *clearDirty* flag set to true.

Interface Name	*ICapePersist*
Method Name	Load
Returns	None

Description

Loads the PO using object data previously placed into the data store.

Arguments

Name	Type	Description
[in] *rootNode*	*ICapePersistReader*	The Persistence Reader Object used to obtain persisted object data. Cannot be null.

Errors

- Any errors raised by the reader object,
- Any errors related to persistent content (e.g. persisted with a newer version of the software),
- PMC is already initialized

Notes

None.

Interface Name	*ICapePersist*
Property Name	IsDirty
Access Mode	Read
Type	Boolean

Description

Gets the dirty status of the PO. Returns true if the PO has been modified since it was last persisted and false if the PO has not been modified since it was last persisted or created.

Errors

No specific errors

Notes

None.

3.5.2 ICapePersistWriter

The *ICapePersistWriter* interface provides the PO with access to the PH's data storage object. The interface has the following methods:

ICapePersistWriter
+ *AddReal(valueName: CapeString, value: CapeReal)*
+ *AddInteger(valueName: CapeString, value: CapeInteger)*
+ *AddBoolean(valueName: CapeString, value: CapeBoolean)*
+ *AddString(valueName: CapeString, value: CapeString)*
+ *AddEnumeration(valueName: CapeString, value: CapeEnumeration)*
+ *AddValue(valueName: CapeString, value: CapeValue)*
+ *AddArrayReal(valueName: CapeString, value: CapeArrayReal)*
+ *AddArrayInteger(valueName: CapeString, value: CapeArrayInteger)*
+ *AddArrayBoolean(valueName: CapeString, value: CapeArrayBoolean)*
+ *AddArrayString(valueName: CapeString, value: CapeArrayString)*
+ *AddArrayEnumeration(valueName: CapeString, value: CapeArrayEnumeration)*
+ *AddArrayValue(valueName: CapeString, value: CapeArrayValue)*
+ *AddArrayByte(valueName: CapeString, value: CapeArrayByte)*
+ *AddNode(nodeName: CapeString): CapeInterface*

Interface Name	ICapePersistWriter
Method Name	AddReal
Returns	None

Description

Adds a *CapeReal* value to the current node.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeReal*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddInteger
Returns	None

Description

Adds a *CapeInteger* value to the current node.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeInteger*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddBoolean
Returns	None

Description

Adds a *CapeBoolean* value to the current node.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeBoolean*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddString
Returns	None

Description

Adds a *CapeString* value to the current node.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeString*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddEnumeration
Returns	None

Description

Adds a *CapeEnumeration*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeEnumeration*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),

- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddValue
Returns	None

Description

Adds a *CapeValue*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeValue*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddArrayReal
Returns	None

Description

Adds a *CapeArrayReal*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeArrayReal*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddArrayInteger
Returns	None

Description

Adds a *CapeArrayInteger*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeArrayInteger*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddArrayBoolean
Returns	None

Description

Adds a *CapeArrayBoolean*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeArrayBoolean*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddArrayString

Returns	None

Description

Adds a *CapeArrayString*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeArrayString*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddArrayEnumeration
Returns	None

Description

Adds a *CapeArrayEnumeration*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description

[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeArrayEnumeration*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddArrayValue
Returns	None

Description

Adds a *CapeArrayValue*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the value.
[in] *value*	*CapeArrayValue*	The value.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddArrayByte
Returns	None

Description

Adds a *CapeArrayByte*-valued data value to the current node of the writer object named *valueName*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the real-valued data object to be added to the current Writer node.
[in] *value*	*CapeArrayByte*	The value of the data.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

Interface Name	ICapePersistWriter
Method Name	AddNode
Returns	CapeInterface

Description

Adds a *child* node to the current node and returns a writer for the new node.

Arguments

Name	Type	Description
[in] *nodeName*	*CapeString*	The name of the node to be added to the tree.

Errors

- Errors related to the persistence media (e.g. out of storage space),
- Duplicate key name

Notes

None.

3.5.3 ICapePersistReader

The *ICapePersistReader* interface provides the PO with access to the PH's data storage object. The interface has the following properties and methods:

ICapePersistReader
+ *ValueNames: CapeArrayString*
+ *GetValueType: CapeArrayString*
+ *GetNodeNames: CapePersistedDataType*
+ *GetReal(valueName: CapeString): CapeReal*
+ *GetInteger(valueName: CapeString): CapeInteger*
+ *GetBoolean(valueName: CapeString): CapeBoolean*
+ *GetString(valueName: CapeString): CapeString*
+ *GetEnumeration(valueName: CapeString): CapeEnumeration*
+ *GetValue(valueName: CapeString): CapeValue*
+ *GetArrayReal (valueName: CapeString): CapeArrayReal*
+ *GetArrayInteger(valueName: CapeString): CapeArrayInteger*
+ *GetArrayBoolean(valueName: CapeString): CapeArrayBoolean*
+ *GetArrayString(valueName: CapeString): CapeArrayString*
+ *GetArrayEnumeration(valueName: CapeString): CapeArrayEnumeration*
+ *GetArrayValue(valueName: CapeString): CapeArrayValue*
+ *GetArrayByte(valueName: CapeString): CapeArrayByte*
+ *GetNode(nodeName: CapeString): CapeInterface*

Interface Name	ICapePersistReader
Property Name	ValueNames
Access Mode	Read Only
Returns	*CapeArrayString*

Description

Returns a *CapeArrayString* containing the names of the data values in the current *node*.

Arguments

None.

Errors

- Errors related to the persistence media (e.g. read error)

Notes

None.

Interface Name	ICapePersistReader
Property Name	NodeNames
Access Mode	Read Only
Returns	*CapeArrayString*

Description

Returns a *CapeArrayString* containing the names of the child nodes in the current node.

Arguments

None.

Errors

- Errors related to the persistence media (e.g. read error)

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetValueType
Returns	*CapePersistedDataType*

Description

Returns the data type of the value stored on the *valueName* node as a *CapePersistedDataType* enumeration value.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the Persistable Data Object.

Errors

- Incorrect, non-existing, key name

Notes

The *CapePersistedDataType* enumeration is defined in Table 1.

Table 1 CapePersistedDataType enumeration

Name	Value
CAPE_PERSISTED_REAL	0
CAPE_PERSISTED_INTEGER	1
CAPE_PERSISTED_BOOLEAN	2
CAPE_PERSISTED_STRING	3
CAPE_PERSISTED_ENUMERATION	4
CAPE_PERSISTED_VALUE	5
CAPE_PERSISTED_ARRAY_REAL	6
CAPE_PERSISTED_ARRAY_INTEGER	7
CAPE_PERSISTED_ARRAY_BOOLEAN	8
CAPE_PERSISTED_ARRAY_STRING	9
CAPE_PERSISTED_ARRAY_ENUMERATION	10
CAPE_PERSISTED_ARRAY_VALUE	11
CAPE_PERSISTED_ARRAY_BYTE	12

Interface Name	ICapePersistReader
Method Name	GetReal
Returns	*CapeReal*

Description

Gets the *CapeReal*-valued data from the *valueName* node of the *ICapePersistReader*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeReal*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetInteger
Returns	*CapeInteger*

Description

Gets the *CapeInteger*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeInteger*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetBoolean
Returns	*CapeBoolean*

Description

Gets the *CapeBoolean*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeBoolean*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetString
Returns	*CapeString*

Description

Gets the *CapeString*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeString*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetEnumeration
Returns	*CapeEnumeration*

Description

Gets the *CapeEnumeration*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeEnumeration*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetValue
Returns	*CapeValue*

Description

Gets the *CapeValue*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeValue*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetArrayReal
Returns	*CapeArrayReal*

Description

Gets the *CapeArrayReal*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeArrayReal*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetArrayInteger
Returns	*CapeArrayInteger*

Description

Gets the *CapeArrayInteger*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeArrayInteger*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetArrayBoolean
Returns	*CapeArrayBoolean*

Description

Gets the *CapeArrayBoolean*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeArrayBoolean*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetArrayString
Returns	*CapeArrayString*

Description

Gets the *CapeArrayString*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeArrayString*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetArrayEnumeration
Returns	*CapeArrayEnumeration*

Description

Gets the *CapeArrayEnumeration*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeArrayEnumeration*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetArrayValue
Returns	*CapeArrayValue*

Description

Gets the *CapeArrayValue*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeArrayValue*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetArrayString
Returns	*CapeArrayString*

Description

Gets the *CapeArrayString*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeArrayString*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetArrayByte
Returns	*CapeArrayByte*

Description

Gets the *CapeArrayByte*-valued data from the *node*.

Arguments

Name	Type	Description
[in] *valueName*	*CapeString*	The name of the node containing the desired *CapeArrayByte*-valued data object.

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Incorrect data type for key name.

Notes

None.

Interface Name	ICapePersistReader
Method Name	GetNode
Returns	*CapeInterface*

Description

Gets the *ICapePersistReader* of a child *node*.

Arguments

Name	Type	Description
[in] *nodeName*	*CapeString*	The name of the node

Errors

- Errors related to the persistence media (e.g. read error),
- Incorrect, non-existing, key name,
- Key refers to a value, not a node.

Notes

None.

3.6 Scenarios

Three different scenarios are presented for the use of the persistence interfaces, saving a flowsheet to file-based storage, retrieving a flowsheet for file-based storage, and copying an object from one location to another.

3.6.1 Save Flowsheet to Persistent Storage

Saving a simulation case is a common activity that may be performed by the user to allow them to return at some future time to a previously saved state of the simulation. Examples of situations where saving a flowsheet is useful include the flowsheet user saving the flowsheet at the end of a workday so that they may pick up where the left off the next day, saving the state prior to modifications to investigate how changing the flowsheet affects the modelled system's operation, or they may wish to archive the state for future reference. Saving a flowsheet is a distinct scenario from retrieving the flowsheet as the saved flowsheet need never be retrieved.

To save a flowsheet, the PME must save the information needed to restore the flowsheet to its current state. That includes the enumerating all of the PMCs present in the flowsheet, how to instantiate these PMCs, connectivity between PMCs, PMC configuration, and stream information. This interface specification focuses on allowing the flowsheeting environment acquire the PMC's internal data using a standardized process so that internal state is placed into the data store. The data stores for all the PMCs within the flowsheet are then consolidated and placed into the data file for placement into persistent storage. The data store can then be reconstituted from the information placed into the data file and use to restore the PMC to its current state at a later time. The PME is responsible for ensuring that all required information is stored in the flowsheet data file.

This scenario is the primary target addressed by this interface specification. Certain classes of PMCs, such as thermodynamic and physical property packages, can be created from a Manager Object, the configuration of which is local to the computer system at which the PMC is created. In order to facilitate a self-consistent storage mechanism, where a simulation involving such a PMC can be integrally stored and consistently be restored at another computer system, or at a later date at the same computer system after the manager configuration may have changed, the design allows for the configuration of the PMC to be stored inside the simulation document.

More-over, via the parameter common interface and the Edit method of the utilities common interface, the user can change the configuration of a PMC. Even if the user does not change the configuration of the PMC, e.g. because the PMC does not provide the means to do so, the internal configuration of the PMC may still be altered internally to the PMC itself. Examples of this include storage of the last solution, storage of an initial guess strategy and storage of textual reports. In order to keep the configuration of the PMC consistent between sessions of the simulation application, the design allows for the configuration of the PMC to be stored inside the simulation document.

3.6.2 Retrieve Flowsheet from Persistent Storage

Retrieving a flowsheet from persistent storage is the complementary action to saving the flowsheet to storage. While related, the scenarios are distinct in that saved flowsheets need not be restored, or a single saved flowsheet may be restored multiple times on multiple machines. That is, even though the act of saving a flowsheet is necessary for restoration of the flowsheet, there is no requirement regarding restoration. The persistence interfaces specified in this document are used to restore the PMCs to their internal states.

The restoration process will mirror the saving process. The PME will create instances of the required PMCs using information stored by the PME. The PMCs will be restored to their previously saved states using

persistence. The PME will restore connectivity between the PMCs and stream configurations. To restore an individual PMC to its previously saved state, the PME extracts the data store from the data file and provides the data store to the PMC using the load method.

3.6.3 Duplicating a PMC

Duplication of a PMC may be desired for various reasons. Duplication of a PMC involves the step of persisting the data of the existing PMC, creating a new, uninitialized instance of the PMC, setting the simulation context, depersisting the data into the new PMC and initializing the new PMC using *ICapeUtilities::Initialize*.

Valid scenarios in which duplication of a PMC may be required include Copy/Paste, Undo/Redo, creating multiple copies of a PMC for solving a flowsheet in parallel and creating a copy of a PMC, which is bound to a particular thread, in another thread.

As the Persistence Store in this scenario has an expected lifetime of the duration of the duplication action, a Persistence Store for this purpose would be optimized for performance rather than for example human readability. The entire Persistence Store may reside in volatile computer memory, and each persisted data item could be stored in binary form as opposed to serializing data to textual representation.

4. Specific Glossary Terms

5. Bibliography

1. “Open Interface Specification: Identification Common Interface”, CAPE-OPEN, 2003.
2. “Open Interface Specification: Manager Common Interface”, CAPE-OPEN, 2024

6. Appendix A - Transitioning between COM-based Persistence and CAPE-OPEN Persistence

6.1 CAPE-OPEN 1.2

This Appendix to the Persistence Specification Document provides an overview of a consistent mechanism to handle the transitioning of persistence from COM-based persistence used in CAPE-OPEN versions 1.x to CAPE-OPEN-based persistence introduced for COBIA in CAPE-OPEN version 1.2.

Under COM-based CAPE-OPEN 1.x, persistence was performed by allowing a CAPE-OPEN PMC to be saved and restored by the PH as part of saving or restoring a flowsheet using COM persistence interfaces.

One of the requirements for the development of COBIA was to provide interoperability between COM-based CAPE-OPEN version 1.1 PHs and COBIA POs, or COBIA-based PHs and COM-based CAPE-OPEN version 1.1 POs. COMBIA was developed to provide interoperability between these two environments, including persistence interoperability. The Interoperability section describes what COMBIA does when a COM-based CAPE-OPEN version 1.x object and a COBIA object are interoperating. The goal of COMBIA interoperability is for the process to be native to both the PME and PMC. For this to occur, COMBIA must structure the data in a consistent format.

The second consideration, called transitioning, is when either the PMC or PME are updated from COM-based persistence to persistence using the CAPE-OPEN persistence interfaces (CAPE-OPEN persistence). In the case of transitioning, either the PME, the PMC, or both have COM-based data that need to be updated or depersisted depending on how the data was last saved. After the transitioning object has been transitioned, it will use CAPE-OPEN persistence for future save- and load operations. The scenarios and transitioning requirements are described in the Transitioning section.

6.2 Future outlook

One of the design criteria of new interfaces introduced in COBIA, is that in future CAPE-OPEN versions, e.g. CAPE-OPEN 2.0 and higher, the COBIA and COM interfaces match 1:1; COM IDL for this purpose will be generated from the same source as COBIA IDL. It is anticipated that COM based CAPE-OPEN 2.0 and up is to use the new ICapePersist interface (and related) as specified in this document.

The information in this chapter focuses on interoperability between CAPE-OPEN 1.0/1.1 COM based implementations (using COM based persistence) and CAPE-OPEN 1.2 COBIA based implementations, using interfaces outlined in this document. The transitioning strategy makes use of some API functions supplied by COBIA. The exact formulation of future COM implementation is out of scope for this appendix.

6.3 Interoperability Scenarios

Since COBIA is platform neutral, a COBIA object cannot be aware of the existence of COM. As a result, the COM based persistence interfaces do not apply, and the CAPE-OPEN based persistence interfaces outlined in this document apply.

Backwards compatibility constraints placed on COBIA during its development have made it necessary for COBIA to provide a means for COBIA and COM to interoperate, including translation of COM-based CAPE-OPEN version 1.0/1.1 functionality into COBIA-based CAPE-OPEN 1.2 and higher functionality. For the most part, the entire interoperability can be accomplished without either COBIA-based or COM-based objects being aware of the existence of the other environment. However, in the case of persistence, this is not possible, because new COBIA based implementations that replace older COM based implementations need to deal with information previously persisted.

This section examines the persistence scenarios that are possible in CAPE-OPEN 1.0/1.1 and CAPE-OPEN version 1.2 and greater interoperation.

Table 2. Persistence Interoperability Scenarios.

PME	**PMC**	
	COM (1.0/1.1)	**COBIA (1.2)**
COM (1.0/1.1)	COM Persistence	Case 1
COBIA (1.2)	Case 2	CAPE-OPEN persistence

Case 1: COMBIA is acting as a PH for the PMC, which is acting like a PO. When the PME calls *IPersistStream::Save*, COMBIA will create a writer object that implements *ICapePersistWriter*, and present that to the PO. The PO then persists itself using the write object. When the PO has completed saving itself, COMBIA creates a name value pair having the name "*COMBIAPersistenceVersion*" on the root node and places the Data Store into either a COM Stream using the *IStream* interface or property bag by flattening the tree structure of the CAPE-OPEN version 2.0 Data Store. COMBIA data format is proprietary to COMBIA, and requires an API call to restore. When the PME requests that the PMC depersist, COMBIA extracts the Data Store from the COM stream and creates a reader object that implements *ICapeReader*. COMBIA then invokes the *ICapePersist::Load* method on the PO using the reader object.

Case 2: COMBIA acts as a PO, saving the data persisted by the PMC from a call to the *IPersistXXX::Save* method. The format of the persisted data is internal to COMBIA, but is can be indicated by the presence of a subnode named "*COBIA*" on the root node, with a value having the key name "*comType*". There are two potential options for the COM persistence interface used, *IPersistStream*, or *IPersistPropertyBag*, which results in two potential storage formats and interoperability protocols, as follows:

- PMC only supports *IPersistStream* or *IPersistStreamInit*: COMBIA creates an COM stream object that implements the *IStream* interface. COMBIA then invoked the *IPersistStream::Save* or *IPersistStreamInit::Save* method passing the *IStream* interface to the PMC for storage. COMBIA then saves the data stream to the writer object provided by the PH. When depersisting, COMBIA recreates the stream object from the reader object provided by the PH and calls *IPersistStream::Load* or *IPersistStreamInit::Load* method passing the *IStream* interface to the PMC.

- PMC Supports *IPersistPropertyBag*: COMBIA implements a Property Bag that implements *IPropertyBag*. COMBIA then invoked the *IPersistPropertyBag::Save* method passing the *IPropertyBag* interface to the PMC for storage. COMBIA then saves the properties from the property bag to the PH's writer object using the property name as the key name. When depersisting, COMBIA recreates the property bag object from the reader object provided by the PH and calls *IPersistPropertyBag::Load* method passing the *IPropertyBag* interface to the PMC.

6.4 Transitioning is a special case of interoperability scenarios where either the PME, PMC, or both are upgrading from COM-based persistence to CAPE-OPEN persistence. Transitioning is initiated when a PH requests a PO to depersist, but the previously stored data was persisted using COM mechanisms. Following transitioning, future persistence will be performed using the persistence interoperability scenario described above. Transitioning Scenarios

Transitioning occurs when either the PMC or the PME is updated from COM-based persistence to CAPE-OPEN-based persistence. The following describes the scenarios for transitioning either a PMC, PME, or both PMC and PME transitioned since the last time that the flowsheet was persisted.

6.4.1 PMC Transitioning

When a PMC transitions, the persistence data from COM may be in one of two formats:

1. Last saved to a COM Stream: The PO receives COM Stream transitioning data.
2. Last saved to a COM Property Bag: The name/value pairs in the reader reflect COM properties.

6.4.2 PME Transitioning

PME transitioning is a little more complicated than PMC transitioning as it is possible that the PMC had previously transitioned and COMBIA saved the Data Store to a COM Stream or COM Property Bag, as described in the interoperation section, above. As a result, the PME may have the following persistence data formats from COM:

1. Last saved to a COM Stream.
    a. COM Stream Data
    b. Data persisted to a COM Stream by COBIA.
2. Last saved as a COM Property Bag: The name/value pairs in the reader reflect COM properties.
    a. COM PropertyBag Data
    b. Data persisted to a COM PropertyBag by COBIA.

6.4.3 Both PMC and PME Transitioned since last persistence

Transitioning operations when both the PME or PMC are transitioning, should be consistent with transitioning operations when either the PME or PMC are the only component transitioning. As a result, the transitioning workflows need to be consistent.

6.5 COM/CAPE-OPEN Interoperability and Transitioning Persistence Data Formats

Transition format is an intermediate data format that is used for COM/CAPE-OPEN persistence interoperation and transitioning. Data is placed into transition format either by a PME when it transitions a PMC from COM-based persistence to CAPE-OPEN persistence or when a CAPE-OPEN PO is persisted using COMBIA for interoperability with a COM-based PME.

Transition format is a standardized layout of a COBIA persistence object, used to store data that was previously persisted via COM persistence. Transition format consists of an indicator value by which the depersisted object can detect that the persisted data is in transition format and indicates the type of COM data included. There are two different transition formats, one for stream data persisted to a COM stream using *IStream* and another for data persisted to a COM property bag using *IPersistPropertyBag* data.

COBIA provides two API methods for converting data to or from transition format. The first method, *DepersistPMCfromCOMdata* invoked by a transitioning PH to depersist a PO using reader object containing transition format data that detects whether the COM data was previously persisted from a COBIA PO to COM

persistence by COMBIA. The second COBIA persistence API routine is *DepersistFromTransitionFormat*, which is called by a PO to convert a reader object from COMBIA format to a reader object in transition format.

6.5.1 Transition format for stream data

The transition format for stream data applies to data that was previously stored by IPersistStream or IPersistStreamInit and stores the entire stream data as a binary array. A COBIA persistence object in transition format for stream data contains:

- A string indicator value with name “TransitionFormat” and string value “Stream”.
- The stream data with name “data” and a binary array value that corresponds to the entire COM stream data.

6.5.2 Transition format for property bag data

The transition format for stream data applies to data that was previously stored by IPersistPropertyBag, and stores the properties previously in the property bag in a particular node. A COBIA persistence object in transition format for property bag data contains:

- A string indicator value with name “TransitionFormat” and string value “PropertyBag”.
- A sub node with name “data”, containing all the values that were in the COM property bag.

6.6 Transitioning Requirements

Transitioning requires the PO and PH during the transition process. In addition, the persisted data needs to be transitioned from the COM-based stream or property bag to a CAPE-OPEN Data Store. The following describe each of these roles:

- **Stream-based Transitioning Data (SBTD):** Data persisted in COM using the *IPersistStream* or IPersistStreamInit interfaces. TOs that are being updated from Stream-based persistence must be able to restore itself using SBTD.
- **Property-based Transitioning Data (PBTD):** Data persisted in COM using the *IPersistPropertyBag* interface. The PH is responsible for transitioning PBTD from a property bag to the reader object. For Property Bag data, the PH exposes the property bag data to the PO as a CAPE-OPEN Reader Object, converting the properties into CAPE-OPEN key/value pairs.

Textual requirements are listed hereafter and referenced by mentioning the software component to which each requirement applies and its number in the global list of requirements. The requirements styled REQ-PO-xx refer to requirements on the PO, typically a primary PMC. Requirements styled REQ-PH-xx refer to requirements on the PH, the object requesting transitioning of the PO, typically the PME.

REQ-PH-01: The PH initiates the transitioning workflow for its own transitioning.

Rationale: the PH uses its internal machinery to restore data from its storage medium. In doing so, it can detect that the last time it saved its data to its storage medium, was either prior to or after transitioning. Hence, the

PH knows that the actions that follow are either for transitioning, or not. Transitioning workflow includes invoking API routines to convert from previously stored data to Transition Format (in case of transitioning to COMBIA) or translating previously stored data to transition format (in case of transitioning to COM based implementation of CAPE-OPEN persistence).

REQ-PO-02: A PO that previously supported COM-based persistence and now supports CAPE-OPEN persistence must support transitioning.

Rationale: Objects instantiated by CAPE-OPEN 1.2 or higher use CAPE-OPEN persistence. Objects that have previously been persisted using a different persistence mechanism must be able to be restored through the transitioning mechanisms described here for backwards compatibility.

REQ-PH-03: A PH that previously supported COM-based persistence and currently supports CAPE-OPEN persistence must support transitioning.

Rationale: Objects instantiated by CAPE-OPEN 1.2 or higher use CAPE-OPEN persistence. Objects that have previously been persisted using a different persistence mechanism must be able to be restored through the transitioning mechanisms described here for backwards compatibility.

REQ-PH-04: The PH provides SBTD to the PO for use in transitioning.

Rationale: SBTD is binary data that the PH does not know how to structure for use in a reader object. The PH must provide this data to the PO to decode and so that it may restore itself.

REQ-PO-05: The PO is responsible for restoring itself using SBTD.

Rationale: The PO knows the structure of information contained in SBTD.

In addition to the requirements on the PO and PH for transitioning, the creation of COBIA and COMBIA for interoperability between COM-based CAPE-OPEN 1.0/1.1 and COBIA-based CAPE-OPEN 1.2 and higher adds the following requirements:

REQ-COBIA-06: COBIA determines whether COMBIA is required to instantiate and/or provide interoperability between a COM-based PME and COBIA-based PMC, or a COBIA-based PME and COM-based PMC.

Rationale: COMBIA is used to instantiate PMCs registered with COBIA or is used to request COM instantiation of COM-based PMCs.

REQ-COMBIA-07: COMBIA acts as a PH when it is used to instantiate and depersist, or persist a CAPE-OPEN 1.2 or greater PMC.

Rationale: COMBIA is hosting the COBIA-based PMC.

REQ-COMBIA-08: When the COM-based PME invokes the Save method on a COM based persistence to persist a CAPE-OPEN 1.2 or higher-based PO, COMBIA acts as a PH and creates a writer object that implements ICapePersistWriter interface for use by the PO. The PO then uses the writer object for persistence.

Rationale: As the PH, COMBIA is responsible for providing the writer object to the PO.

REQ-COMBIA-9: When COMBIA is acting as a PH, COMBIA will only implements stream-based COM persistence, and persists and depersists the Data Store from COM-based PME using IPersistStream interface.

Rationale: The conforms to the minimal requirements for a PME for use in COM-based persistence. In addition, it simplifies the PME transitioning process.

REQ-COBIA-10: COMBIA will provide a mechanism to create a reader object that implements ICapePersistenceReader from SBTD persisted by COMBIA under REQ-COMBIA-10.

Rationale: COMBIA persists the data store from invocation of the CAPE-OPEN ICapePersistence::Save method into COM stream-based storage.

REQ-COMBIA-11: COMBIA acts as a PO when it is used by a COM-based PME to instantiate and depersist, or persist a COM CAPE-OPEN 1.0/1.1 PMC.

Rationale: COMBIA is hosting the COM-based PMC.

REQ-COMBIA-12: When the CAPE-OPEN 1.2 PME invokes the ICapePersist::Save method to persist a COM-based PO, COMBIA determines whether the COM-based PMC supports Property Bag or Stream based persistence. COMBIA will select Property bag persistence, if supported.

Rationale: Property bag persistence is similar to CAPE-OPEN persistence.

REQ-COMBIA-13: When COMBIA uses IPersistStream::Save to persist a COM-based PMC, the data is stored as SBTD.

Rationale: the PMC appears as COMBIA PMC to the PH, through the ICapePersist interface.

REQ-COMBIA-14: When COMBIA uses IPersistPropertyBag::Save to persist a COM-based PMC, the data is stored as PBTD.

Rationale: the PMC appears as COMBIA PMC to the PH, through the ICapePersist interface.

6.7 Use Cases:

In general, COMBIA internally handles the conversion of data between COM and CAPE-OPEN format during the interoperability cases. As a result, the interoperability cases do not alter the persistence behavior of a COM PME or PMC, or a COBIA PH or PO. For this reason, no additional Use Cases for the PME/PH or PMC/PO are associated with COM/COBIA interoperability.

Transition Scenario 1, has a possibility that the PO was previously a PMC that was previously saved using COM persistence interfaces. In this case, the stream data was saved into a COM Stream or Property Bag provided to the PO either directly by the PME or by COMBIA acting as the PME. This scenario introduces steps to extract the COM Stream or Property Bag data from one created by COMBIA. The depersistence process of a PO that may have previously been saved to a COM Stream or Property Bag covered in Use Case 1.

Transitioning Scenario 2 requires the PH to provide the PO with data that was previously provided to it through COM interfaces. The PH has no way to know whether the COM data was provided directly by COM-based PMC or by a PO interoperating through COMBIA. Further, the PH has no way to know whether COBIA will use COMBIA to provide interoperability support for the PO if it is a COM-based PMC. For these reasons, COBIA provides a Persistence API Routine that detects whether the COM data was persisted by COMBIA and converts the data, if required, to restore the PO/PMC.

UC-Transition-001 RESTORING A PO FOR WHICH A PREVIOUS VERSION USED COM-BASED PERSISTENCE (TRANSITIONING SCENARIO 1).

Actor: <**PO**>

Priority: <**High**>

Status: <Supported through DepersistFromTransitionFormat API routine >

Pre-conditions: <The PO has been instantiated> <The PO is COBIA based and will therefore use CAPE-OPEN persistence> <The PH has invoked the ICapePersist.Load method on the PO, providing a reader object that may be in transition format>

Flow of events:

- The PO calls the *DepersistFromTransitionFormat* method passing the reader object as the reader parameter and a valid pointer for a transition format reader as the transitionData parameter.
- The PO depersists itself using UC-Persistence-008 in one of the following ways:
    - If the return value from invoking *DepersistFromTransitionFormat* is true. The object returned using the transitionData out parameter is used as the reader object in performing UC-Persistence-008, where the data provided by the reader is in transition format (see section 6.5).
    - If the return value of *DepersistFromTransitionFormat* is false. The reader is not in transition format and can be used in UC-Persistence-008.

Post-conditions: <The PO is depersisted.>

Errors: <None>

Uses: <UC-Persistence-008>

Extends: <None>

UC-Transition-002 RESTORING A PO BY A PME FOR WHICH A PREVIOUS VERSION USED COM-BASED PERSISTENCE (TRANSITIONING SCENARIO 2).

Actor: <**PH**>

Priority: <**High**>

Status: < Supported through *DepersistPMCFromTransitionFormat* API routine >

Pre-conditions: < PME has instantiated the PO><Flowsheet was last persisted using COM-based persistence.>

Flow of events:

- The PH instantiates the PO.
- The PH acquires the object storage data last saved by the PME.
- The PH places the restoration data for each PO into a reader object in transition format. Conversion of the previously stored data to transition format is the responsibility of the PH (see section 6.5 for further information).
- The PH restores the PO by invoking the *DepersistPMCFromTransitionFormat* method passing the PO as the persistedObject argument and reader object as the transitioningData argument.

Post-conditions: <The PO has been depersisted and is ready to be initialized by the PH>

Errors: <None.>

Uses: <None.>

Extends: <None.>

6.8 COBIA API Persistence Routines

COBIA API Routine	
Method Name	*DepersistPMCFromTransitionFormat*
Returns	*void*

Description

This function helps a COBIA PH transition from COM persistence to CAPE-OPEN persistence. When transitioning, the PH creates an instance of the PO and then places the data last stored using COM-based persistence into a Reader Object in transition format. The PH invokes this method which determines whether the COM data were last stored by COMBIA-based persistence interoperation. In this case, the persisted data store is restored to a reader object and used to restore the PO; otherwise, the method restores the PO from transition format.

After this call is successfully completed, if the PO is a primary PMC object, the PH invokes *ICapeUtilities::Initialize* on the restored PO to complete CAPE-OPEN initialization of the PO.

Arguments

Direction	Type	Name	Description
in	*ICapePersist**	*persistedObject*	The object that is being depersisted.
in	*ICapePersistenceReader**	*transitionReader*	Reader object in transition format populated by the PH.

Exceptions

COBIAERR_NoSuchInterface: The PO does not provide the proper persistence interface.

Any other Error raised by the PO during Depersistence: The error condition can be obtained directly from the PO.

Notes

None.

COBIA API Routine	
Method Name	*DepersistFromTransitionFormat*
Returns	*CapeBoolean*

Description

This method is invoked by a PO that may have previously persisted itself though COM persistence interfaces, when it existed as a COM-based PMC. If the reader object is in transition format, this method is invoked to determine whether the reader object contains a CAPE-OPEN data store last persisted by COMBIA. If the contents of the reader object are a CAPE-OPEN data store the method creates and returns a reader object for the PO to depersist from using CAPE-OPEN depersistence methods and returns true.

If the method returns *false*, the transition format data was last persisted by the PMC using COM-based persistence.

At the start of depersistence, PO that may have been previously persisted using COM persistence invokes this method to convert COMBIA formatted persistence data into transition format data.

```
Load(reader);
CapePersistenceReader transitionFormat;
if (DepersistFromTransitionFormat(reader, transitionFormat)
      LoadFromTransitionFormat(transitionFormat);
else
      LoadFromNewFormat(reader);
```

Loading from reader:

1. PO can test for TF
    a. True: Was data saved by COMBIA?
        i. True: needs API routine to restore the reader object from COMBIA data.
        ii. False: last saved by PMC in COM format. Handle TF data itself.
    b. False: restore from reader (not transitioning)

Direction	Name	Type	Description
in	*reader*	*ICapePersistenceReader**	The reader past to the persisted object as an argument to *ICapePersist::Load* method.
out	*transitionedReader*	*ICapePersistenceReader**	A Reader Object restored from the transition format data if the return value is true, otherwise *CapeNull*.

Errors

None.

Notes

None