amdocs rating

Rating 6.0
Pricing Engine APIs

Document Information

Software Version:	**6.0**
Publication Date:	**January 2005; updated May 2006 for SP7**
Catalog Number:	**212526**

Contents

1. INTRODUCTION

This document describes the APIs provided by the Pricing Engine.

The Pricing Engine is the brain of the Amdocs Rating component. It performs all the business logic required by Rating, Prepaid Server, or any other price-related processor. The Pricing Engine receives an event from an invoker and according to the subscriber associated with the event, qualifies the event to the relevant pricing items. Once the event has been qualified, the Pricing Engine processes it – calculating the charges and allowances – and updates the data storage with the results.

To provide this functionality, the Pricing Engine uses the reference data defined in the Product Catalog with all the dynamic definitions and business logic.

The Pricing Engine is implemented as a callable thread-safe library with a well-defined interface. Currently, the following Amdocs Billing platform components and processes invoke methods contained in the library:

- Product Catalog – Ensures the validity of the Implementation Repository.
- Acquisition and Formatting (A&F) – Sends events to the Rating process and utilizes the event APIs to create and populate the events.
- Postpaid Rater – Receives the events prepared by A&F and invokes the Pricing Engine for processing.
- Prepaid Server – Prepares the events for authorization or rating; and invokes the Pricing Engine for processing.
- Outcollect – Retrieves events that have been rated for outcollect subscribers in order to format them and send them to the clearinghouse for collection. The outcollect process utilizes the query entities APIs.
- Extract Process – Retrieves events or performance indicators (PIs) for downstream processing such as Billing. These extract processes utilize the query entities APIs.
- Rerating process – Retrieves events that have been rated erroneously, together with their performance indicators, and re-initializes the system to a point where rerating of the events brings about the required result. These rerating processes utilize query entities APIs to retrieve the relevant population, as well as Rating APIs.
- Usage queries – Retrieves events or performance indicators for online Customer Management. These processes utilize the query entities APIs.
- PI Maintenance – Purges self-contained (non-cross-cycle) performance indicators (PIs) that have been archived to the persistent database and do not have any effect on future transactions. This process may also construct next-cycle-month PIs that are cross-cycle in nature and for

which no next-cycle event has been received. These processes use the archiving APIs.

Target Audience

This document assumes that the reader is familiar with the object-oriented approach and with the C++ language.

Terminology

The following specialized terms are used in this document.

Term	Definition
API	Application program interface
Envelope	Any process that activates and controls the PE
IR	Implementation repository
PC	Product Catalog
PE	Pricing Engine
PI	Performance indicator

2. Pricing Engine Life Cycle Management

This chapter describes the various ways in which the Pricing Engine can be initialized and released. All the methods described in this chapter belong to the Pricing Engine interface class.

Initializing the Pricing Engine

The Pricing Engine must be initialized before it can be used. The initialization is separated into two levels.

- Process level initialization

  A process that needs to work with the Pricing Engine and must initialized in one of its initialization modes before it can be used. This initialization level is performed only once for a process, and **it is not thread-safe**.

  At this level, there are two initialization modes:

  - Data Dictionary initialization

    In this mode, the Implementation Repository is loaded, processed and optimized. It is used when data services are required (i.e., to format data into entity objects and vice versa).

  - Full initialization

    This mode initially performs the Data Dictionary initialization and then it loads data objects (e.g., Product Catalog, Cycle Management, etc.) required for the pricing process. It is used when event processing is required.

- Thread-level initialization

  A thread that executes Pricing Engine activities must be initialized before it can be used. This initialization level is performed only once for a thread and must occur in that thread. The Pricing Engine initializes the thread local storage (TLS) with details required by its activities.

The following sections describe the various initialization methods.

Data Dictionary Initialization: initDataDictionary

note *This method will be deprecated in the next Pricing Engine version. Therefore, it should no longer be used*

initDataDictionary is a static method used for initializing the Pricing Engine in a Data Dictionary initialization mode. It is a process-level initialization that must be called prior to any other Pricing Engine request and only once.

The Pricing Engine is initialized with the effective Implementation Repository that resides in the Oracle database specified by the connect string.

The data services expect the names of entities and the names of their attributes as defined in the Implementation Repository. The data services are case insensitive with respect to these names and the space character is treated as an underscore character (i.e., **GSM voice** is converted to **gsm_voice**). This mode is used by Acquisition & Formatting (A&F) to format events to Rating. Case insensitive is the default mode of work.

When the engine initialization fails (FALSE return status), the envelope cannot continue to work with the engine and will abort. Failure of initialization in this mode is usually due to an invalid Implementation Repository (IR). The recommended response is to roll back to the previous IR version, run the envelope again, and check the IR outside the production environment. For additional information, check the errors reported by the Pricing Engine; they may indicate the cause of failure.

Syntax

```
static bool PricingEngine::initDataDictionary(const
UtString& i_connectString, bool i_caseInsensitive = true,
long i_futureVersionLoadPeriod = 1, long
i_versioningCoveragePeriod = 1, const UtString&
i_eventTableStatus = DEFAULT_EVENT_TABLE_STATUS, const
cyclesVec& i_cycles = cyclesVec())
```

Parameters

Name	Direction	Type	Description
i_connectString	In	UtString	The connect string to the Oracle database where Implementation Repository resides.
i_caseInsensitive	In	Bool	The case insensitivity indicator. By default, TRUE (case sensitive).
i_futureVersionLoadPeriod	In	Long	Future version scope in days.
i_versioningCoveragePeriod	In	Long	Past version scope in days.

Name	Direction	Type	Description
i_eventTableStatus	In	const UtString&	Event table status: DYNAMIC – BLOB representation of the rated event attributes. HYBRID – Part of the attributes will be stored as BLOB, and part as fielded attrinutes. FIELDED – F ielded representation of the rated event attrinutes.
i_cycles	In	const cyclesVec&	Vector of billing cycles. By default, empty.
	Return Value	Bool	The initialization status. TRUE = successful initialization.

Data Dictionary Initialization: initDataDictionary

initDataDictionary is a static method used for initializing the Pricing Engine in a Data Dictionary initialization mode. It is almost identical to the method described in the previous section. The only difference is that in this method all input parameters are grouped into the InitParameters class and two additional parameters can be provided: encoding and partition ID.

Syntax

```
InitParameters(const UtString& i_connection,
                  const UtString& i_encoding =
DEFAULT_ENCODING,
            bool i_caseInsensitive = false,
            long i_futureVersionLoadPeriod = 1,
            long i_versioningCoveragePeriod = 1,
            long i_partitionID = -1,
const UtString&  i_eventTableStatus =

DEFAULT_EVENT_TABLE_STATUS,
                        const cyclesVec& i_cycles =
cyclesVec());


static bool PricingEngine::initDataDictionary(const
InitParameters& i_params);
```

Parameters

Name	Direction	Type	Description
`i_params`	In	`const InitParameters&`	All input parameters as were described in the previous method.
	Return Value	Bool	The case insensitivity indicator. By default, TRUE (case sensitive).

Full Initialization: init

init is a static method used for initializing the Pricing Engine in a full initialization mode (process-level initialization). It must be called only once, prior to any other Pricing Engine request .

The Pricing Engine is initialized with the configuration file, which is located in the path provided to the method. The configuration file contains all the information required for the initialization. This mode is used when event processing is required (e.g., Prepaid and Postpaid).

When the Pricing Engine initialization fails (FALSE return status), the envelope cannot continue to work with the engine and will abort. There are several reasons that may lead to initialization failure. The following list gives possible reasons and recommended actions. The reason for a particular failure can be found in the Pricing Engine error report.

- Configuration invalid – The envelope should fix the configuration file and run again.
- Implementation invalid – The envelope should roll back to the previous implementation version (IR +PC entities) and run again. Then check the implementation outside the production environment.
- Reference data cannot be loaded – The reference database should be fixed.

Syntax

```
static bool PricingEngine::init(const UtString&
i_configuration,
const cyclesVec& i_cycles = cyclesVec())
```

Parameters

Name	Direction	Type	Description
i_configuration	In	const UtString&	The configuration file path and name.
`i_cycles`	In	`const cyclesVec&`	Vector of billing cycles. By default, empty.

Name	Direction	Type	Description
	Return Value	Bool	The initialization status. TRUE = successful initialization.

initRerate

note *This method will be deprecated in the next PE version. the ordinary init should be used for initialization, and setCycleInfo – for the cycle information.*

initRerate is a static method used for initializing the Pricing Engine in full mode for a rerate process. It receives the cycle information on which the rerate process is operated.

Syntax

```
static bool PricingEngine::initRerate(const UtString&
i_configFile, long i_cycleCode, long i_cycleMonth, long
i_cycleYear)
```

Parameters

Name	Direction	Type	Description
i_configFile	In	const UtString&	The configuration file name including its full path.
i_cycleCode	In	Long	The cycle code for Rerate.
i_cycleMonth	In	Long	The cycle month for Rerate.
i_cycleYear	In	Long	The cycle year for Rerate.
	Return Value	Bool	The initialization status. TRUE = Successful initialization.

setCycleInfo

This method is used to set the cycle code, month, and year the PE is going to work on. It is used when re-rating is required. It sets an indication that re-rate is in progress, and the cycle month and year are not taken from the cycle state table, but are determined by the input parameters of the method. This method can be recalled whenever cycle information must be changed.

Syntax

```
static bool PricingEngine::setCycleInfo(long i_cycleCode,
long i_cycleMonth, long i_cycleYear)
```

Parameters

Name	Direction	Type	Description
i cycleCode	In	Long	The cycle code for Rerate.
i cycleMonth	In	Long	The cycle month for Rerate.
i cycleYear	In	Long	The cycle year for Rerate.

Reinitializing the Pricing Engine

Re-initialisation allows runtime incremental update of the reference data (Implementation Repository, Product Catalog, Auxiliary Object versions, and Billing cycle information) used by the PE. For example, it is not required to stop Rater (Post-paid or Prepaid) to load/unload the reference data. Re-initialisation can be performed in a separate thread. This allows incremental loading/unloading of the PE reference data to/from the memory at runtime, without affecting other threads. Each running thread can continue to run using the previous version of the reference data, until a new version becomes available and the thread decides to switch to the new reference storage using the etReferenceDataStorage method (see description below).

Subsequent sections describe the various re-initialization methods.

Data Dictionary Re-initialization: reinitDataDictionary

reinitDataDictionary is a static method used for re-initializing the Pricing Engine in a Data Dictionary initialization mode. It should be called after data dictionary initialization of Pricing Engine only. It is called each time new incremental reference data must be loaded to PE.

The Pricing Engine is re-initialized with the same configuration file as the previously called initDataDictionary method. All the information required for the re-initialization is already loaded to the Pricing Engine memory by the method. When the execution of the method is completed, the new, freshly loaded reference data becomes available to the event processing. The entire thread running reinit is ready to use it. All other threads should call setReferenceDataStorage to switch to the new reference data storage. Otherwise, they will continue using old reference data.

When the Pricing Engine initialization fails (FALSE return status), the envelope can continue to work with the engine using the previously loaded reference storage. Failure of initialization in this mode is usually due to invalid Implementation Repository (IR). For additional information, check the errors reported by the PE; they may indicate the cause of failure.

Syntax

```
static bool PricingEngine::reinitDataDictionary(const
cyclesVec& i_cycles = cyclesVec())
```

Parameters

Name	Direction	Type	Description
i_cycles	In	const cyclesVec&	Vector of billing cycles. By default, empty.
	Return Value	Bool	The initialization status. TRUE = successful initialization.

Full Re-initialization: reinit

reinit is a static method used for re-initializing the Pricing Engine in a full initialization mode. This mode is used when event processing is required (e.g., Prepaid and Postpaid). It should be called after full initialization of

Pricing Engine only. It is called each time new incremental reference data must be loaded to PE.

The Pricing Engine is re-initialized with the same configuration file as the reviously called init method. All the information required for re-initialization is already loaded to the Pricing Engine memory by the init method. When the execution of the method is completed, the new, freshly loaded reference data becomes available to the event processing. The entire thread running reinit is ready to use it. All other threads should call setReferenceDataStorage to switch to the new reference data storage. Otherwise they will continue using the old reference data.

When the Pricing Engine initialization fails (FALSE return status), the envelope can continue to work with the engine using the previously loaded reference storage. There are several reasons that may lead to initialization failure. The reason for a particular failure can be found in the PE error report. Following is the list of possible reasons and recommended actions:

- Implementation invalid – The envelope should roll back to the previous implementation version (IR +PC entities) and run again. Check the implementation outside the production environment.
- Reference data cannot be loaded – The reference database should be fixed.

Syntax

```
static bool PricingEngine::reinit(const cyclesVec&
i_cycles = cyclesVec())
```

Parameters

Name	Direction	Type	Description
i_cycles	In	const cyclesVec&	Vector of billing cycles. By default, empty vector by default.
	Return Value	Bool	The initialization status. TRUE = successful initialization.

Thread-level Initialization: initThread

initThread is a static method used for initializing the thread local storage (TLS) of the thread in which Pricing Engine is to be activated. It must be called prior to any other Pricing Engine request and after the process initialization. It must be activated only once for a thread.

The method initializes the TLS with details required by the Pricing Engine activities.

The method receives the thread sequence number, which is a number between 1 and the maximum number of threads that can be attached to the Pricing Engine (not the thread ID). The Pricing Engine expects its clients to enumerate each thread according to this rule. The sequence is used by the Pricing Engine to enable optimized access to the Usage database.

Syntax

```
static void PricingEngine::initThread(long
i_threadNumber)
```

Parameters

Name	Direction	Type	Description
i_threadNumber	In	Long	The thread sequence number.

Thread-level Initialization: initThread

initThread is a static method used for initializing the thread local storage (TLS) of the thread in which a Pricing Engine is to be activated. It must be called prior to any other Pricing Engine request and after the process initialization. It must be activated only once for a thread.

The method initializes the TLS with details required by the Pricing Engine activities.

The method receives the thread sequence number, which is a number between 1 and the maximum number of threads that can be attached to the Pricing Engine (it is not the thread ID). The Pricing Engine expects its clients to enumerate each thread according to this rule. The sequence is used by the Pricing Engine to enable an optimized access to the Usage database.

Syntax

```
static void PricingEngine::initThread(long
i_threadNumber)
```

Parameters

Name	Direction	Type	Description
i_threadNumber	In	Long	The thread sequence number.

Thread-level Re-initialization: setReferenceDataStorage

setReferenceDataStorage is a static method used for setting reference data storage for the current process thread. It is part of thread initialization, and must be called before the initThread method. setReferenceDataStorage can be called each time after the reference data is loaded to Pricing Engine by one of the re-initialization methods to switch to the new reference data storage.

Currently, the method should be used with a default value of NULL for its input parameter, i.e., it always takes the last loaded reference data storage. The return value indicates whether the reference data storage was switched to the new one or not.

Syntax

```
static bool PricingEngine::setReferenceDataStorage(const
IReloadableDataObject* i_pPEReferenceDataStorage = 0)
```

Parameters

Name	Direction	Type	Description
i_pPEReferenceDataStorage	In	const IReloadableDataObject*	

Name	Direction	Type	Description
	Return Value	bool	TRUE – The thread reference data storage was updated to new one. Otherwise,FALSE.

Releasing the Pricing Engine

The Pricing Engine has two levels of release:

- Process-level release

  Before termination of a process, all resources that were acquired, during the process level initialization, should be released. This release level is used also when it is required to perform a re-initialization of the engine.

- Thread-level release

  Before termination of a thread, all resources that were allocated, during the thread level initialization, must be released.

releaseThread

This static method is used to release the resources acquired by the Pricing Engine during a thread-level initialization. The method must be activated in the thread before it exits.

Syntax

```
static void PricingEngine::releaseThread()
```

Parameters

None

release

This static method is used to release the resources acquired by the Pricing Engine during the process level initialization. The method is called when the envelope (the process that hosts the Pricing Engine) aborts or exits or requires re-initialization. The method receives a parameter which specifies whether the release is required for re-initialization or for termination.

Syntax

```
static void PricingEngine::release(ReleaseMode
i_releaseMode = RELEASE_REINITIALIZE)
```

Parameters

Name	Direction	Type	Description
i_releaseMode	In	ReleaseMode	The release mode. Valid Values: ▪ RELEASE_REINITIALIZE ▪ RELEASE_TERMINATE By default, the release mode is RELEASE_REINITIALIZE.

3. Creating Entities

Pricing Engine entities can be identified in two ways:

- Entity type and entity name. The entity type can be one of the following:
    - Event
    - Performance indicator (PI)
    - Customer parameter
    - Customer offer
    - Customer offer parameter
    - Cycle change
    - External record

  Entity name specifies an entity in a specific entity type (e.g., *Event*, *GSM voice*.). When referring to a specific entity type, its name will be referred to as the type (e.g., event of type *GSM voice*).
- TypeID – A unique identifier generated by the Product Catalog for each defined entity.

In general, an entity object can be created in three modes for different purposes of use:

- Initialized

  In this mode, the entity object is initialized and it expects to be populated attribute by attribute.
- Initialized by binary data

  In this mode, the entity object is created and initialized by the given binary data.
- Not Initialized

In this mode, the entity object is not initialized and it expects to be populated by binary data.

This section describes the methods of the Pricing Engine used to create interface objects to entities.

Creating Interface Object to Events

This section describes the methods used by the Pricing Engine to create interface objects to events.

createEvent

This static method is used for creating an entity object, which is used as an interface to an event for setting and getting the attribute values of the event. The method receives the event type for which the interface is required.

When the create operation fails (e.g., attempt to create an instance of an event with an unknown type), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEvent(const
UtString& i_eventType, EntityInitMode i_entityInitMode =
INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_eventType	In	const UtString&	The event type for which the interface is created.
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid Values: ▪ INITIALIZE_ENTITY ▪ DON'T_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle".

createEvent(Effective date)

This static method is for creating an entity object that will be used as an interface for setting and getting the attribute values of an event. The method receives the event type for which the interface is required, and the effective date, enabling the entity to be created in the structure that is effective for the required date.

When the create operation fails (for example, having attempted to create an instance of an event with a type that is unknown in the specified date's effective version), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEvent(const
UtString& i_eventType, const DateType& i_effectiveDate,
EntityInitMode i_entityInitMode = INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_eventType	In	const UtString&	The event type for which the interface is to be created.

Name	Direction	Type	Description
i_effectiveDate	In	const DateType&	The effective date according to which structure of the event is to be selected. (The correct event structure is the one that is in the version that is effective for the specified date.)
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ▪ INITIALIZE_ENTITY ▪ DON'T_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createEvent(Binary)

This static method is used for creating an entity object, initialized with the given binary data, which is used as an interface to an event. The interface is used for setting and getting the attribute values of the event. The method receives the Event type for which the interface is required. In addition, it receives the binary details.

When the Create operation fails (e.g., attempt to create an instance of an event with an unknown type), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEvent(const
UtString& i_eventType, char *i_pData, long i_dataSize,
BinaryContents i_binaryContents)
```

Parameters

Name	Direction	Type	Description
i_eventType	In	const UtString&	The event type for which the interface is created.
i_pData	In	char *	A pointer to the binary data.
i_dataSize	In	Long	The size of the data in the binary data buffer.

Name	Direction	Type	Description
i_binaryContents	In	BinaryContents	The binary contents which can be: ▪ ENTITY_DYNAMIC_ DATA ▪ ENTITY_OBJECT_ DATA
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createEventByTypeID

This static method is used for creating an entity object, which is used as an interface to an event. The interface is used for setting and getting the attribute values of the event. The method receives the type ID of the event for which the interface is required.

When the Create operation fails (e.g., attempt to create an instance of an event with an unknown type ID), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEventByTypeID(long
i_eventTypeID, EntityInitMode i_entityInitMode =
INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_eventTypeID	In	long	The event type ID for which the interface is created.
i_entityInitMode	In	EntityInitMode	The entity initialization mode which can be one of ▪ INITIALIZE_ENTITY ▪ DON'T_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createEventByTypeID(Effective Date)

This static method is for creating an entity object that will be used as an interface for setting and getting the attribute values of an event. The method receives the type ID of the event for which the interface is required, and the effective date, enabling the entity to be created in the structure that is effective for the required date.

When the create operation fails (for example, having attempted to create an instance of an event with a type ID that is unknown in the specified date's effective version), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEventByTypeID(long
i_eventTypeID, const DateType& i_effectiveDate,
EntityInitMode i_entityInitMode = INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_eventTypeID	In	long	The type ID of the event for which the interface is to be created.
i_effectiveDate	In	const DateType&	The effective date according to which structure of the event is to be selected. (The correct event structure is the one that is in the version that is effective for the specified date.)
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DON'T_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createEventByTypeID(binary)

This static method is used for creating an entity object, initialized with the given binary data, which is used as an interface to an event. The interface is used for setting and getting the attribute values of the event. The method receives the type ID of the event for which the interface is required. In addition, it receives the binary details.

When the Create operation fails (e.g., attempt to create an instance of an event with an unknown type ID), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEventByTypeID(long
i_eventTypeID, char *i_pData, long i_dataSize,
BinaryContents i_binaryContents)
```

Parameters

Name	Direction	Type	Description
i_eventTypeID	In	Long	The event type ID for which the interface is created.
i_pData	In	Char *	A pointer to the binary data.
i_dataSize	In	Long	The size of the data in the binary data buffer.
i_binaryContents	In	BinaryContents	The binary contents which can be: ▪ ENTITY_DYNAMIC_ DATA ▪ ENTITY_OBJECT_ DATA
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

Creating Interface Object to Performance Indicator

This section describes the methods used by the Pricing Engine to create an interface object to a performance indicator.

createPI

This static method is used to create an entity object, which is used as an interface to a performance indicator (PI). The interface is used for setting and getting the attribute values of the performance indicator. The method receives the type of the performance indicator for which the interface is required.

If the Create operation fails (e.g., an attempt to create an instance of a performance indicator with an unknown type), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createPI(const UtString&
i_name, EntityInitMode i_entityInitMode =
DONT_INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_piType	In	const UtString&	The performance indicator type for which the interface is created.
i_entityInitMode	In	EntityInitMode	The PI initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DONT_INITIALIZE_ ENTITY By default, the init mode is DON'T_INITIALIZE_ ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createPI(Binary)

This static method is used to create an entity object, initialized with the given binary data, which is used as an interface to the performance indicator (PI). The interface is used for setting and getting the attribute values of the performance indicator. The method receives the type of the performance indicator for which the interface is required and the binary data. In addition, it receives the binary details.

If the Create operation fails (e.g., an attempt to create an instance of a performance indicator with an unknown type), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createPI(const UtString&
i_piType, char *i_pData, long i_dataSize, BinaryContents
i_binaryContents)
```

Parameters

Name	Direction	Type	Description
i_piType	In	long	The performance indicator type for which the interface is created.
i_pData	In	char *	A pointer to the binary data.

Name	Direction	Type	Description
i_dataSize	In	long	The size of the data in the binary data buffer.
i_binaryContents	In	BinaryContents	The binary contents which can be: ■ ENTITY_DYNAMIC_DATA ■ ENTITY_OBJECT_DATA
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createPIByTypeID

This static method is used to create an entity object, which is used as an interface for setting and getting the attribute values of the performance indicator. The method receives the type ID of the performance indicator for which the interface is required.

If the Create operation fails (e.g., an attempt to create an instance of a performance indicator with an unknown type ID), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createPIByTypeID(long
i_piTypeID, EntityInitMode i_entityInitMode =
DONT_INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_piTypeID	In	Long	The performance indicator type ID for which the interface is created.
i_entityInitMode	In	EntityInitMode	The PI initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DONT_INITIALIZE_ENTITY By default, the init mode is DONT_INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note *The Pricing Engine allocates the entity object internally, and it is the caller’s responsibility to free it or recycle it. For more information, see "Entity life cycle"*

createPIByTypeID(Binary)

This static method is used to create an entity object, initialized with the given binary data, which is used as an interface for setting and getting the attribute values of a performance indicator. The method receives the type ID of the performance indicator for which the interface is required. In addition, it receives the binary details.

If the Create operation fails (e.g., an attempt to create an instance of a performance indicator with an unknown type ID), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createPI(long i_piTypeID,
char *i_pData, long i_dataSize, BinaryContents
i_binaryContents)
```

Parameters

Name	Direction	Type	Description
i_piTypeID	In	long	The performance indicator type ID for which the interface is created.
i_pData	In	char *	A pointer to the binary data.
i_dataSize	In	long	The size of the data in the binary data buffer.
i_binaryContents	In	BinaryContents	The binary contents which can be: ■ ENTITY_DYNAMIC_ DATA ■ ENTITY_OBJECT_DATA
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note *The Pricing Engine allocates the entity object internally, and it is the caller’s responsibility to free it or recycle it. For more information, see "Entity life cycle"*

Creating Interface Object to Customer Offer Parameters

This section describes the methods used by the Pricing Engine to create an interface object to Customer Offer Parameters.

createCustomerOfferParameters

This static method is used to create an entity object, which is used as an interface for setting and getting the attribute values of the Customer Offer Parameters. The method receives the desired initialization mode.

If the Create operation fails, a Null value is returned.

Syntax

static Entity * PricingEngine::createCustomerOffer Parameters(EntityInitMode i_entityInitMode = INITIALIZE_ENTITY)

Parameters

Name	Direction	Type	Description
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ▪ INITIALIZE_ENTITY ▪ DONT_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

The Pricing Engine allocates the Entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see" in Chapter 8 ("Entity APIs").

Creating Interface Object to Customer Offers

This section describes the methods of the Pricing Engine used to create an interface object to customer offers.

createCustomerOffers

This static method is used to create a customer offers object, which is used as an interface to the customer offers. The interface is used for setting and getting the attribute values of the event. The method receives the desired initialization mode.

If the Create operation fails, a Null value is returned.

Syntax

```
static Entity *
PricingEngine::createCustomerOffers(EntityInitMode
i_entityInitMode = INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DONT_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

Creating Interface Object to External Record

This section describes the methods used by the Pricing Engine to create an interface object to the Customer offers.

createExternalRecord

This static method is used to create an entity object, which is used as an interface to the external record. The interface is used for setting and getting the attribute values of the external record. The method receives the external record type and desired initialization mode.

When the create operation fails (e.g., attempt to create an instance of an external record with an unknown type), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createExternalRecord(const
UtString& i_externalRecordType, EntityInitMode
i_entityInitMode = INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DONT_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

Creating Interface Object to Cycle Change

This section describes the method used by the Pricing Engine to create an interface object to the Cycle Change.

createCycleChange

This static method is used to create an entity object that is used as an interface to Cycle Change. This interface is used for setting and getting the attribute values of Cycle Change. The method receives the desired initialization mode.

When the Create operation fails (e.g., attempt to create an instance of an External Record of an unknown type), a Null value is returned.

Syntax

```
static Entity* createCycleChange( EntityInitMode
i_initMode  = INITIALIZE_ENTITY);
```

Parameters

Name	Direction	Type	Description
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DONT_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle".

Creating Interface Object to External Record

This section describes the methods used by the Pricing Engine to create an interface object to the External Record.

createExternalRecord

This static method is used to create an entity object that is used as an interface to the External Record. This interface is used for setting and getting the attribute values of the External Record. The method receives the External Record type and desired initialization mode.

When the Create operation fails (e.g., attempt to create an instance of an External Record of an unknown type), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createExternalRecord(const
UtString& i_externalRecordType, EntityInitMode
i_entityInitMode = INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_externalrecordType	In	const UtString&	The type of the External Record for which the interface is created.
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DONT_INITIALIZE_ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createExternalRecord(Binary)

This static method is used to create an entity object initialized with the given binary data, which is used as an interface to the external record. The interface is used for setting and getting the attribute values of the external record. The method receives the external record type. In addition, it receives the binary details.

When the create operation fails (e.g., attempt to create an instance of an external record with an unknown type), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createExternalRecord(const
UtString& i_externalRecordType, char *i_pData, long
i_dataSize, BinaryContents i_binaryContents)
```

Parameters

Name	Direction	Type	Description
i_externalRecord Type	In	const UtString&	The External Record type for which the interface is created.
i_pData	In	char *	A pointer to the binary data.
i_dataSize	In	long	The size of the data in the binary data buffer.
i_binaryContents	In	BinaryContents	The binary contents which can be: ■ ENTITY_DYNAMIC_ DATA ■ ENTITY_OBJECT_ DATA
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createExternalRecordByTypeID

This static method is used to create an entity object, which is used as an interface for setting and getting the attribute values of the external record. The method receives the type ID of the external record for which the interface is required. In addition, the method receives the desired initialization mode.

When the create operation fails (e.g., attempt to create an instance of an external record with an unknown type ID), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createExternalRecord(long
i_externalRecordTypeID, EntityInitMode i_entityInitMode =
INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_externalRecordTypeI D	In	long	The External Record type ID for which the interface is created.
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ ENTITY ■ DONT_ INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createExternalRecordByTypeID(Binary)

This static method is used to create an entity object, initialized with the given binary data, which is used for setting and getting the attribute values of the external record. The method receives the type ID of the external record for which the interface is required. In addition, it receives the binary details.

When the create operation fails (e.g., attempt to create an instance of an external record with an unknown type ID), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createExternalRecord(long
i_externalRecordTypeID, char *i_pData, long i_dataSize,
BinaryContents i_binaryContents)
```

Parameters

Name	Direction	Type	Description
i_externalRecordTypeI D	In	Long	The External Record type ID for which the interface is created.
i_pData	In	char *	A pointer to the binary data.
i_dataSize	In	Long	The size of the data in the binary data buffer.

Name	Direction	Type	Description
i_binaryContents	In	BinaryContents	The binary contents which can be: ■ ENTITY_ DYNAMIC_DATA ■ ENTITY_OBJECT _DATA
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

Creating Interface Object to Entity (Generic API)

This section describes the generic methods used by the Pricing Engine to create an interface object to an entity.

createEntity

This static method provides a generic way to create an entity object, which is used as an interface to an entity of one of the following entity types: event, performance indicator, external record, customer offers and customer parameters. The interface is used for setting and getting the attribute values of the entity. The method receives the entity type and name of the required entity for which the interface is required. In addition the method receives the desired initialization mode.

When the create operation fails (e.g., attempt to create an instance of an entity with an unknown type and name), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEntity(const
UtString& i_entityType, const UtString& i_entityName,
EntityInitMode = INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_entityType	In	const UtString&	The type of the entity for which the interface is created.
i_entityName	In	const UtString&	The name of the entity for which the interface is created.

Name	Direction	Type	Description
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ ENTITY ■ DONT_ INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createEntity (binary)

This static method provides a generic way to create an entity object initialized with given binary data, which is used as an interface to an entity of one of the following entity types: event, performance indicator, external record, customer offers and customer parameters. The interface is used for setting and getting the attribute values of the entity. The method receives the type and name of the required entity for which the interface is required. In addition, the method receives the binary details.

When the create operation fails (e.g., attempt to create an instance of an entity with an unknown type and name), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEntity(const
UtString& i_entityType, const UtString& i_entityName,
char *i_pData, long i_dataSize, BinaryContents
i_binaryContents)
```

Parameters

Name	Direction	Type	Description
i_entityType	In	const UtString&	The type of the entity for which the interface is created.
i_entityName	In	const UtString&	The name of the entity for which the interface is created.
i_pData	In	char *	A pointer to the binary data.
i_dataSize	In	long	The size of the data in the binary data buffer.

Name	Direction	Type	Description
i_binaryContents	In	BinaryContents	The binary contents which can be: ■ ENTITY_DYNAMIC_ DATA ■ ENTITY_OBJECT_ DATA
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createEntityByTypeID

This static method provides a generic way to create an entity object, which is used as an interface to an entity of one of the following entity types: event, performance indicator, external record, customer offers and customer parameters. The interface is used for setting and getting the attribute values of the entity. The method receives the type ID of the entity for which the interface is required. In addition, the method receives the desired initialization mode.

When the create operation fails (e.g., attempt to create an instance of an entity with an unknown type ID), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEntity(long
i_entityTypeID, EntityInitMode = INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_entityTypeID	In	long	The type ID of the entity for which the interface is created.
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DONT_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

note *The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"*

createEntityByTypeID(Effective date)

This static method is for creating an entity object that will be used as an interface for setting and getting attribute values from entities of various entity types. The entity types are event, performance indicator, external record, customer offers, and customer parameters. The method receives the type ID of the entity for which the interface is required, and the effective date, enabling the entity to be created in the structure that is effective for the specified date.

In addition, the method receives the desired initialization mode.

When the create operation fails (for example, having attempted to create an instance of an event with a type ID that is unknown in the specified date's effective version), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEntity(long
i_entityTypeID, const DateType& i_effectiveDate,
EntityInitMode = INITIALIZE_ENTITY)
```

Parameters

Name	Direction	Type	Description
i_entityTypeID	In	long	The type ID of the event for which the interface is to be created.
i_effectiveDate	In	const DateType&	The effective date according to which structure of the event is to be selected. (The correct event structure is the one that is in the version that is effective for the specified date.)
i_entityInitMode	In	EntityInitMode	The entity initialization mode. Valid values: ■ INITIALIZE_ENTITY ■ DONT_INITIALIZE_ ENTITY By default, the init mode is INITIALIZE_ENTITY.
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

createEntityByTypeID(binary)

This static method provides a generic way to create an entity object initialized with a given binary data, which is used as an interface to one of the following entity types: event, performance indicator, external record, customer offers and customer parameters. The interface is used for setting and getting the attribute values of the entity. The method receives the type ID of the required type for which the interface is required. In addition, the method receives the binary details.

When the create operation fails (e.g., attempt to create an instance of an entity with an unknown type ID), a Null value is returned.

Syntax

```
static Entity * PricingEngine::createEntity(long
i_entityTypeID, char *i_pData, long i_dataSize,
BinaryContents i_binaryContents)
```

Parameters

Name	Direction	Type	Description
i_entityTypeID	In	Long	The type ID of the entity for which the interface is created.
i_pData	In	char *	A pointer to the binary data.
i_dataSize	In	long	The size of the data in the binary data buffer.
i_binaryContents	In	BinaryContents	The binary contents which can be: ▪ ENTITY_DYNAMIC_ DATA ▪ ENTITY_OBJECT_DATA
	Return Value	Entity *	The entity object. The method returns Null if the operation fails.

The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"

4. PROCESSING EVENTS

In general, the Pricing Engine can process events in two modes.

- Stateless – In this mode, the Pricing Engine receives all the data it requires for event processing from its invoker. It accesses neither the usage database nor the customer database during event processing.
- Stateful – In this mode, the Pricing Engine accesses the usage database and the customer database during event processing. Therefore, it must receive references to these databases.

This chapter describes the Pricing Engine methods used for event processing and the required preliminaries prior to the processing itself. In addition it also describes a number of classes, might passed to event processing in order to optimize qualification time and to enable post event process intervention.

Stateless Processing of an Event

This section describes how the Pricing Engine activates stateless processing.

process

This method is used to activate the stateless processing of an event. The event entity is in put to the method and contains updated fields, which are part of the result of processing the event.

This method has the same name as the standard event processing method; however, it is differentiated from the standard method by its different set of parameters.

The method receives a customer offers entity and a customer parameters entity. These entities are directly used by the Pricing Engine instead of fetching them from the customer database. Note that these parameters are optional and can be NULL (either one or both of them); in which case, no customer offers or customer parameters are available to the event processing logic.

The method receives a vector of performance indicator (PI) external record names. The PI entities generated during the processing logic as defined in event processing output are extracted to external records. The name of each required external record is transferred as a parameter. The resulting external record entities are returned in the output PI entity vector parameter. Note that if the PI external record names parameter is null, then the PI entities are available as output.

The PI extract flow is as follows:

- For every PI entity in the input list:
  - For every mapping case defined for the associated pricing item type (PIT)

- If the external record defined in the mapping case belongs to list of external record names, then:
  - Extract the PI to the external record.
  - Add the external record to the result list.

This method receives a vector of event external record name(s). The updated event entity will be extracted to the external records which the external record names parameter references. The resulting external record entities are returned in the output event entity vector parameter. Note that if the event external record names parameter is null, then no extract of the event entity is performed.

The event extract flow is as follows:

- For every mapping case defined for the event entity.
  - If the external record defined in the mapping case belongs to a list of external record names, then:
    - Extract the event to the external record.
    - Add the external record to the result list.

The method receives an object that implements a method to be activated at the end of the processing (postProcess hook method).

The method receives a qualification cookie, which identifies a qualification result-set (a set of items that were qualified in the past for a specific event). This parameter enables the use of this result-set instead of performing the qualification phase again on the event.

The method returns a status, which can be:

- PROCESS_SUCCESS – The event has been processed successfully.
- PROCESS_FATAL – A server problem was detected during event processing. The process (job) must abort.

Syntax

```
static EventProcessStatus PricingEngine::process(Entity
*io_pEvent, Entity* i_pCustomerOffers,
Entity*i_pCustomerParameters, STD_NAMESPACE
vector<Entity*>* i_pVecCustomerOfferParameters, const
STD_NAMESPACE vector<UtString>*
i_pVecPIExternalEntityNames, STD_NAMESPACE
vector<Entity*>& o_pVecPI, const STD_NAMESPACE
vector<UtString>* i_pVecEventExternalEntityNames,
STD_NAMESPACE vector<Entity*>& o_pVecEvent,
IPostEventProcess* i_pCallback = 0, QualificationCookie
*&io_pCookie = QualificationCookie::default());
```

Parameters

Name	Direction	Type	Description
io_pEvent	In/out	Entity *	The Event entity pointer.
i_pCustomer Offers	In	Entity *	The Customer Offers entity pointer.

Name	Direction	Type	Description
i_pCustomer Offers	In	Entity *	The Customer Parameters entity pointer.
i_pVecCusto merOfferPara meters	In	STD_NAMESPACE vector<Entity*>*	The Customer Offer Parameters vector.
i_pVecPI External Record Names	In	const STD_NAMESPACE vector<UtString>* i_pVecPIExternal Records	A vector of external record names to be used for extracting the generated PIs to external records. If this parameter is NULL, then the generated PIs themselves appear in the output entity list.
o_pVecPI	Out	STD_NAMESPACE vector<Entity*>& o_pVecPI	A vector of external record or PI entities that resulted from the event processing logic. Note that the vector contains PI entities if i_pVecPIExternalEntity Names is NULL and external record entities otherwise.
i_pVecEvent External Record Names	In	const STD_NAMESPACE vector<UtString>* i_pVecEventExternal Records	Used for mapping the Event entity to external record(s). If this parameter is NULL, then no extract of the event to external record(s) is performed.
o_pVecEvent	Out	STD_NAMESPACE vector<Entity*>& o_pVecEvent	A vector of external record entities that are extracted from the Event entity. Note that an extract of the event to external record(s) is made only if the i_pVecEventExternalEntity Names parameter is not NULL.
i_pCallback	In	IPostEventProcess*	A pointer to an object derived from the IPostEventProcess class, which implements the postProcess hook method. The parameter has a NULL default value that specifies that a callback does not exist.
io_pCookie	In/Out	QualificationCookie* &	■ The qualification cookie. By default the qualification cookie is ignored.

note *The Pricing Engine allocates the entity object internally, and it is the caller's responsibility to free it or recycle it. For more information, see "Entity life cycle"*

Preparing a Database Connection

In order to optimize the time for database operations used for the stateful event processing, the Pricing Engine prepares its SQL statements (parsing them using the database engine) and caches the parsed statements for further use.

Pricing Engine APIs that involve database operations receive objects containing the cached statements.

It is advised that this preparation be performed only once for a database connection and be pooled and cached by the Pricing Engine envelope (the Pricing Engine's host process).

prepareConnection

This static method is used to prepare a database connection to be used later by the Pricing Engine for event processing, query and extract. The database connection is received as an object of DBConnection type. The DBConnection type is the base for all database connection types (e.g., ODBCConnection, OracleConnection, etc.). The caller of this method needs to create an object of the specific database connection required with the relevant connection context parameters (e.g., ODBC connection context is defined by two parameters – environment handle and connection handle) and, afterward, pass it as parameter to the method.

A database connection can be prepared for the following types (currently, all of the types except for None are only supported for the ODBC connection):

- Customer

  The database connection is prepared with the SQL statements required to access the customer's data.

- Usage

  The database connection is prepared with the SQL statements required to access the performance indicator (PI), rated event data and rejected event data.

- Customer and Usage

  The database connection is prepared with the SQL statements required to access customer, PI, rated event data and rejected event data.

- None

  The database connection does not include prepared statements. Used for query and extract.

The method returns an interface pointer of IDBContext type, which encapsulates the prepared SQL statements for the specified connection.

Syntax

```
static IDBContext
*prepareConnection(acm::common::io::IConnection*
i_pDBConnection, DBContext::DBContextType i_dbContextType
= NO_TYPE)
```

Parameters

Name	Direction	Type	Description
i_pDBConnection	In	acm::common::io::IConnection*	The database connection for which the SQL statement should be prepared.
dbContextType	In	DBContext::DBContextType	Database context type for which the connection should be prepared. It can be one of: ■ CUSTOMER ■ USAGE ■ CUSTOMER_ USAGE ■ NO_TYPE
	Return Value	IDBContext *	The pointer to the object that caches the prepared SQL statement for the specified connection. In case of an error, the return value is NULL.

note

The IDBContext pointer, which is returned by the prepare method, points to an object allocated by the Pricing Engine. Callers of this method are responsible to free the object when they have finished using it.

Processing an Event

This section describes how the Pricing Engine activates stateful processing.

process

This static method is used to activate the stateful Pricing process on an event. It is also used to rerate events.

The method receives the Customer database and Usage database contexts to which the database operation is applied. It assumes that these database contexts have the correct database context type. The Pricing Engine produces an error if it receives a database context prepared incorrectly (e.g., a database context that was prepared with DBContextType::None cannot be used as a Customer database context).

The method receives output parameters that store the number of records dispatched by the processed event.

The method receives an object that implements a method to be activated at the end of the processing (postProcess hook method).

The method receives a qualification cookie, which identifies a qualification result set (a set of items that were qualified in the past on a specific event), and uses this result set instead of re-qualifying the event.

The method also receives a Boolean indication that specifies whether the work should be updated to the data store (not committed). The postProcess hook method is activated prior to the update.

The method returns a status, which can be:

- PROCESS_SUCCESS – The event has been processed successfully.
- FATAL – A severe problem was detected during event processing. The process (job) should be aborted.
- REJECT – The event has been rejected because of business reasons or extension function failure. The process (job) must roll back the transaction (the transaction context might be inconsistent) and must activate the reject API method to report the event to the Error system.
- RETRY – Event processing might lead to a deadlock or incorrect results; therefore it should be retried. The transaction must be rolled back and the event processing should start again.

Syntax

```
static EventProcessStatus PricingEngine::process
(IDBContext& i_usageDBContext, IDBContext&
i_customerDBContext, Entity *io_pEvent,
IPostEventProcess* i_pCallback = 0, QualificationCookie
*&io_pCookie = QualificationCookie::default(), bool
i_persistWork = true);
```

Parameters

Name	Direction	Type	Description
i_usageDB Context	In	IDBContext&	The database context prepared for the usage database.
i_customer DBContext	In	IDBContext&	The database context prepared for the customer database.
io_pEvent	In/out	Entity *	The event entity pointer.
o_dispatched Records	Ot	Long	The number of dispatched records.

Name	Direction	Type	Description
i_pCallback	In	IPostEventProcess*	A pointer to an object derived from the IPostEventProcess class, which implements the postProcess hook method. The parameter has a NULL default value that specifies that a callback does not exist.
io_pCookie	In/Out	QualificationCookie *&	The qualification cookie. By default, the qualification cookie is ignored.
i_persist Work	In	Bool	The indication whether to update the usage database context with modifications performed on the event and related performance indicators.
	Return Value	EventProcessStatus	The event processing status: ■ PROCESS_SUCCESS ■ FATAL ■ REJECT ■ RETRY

note

The same database context can be used both for the customer and usage database.

process and releaseReservation

These static methods are used for online charging. In the online charging model, the Pricing Engine needs to process:

- Reserve requests before the call/session starts and during the call/session
- Debit request of the same call/session after part/all of the call/session ends
- Release request

The Pricing Engine manages Performance Indicators, which might need to be updated during a reserve request and to be rolled back before processing the next reserve request, debit, or release request in the same session. The decision whether or not to update the PI values due to a reserve request is a business decision. The PE supports the following business decisions: update PI s of all PIT roles, update PIs of specific PIT roles, or not update any PIs.

process

This method receives the Customer database and Usage database contexts to which the database operation is applied. It assumes that these database contexts have the correct type. The Pricing Engine produces an error if it receives a database context prepared incorrectly (e.g., a database context that was prepared with DBContextType::None cannot be used as a Customer database context).

The method receives the associated session identification, the period (in seconds) after which the session reservation should be released, and the type of reservation request. Pricing Engine supports three request types:

- **Reserve Request –** The following two reserve options are supported:
  - **Full Amount Reserve Request** – Reserve request of the amount since the beginning of the session (i.e., each reserve request arrives with the previous quota reserved amount and the current reserved amount for this session)
  - **Quota Reserve Request** – Reserve request of only the next quota reserved amount that needs to be authorized.
- **Debit Request –** The following two debit options are supported:
  - **Partial Debit Request** – Debit request that releases part of the reservation (only the debit amount) previously requested for the session
  - **Full Debit Request –** Debit request that releases all reservations previously requested for the session
- **Release Request –** This request releases all reservation previously requested for the session with no further processing.
- In the context of PI reservation, the event processing consists of the following steps: Perform the qualification processing and initialization phase. For every qualified PI:
  - Release expired reservations.
  - Locate the reservation of the processed session.
  - In case of partial debit request, save the reserve enabled PI attribute values in a temporary variable.
  - If the request is not a quota reserve, for every reserve enable PI attribute, add its reserved amount to its current value.
  - In case of a full amount and quota reserve request, save the reserve enabled PI attribute values in a temporary variable.
- Execute computation handlers for each of the qualified pricing items according to their roles and polices:
  - In case of a full amount reserve request, for every qualified reserve enabled PI attribute:
    - Subtract its current value from the old value, and keep the result in the attribute reserved amount.
    - Save the PI reservation.
  - In case of quota reserve request, for every qualified reserve enabled PI attribute:

    - Add the subtraction of its current value from the old value to the reserved amount.
    - Save the PI reservation.
- In case of a partial debit request, for every qualified reserve enable PI attribute:
    - Subtract its old value (before adding to it the reserved amount) from its current value and keep the result in the reserved amount of the PI reservation.
    - Update its value to the old one.
- In case of a full debit request, remove the session reservation from the PI reservation.

The method receives an object that implements a method to be activated at the end of the processing (postProcess hook method).

The method receives a qualification cookie, which identifies a qualification result set (a set of items that were qualified in the past on a specific event), and uses this result set instead of re-qualifying the event.

The method also receives a Boolean indication that specifies whether the work should be updated to the data store (not committed). The postProcess hook method is activated prior to the update.

The method returns a status, which can be:

- PROCESS_SUCCESS – The event has been processed successfully.
- FATAL – A severe problem was detected during event processing. The process (job) should be aborted.
- REJECT – The event has been rejected due to business reasons or an extension function failure. The process (job) must roll back the transaction (the transaction context might be inconsistent),and must activate the reject API method to report the event to the Error system.
- RETRY – Event processing might lead to a deadlock or incorrect results; therefore, it should be retried. The transaction must be rolled back, and the event processing should start again.

Syntax

```
static EventProcessStatus process(IDBContext*
i_usageContext, IDBContext* i_customerContext, Entity*&
io_pEvent, int64 i_sessionID,
long i_reservTimeout, RequestType i_requestType,
IPostEventProcess* i_pCallback = NULL,
QualificationCookie*& io_pCookie =
QualificationCookie::GetDefaultInstance(), bool
i_persistWork = true )
```

Parameters

Name	Direction	Type	Description
i_usageDB Context	In	IDBContext&	The database context prepared for the Usage database.
i_customer DBContext	In	IDBContext&	The database context prepared for the Customer database.
io_pEvent	In/out	Entity *	The Event entity pointer.
i_sessionID	In	int64	The associated session ID
i_reservTimeout	In	long	The period (in seconds) after which the session reservation should be released
i_requestType,	In	RequestType	The request type. Possible values: `FULL_RESERVE, QUOTA_RESERVE, FULL_DEBIT, PARTIAL_DEBIT, RELEASE_RESERVATIONS, NON_RESERVE`
i_pCallback	In	IPostEventProcess*	A pointer to an object derived from the IPostEventProcess class, which implements the postProcess hook method. The parameter has a NULL default value, which specifies that the callback does not exist.
io_pCookie	In/Out	QualificationCookie*&	The qualification cookie. By default, the qualification cookie is ignored.
i_persistWork	In	bool	The indication whether to update the Usage database context with the modifications performed on the event and related PIs.
	Return Value	EventProcessStatus	The event processing status: ▪ PROCESS_SUCCESS ▪ FATAL ▪ REJECT ▪ RETRY

releaseReservation

This static method is used to release all the reservations made for a specific session.

The release request processing consists of the following steps:

- For every reserve enabled PI of the subscriber:
    - Check if the PI holds a reservation for the session.
    - If it does, for every quantity attribute, add its reserved amount to its current value.
- Remove the session reservation from the PI reservation.

The method receives the Customer database context and Usage database context to which the database operation is applied, customer ID, subscriber ID, session ID, and session start time.

Syntax

```
static EventProcessStatus releaseReservation(IDBContext*
i_usageContext, IDBContext* i_customerContext, long
i_cycleCode, long i_customerID, long i_subscriberID, long
i_eventTypeID, _int64 i_sessionID, DateType
i_sessionStartTime);
```

Parameters

Name	Direction	Type	Description
i_usageDB Context	In	IDBContext&	The database context prepared for the Usage database.
i_customer DBContext	In	IDBContext&	The database context prepared for the Customer database.
i_cycleCode	In	long	Cycle code
i_customerID	In	long	Customer identifier.
i_subscriberID	In	long	Subscriber identifier.
i_eventTypeID	In	long	Event type ID.
i_sessionID	In	_ int64	The session identification.
i_sessionStartTi me	In	DateType	The start time of the session.
	Return Value	EventProcessSta tus	The event processing status: ▪ PROCESS_SUCCESS ▪ FATAL ▪ REJECT ▪ RETRY

Qualification Cookie

During event processing or Event authorization, the event is qualified for the items that participate in its processing. This qualification phase requires processing time, which can be minimized if qualification results can be cached and used afterward.

Example

In a prepaid process, authorization for the event is needed usually more than once during a session. In such a case, the qualification can be made in the initial Authorize request, and the qualification results can be passed to the process requests so that afterwards the qualification phase can be skipped.

The qualification cookie identifies a qualification result-set. It is returned from each authorization request and can be cached by the client. Authorization and process requests can receive qualification result-sets, use them and save the time that the qualification process takes.

Qualification cookies use memory resources and must be released by clients when not required.

Post-Event Process

The Pricing Engine enables its clients to perform additional updates on an event after it was processed but before it is updated to the data store.

The following sections describe the IPostEventProcess class that was designed for this purpose.

IPostEventProcess

The IPostEventProcess interface class is the base class for all post event process activities. Clients that need post event processing activities must inherit this class and override the postProcess method with the required business logic.

Inherited classes are required to override this virtual method. This method must be implemented with the required logic of post event processing. The Pricing Engine activates the method after the event was processed but before the event is updated. The method is not activated when the event is rejected.

Syntax

```
virtual void postProcess(Entity *io_pEvent);
```

Parameters

Name	Direction	Type	Description
io_pEvent	In/Out	Entity *	The interface entity to the event.

Rejecting an Event

During event processing, the Pricing Engine and pricing item type (PITs) implementation can detect states in which the processing cannot continue – usually as a result of data inconsistencies or violation of the pricing rules (e.g., the subscriber to which an event relates does not exist in the Customer database). These events should be reported (**rejected**) to the error system in order to handle them afterwards by the Rejected Investigation Tool (RIT).

reject

This static method is used to report a rejected event (detected by the Pricing Engine) to the error system. The details of the event and the reject details are stored in the usage database. The caller is responsible for rolling back work done on this database prior to calling the reject method and for committing the work to the database afterwards.

Syntax

```
static bool reject (IDBContext& i_dbContext, Entity
&i_event)
```

Parameters

Name	Direction	Type	Description
i_dbContext	In	IDBContext&	The database context prepared for the usage database.
i_event	In	Entity &	The event to be rejected.
	Return Value	Bool	The rejected event report status. TRUE = Successful report.

Rerating

A rerate process always starts with the Pricing Engine initialization for rerate.

The supported rerate levels are:

- Customer – All the events of the customer and the subscriber beneath it are about to be rerated.
- Subscribers of a same customer – All the events of several subscriber of a same customer are about to be rerated.
- Single subscriber – All the events of a subscriber are about to be rerated.

The rerate processing flow, for every rerate level (customer, subscriber, etc.), comprises the following steps

- The performance indicators of the rerate population (depending on the population granularity) must be initialized to their last state before beginning the cycle.
- The rerated events are processed.
- The performance indicators are finalized in the usage database.

This section describes the methods used for the first and the third steps of the rerate. The second step is performed by the stateful process method.

reInitializeCustomerPIs

This static method is used to set the context on which customer or subscribers of a customer the rerate process is activated. Rerate may be operated at the customer level for one or more subscribers of the same customer. At the end of this method, the performance indicators of this customer\subscribers of the rerate cycle are deleted from the database.

Syntax

```
static PIProcessStatus
PricingEngine::reInitializeCustomerPIs(IDBContext*
i_pUsageDBContext, const UtString& i_customerID, const
STD_NAMESPACE vector<const PIEntity*>& i_customerPIs,
const STD_NAMESPACE vector<UtString>* i_pSubscribers =
NULL)
```

Parameters

Name	Direction	Type	Description
i_pUsageDBContext	In	IDBContext*	The usage database context to be used during the re-initialization step.
i_customerID	In	const UtString&	The ID of the customer on which to activate rerate.
i_customerPIs	In	const STD_ NAMESPACE vector<const PIEntity*>&	A vector containing the cross cycle performance indicator entities of the cycle before the rerate.
i_pSubscribers	In	const STD_ NAMESPACE vector<UtString >*	A vector containing the subscriber IDs, if the rerate is done per subscriber and not at the customer level. Default value is NULL (rerate for customer).
	Return Value	PIProcessStatus	The status returned from performance indicator processing in rerate: ■ PI_PROCESS_ SUCCESS ■ PI_PROCESS_ FATAL ■ PI_PROCESS_ RETRY

finalizeCustomerPIs

This static method is used to end the rerate process at the customer or subscriber population level. It restores cross cycle performance indicators to the database that were deleted in the reInitializeCustomerPIs step and not loaded to the database during the event processing step. This method marks the customer or subscribers for rerate in the next cycle to the rerate cycle, if they have cross cycle PIs for the next cycle that were created prior to rerating.

Syntax

```
static bool
PricingEngine::finalizeCustomerPIs(IDBContext*
i_pUsageDBContext, IDBContext* i_pCustomerDBContext)
```

Parameters

Name	Direction	Type	Description
i_pUsageDBContext	In	IDBContext*	The usage database context to be used during the re-initialization step.
i_pCustomerDBContext	In	IDBContext*	The usage database context to be used during the re-initialization step.
	Return Value	PIProcessStatus	The status returned from performance indicators processing in Rerate: ■ PI_PROCESS_ SUCCESS ■ PI_PROCESS_ FATAL ■ PI_PROCESS_ RETRY

finalizeReratedPIs

After the CrossCyclePIForRerateQueryParameters object is created and subsequently used in a cursor, each performance indicator that is fetched creates a need to mark the subscriber or the customer for re-rating. This interface receives the fetched performance indicator and determines on which level rerating is needed.

Syntax

```
static PIProcessStatus
PricingEngine::finalizeReratedPIs(const Entity&
i_PiEntity, IDBContext* i_pUsageDBContext, IDBContext*
i_pCustomerDBContext)
```

Parameters

Name	Direction	Type	Description
i_PiEntity	In	const Entity&	The PIEntity containing the data according to which the subscriber/customer should be marked for rerate
i_pUsageDBContext	In	IDBContext*	The usage DB context, which is used for re-rate marking.
i_pCustomerDBContext	In	IDBContext*	The customer DB context. It is used in loading the customer data, for marking the level for rerate.
	Return Value	PIProcessStatus	The status returned from performance indicator marking for re-rate. Possibilities: ▪ PI_PROCESS_SUCCESS ▪ PI_PROCESS_FATAL ▪ PI_PROCESS_RETRY

5. QUERYING AND EXTRACTING ENTITIES

The Pricing Engine provides an API to retrieve entities stored in the data stores; for example, events and performance indicators; and to retrieve external records, which have resulted from the mapping of former entities to external records:

- Performance Indicator

  Contains accumulated usage information; stored in the TimesTen database table.

- Rated Event

  Contains event data after all stages of processing (formatting, guiding to customer, guiding to service and rating); stored in the Oracle partitioned table.

- Rejected Event

  Contains data of rejected event together with extended error information and handling statuses; stored in the Oracle partitioned table.

- External Record

  Contains data mapped from a performance indicator or rated event. An external record has no persistency, and it is mapped during the query according to the mapping rules (extract).

The population of the required entities can be filtered according to parameters relevant to each entity type. Refer to corresponding query parameter descriptions in the sections below.

The result of a query is:

- An EntityCursor object, which enables iteration on the queried entities when an external records list of names, was not provided to the query.
- An EntityCursor object, which enables iteration on external records when the external records list of names is provided to the query. In this case, the entities are fetched from their data stores and those for which mapping is defined are mapped. The cursor returns the mapped external records, one by one.

Querying Entities and Extracting External Records

This section describes how the Pricing Engine queries entities and extracts external records through the use of query parameters.

Query

This method is used to query entities or extract external records according to the query parameters provided. This method must be called only after initializing the Pricing Engine.

Currently, the following query parameter classes were designed for querying different type of entities

- PIQueryParameters class

  An instance of this class is used to transfer the parameters for querying performance indicator (PI) entities.
- CrossCyclePIQueryParameters class

  An instance of this class is used to transfer the parameters for querying cross cycle performance indicator (PI) entities.
- EventQueryParameters class

  An instance of this class is used to transfer the parameters for querying event entities.
- RejectEventQueryParameters class

  An instance of this class is used to transfer the parameters for querying rejected event entities.
- OutCollectQueryParameters class

  An instance of this class is used to transfer the parameters for querying out-collect event entities.

Syntax

```
static EntityCursor* query( IDBContext* i_pDBContext,
const IQueryParameters& i_queryParameters,
const STD_NAMESPACE vector<UtString>*
i_pVecExternalEntity = NULL, IDBContext* i_pTTContext =
NULL)
```

Parameters

Name	Direction	Type	Description
i_pDBContext	In	IDBContext *	The database context in which the usage entities are stored (e.g., performance indicators in TimesTen and rated events in Oracle).
i_queryParameters	In	const IQueryParameters&	A reference to the query parameters object.

Name	Direction	Type	Description
i_pVecExternalEntity	In	const vector<UtString> *	The types of the external records to be extracted. By default, the list is NULL, which means that a query is performed.
i_pTTContext	In	IDBContext *	The database context in which the customer data is stored. By default, the database context is NULL. The database context must be provided when an extract is required.
	Return Value	EntityCursor*	Pointer to an entity cursor that enables fetching the result one by one.

note

The Pricing Engine allocates the EntityCursor object internally, and it is the caller's responsibility to free it.

Syntax

```
static EntityCursor* query( IDBContext*
i_pPresentDBContext,
IDBContext* i_pHistoryDBContext,
const IQueryParameters& i_queryParameters,
const STD_NAMESPACE vector<UtString>*
i_pVecExternalEntity = NULL,
IDBContext* i_pCustomerDBContext = NULL)
```

Parameters

Name	Direction	Type	Description
i_pPresentDBContext	In	IDBContext *	The database context in which the usage entities are stored (e.g., performance indicators in TimesTen and rated events in Oracle).
i_pHistoryDBContext	In	IDBContext *	The database context in which the history usage entities are stored (e.g., performance indicators and rated events in Oracle).
i_queryParameters	In	const IQueryParameters&	A reference to the query parameters object.

Name	Direction	Type	Description
i_pVecExternalEntity	In	const vector<UtString> *	The types of the external records to be extracted. By default, the list is NULL, which means that a query is performed.
i_pCustomerDB Context	In	IDBContext *	The database context in which the customer data is stored. By default, the database context is NULL. The database context must be provided when an extract is required.
	Return Value	EntityCursor*	Pointer to an entity cursor that enables fetching the result one by one.

The Pricing Engine allocates the EntityCursor object internally, and it is the caller's responsibility to free it.

EntityCursor

This object is used as an entity cursor for the result of a Pricing Engine query or extract. The following sections describe the methods of the EntityCursor.

open

This method prepares the cursor for fetching data (must be called prior to the other methods).

Syntax

```
bool open()
```

Parameters

Name	Direction	Type	Description
	Return Value	bool	Returns false only if the open request failed.

fetch

Fetch the next qualified entity. The method returns a status, which can be:

- FETCH_SUCCESS – The fetch operation finished successfully. The Entity object was populated with the next record data.
- FATCH_FAILURE – The fetch operation failed.
- FETCH_NO_MORE_ROWS – The end of the cursor.

Syntax

```
FetchStatus fetch(Entity*& o_pEntity)
```

Parameters

Name	Direction	Type	Description
o_pEntity	Out	Entity*	Allocated entity if available or null pointer if end of cursor has been reached.
	Return Value	FetchStatus	The status of the fetch operation: ■ FETCH_SUCCESS ■ FETCH_FAILURE ■ FETCH_NO_MORE_ROWS

note

The Pricing Engine allocates the entity object internally and it is the caller's responsibility to free it.

close

This method closes an opened cursor. The method must be called:

- At the end of the iteration.
- When an error has occurred during the fetch.

Syntax

```
bool close()
```

Parameters

Name	Direction	Type	Description
	Return Value	Bool	Returns FALSE only if the close request failed.

getErrorInfo

When there is an error in the EntityCursor, this interface can get the error information.

In some cases more than one error occurs as a result of a single cursor operation. Therefore this interface should be called until the return code is false. (False means that there are no more errors to fetch.)

The method returns a status, which can be:

- SQL_OK – The operation finished successfully.
- SQL_LOCKED_TIMEOUT – The operation failed because of a timeout.
- SQL_TERMINATED_CONNECTION – The operation failed because of a connection failure (connection has become invalid).
- SQL_EXEC_ERROR – A general error was caused by an operation failure
- SQL_NOT_FOUND – There are no more rows to fetch.

Syntax

```
bool getErrorInfo(UtString& o_errorMsg, SqlReturnStatus&
o_returnStatus)
```

Parameters

Name	Direction	Type	Description
o_errorMsg	Out	UtString&	The message that was received from the underlying database.
o_returnStatus	Out	SqlReturnStatus	The status of the fetch operation: ▪ FETCH_SUCCESS ▪ FETCH_FAILURE ▪ FETCH_NO_MORE_ROWS
	Return Value	Bool	TRUE if error information is returned. FALSE if there are no more error messages to fetch.

QueryParameters Base Class

The QueryParameters class is the base class for all the query parameter classes (e.g., PIQueryParameters, EventQueryParameters, etc.). It is derived from the IQueryParameters interface class. It implements the common functionality required by all the specific query parameters classes.

The following section describes the methods of the QueryParameters class (these methods are available in all inherited classes).

addAdditionalWhereClause

This method is useful when it is required to customize the *Where* clause of the query. The method joins the given SQL expression to the query's *Where* clause with an *and* and sets it as the new query's *Where* clause.

Syntax

```
void addAdditionalWhereClause(const UtString&
i_additionalWhereClause)
```

Parameters

Name	Direction	Type	Description
i_additionalWhereClause	In	const UtString&	The additional *Where* clause

addOrderByElement

This method is useful when it is required to sort the result of the query. To sort the result of the query, the specified column is added to the current columns that are being used. The direction of sorting the values in the new column can also be specified.

Syntax

```
void addOrderByElement(const UtString& i_orderByElment,
bool i_ascending =true)
```

Parameters

Name	Direction	Type	Description
i_orderByElment	In	const UtString&	The column name
i_ascending	In	Bool	The sort order. By default the sort order is ascending

setTableAlias

This method is useful when it is required to use an alias name for a table. An alias can be defined for a native table (i.e., the table on which the query is executed) when transferring its name as the table name or to a new table which is used in a join statement.

Syntax

```
void setTableAlias(const UtString& i_tableName, const
UtString& i_aliasName)
```

Parameters

Name	Direction	Type	Description
i_tableName	In	const UtString&	The table name
i_aliasName	In	const UtString&	The alias name

addAliasToFromClause

This method is useful when it is required to add a table to the from clause of the SQL statement for join purposes. The value of the alias parameter is treated as a table when the alias was not defined prior to the call of this method. The order between the new added table and the current tables in the from clause can be controlled by the alias insertion order parameter.

Syntax

```
void addAliasToFromClause(const UtString& i_aliasName,
AliasInsertionOrder i_aliasInsertionOrder)
```

Parameters

Name	Direction	Type	Description
i_aliasName	In	const UtString&	The alias name
i_aliasInsertionOrder	In	const UtString&	The alias insertion order which can be: ■ FRONT – Add the alias as the first in the aliases list ■ BACK – Add the alias as the last in the aliases list

setHintClause

This method is useful when it is required to optimize Select by inserting a hint clause.

Syntax

```
void setHintClause (const UtString& i_hintClause)
```

Parameters

Name	Direction	Type	Description
i_ hintClause	In	const UtString&	The hint clause

setFetchArraySize

This method is used in Oracle mode only. It sets the size of the fetched record array.

Syntax

```
void setFetchArraySize (long i_fetchArraySize)
```

Parameters

Name	Direction	Type	Description
i_ fetchArraySize	In	long	The size of the fetched record array.

PIQuery Parameters

An object of this class is used in order to transfer query parameters with a request to query performance indicators (PIs). The PI query parameters are divided into two groups:

- Mandatory parameters
  - Cycle code: Query for PIs of a specific cycle code.
  - Cycle month: Query for PIs of a specific cycle month.
  - Cycle year: Query for PIs of a specific cycle year.
- Optional parameters
  - Agreement ID: If specified, query for PIs of a specific agreement.
  - Pricing Item ID: If specified, query for PIs of a specific pricing item.
  - Customer ID: If specified, query for PIs of a specific customer.
  - Offer instance: If specified, query for PIs of a specific offer instance.

It is derived from the base class IQueryParameter interface class.

A developer, performing a PIs query, must:

- Create an object of this type (PIQueryParameters), providing the mandatory parameters for the query.
- Populate the optional parameters as necessary.

- Activate the query method of the Pricing Engine.

The following sections describe the methods of the PIQueryParameters class.

PIQueryParameters

The constructor of this class sets the mandatory query parameters for the performance indicators (PIs) query. The result of this query is reduced to only those PIs belonging to the specified cycle information.

Syntax

```
PIQueryParameters(long i_cycleCode, long i_cycleMonth,
long i_cycleYear)
```

Parameters

Name	Direction	Type	Description
i_cycleCode	In	long	The cycle code
i_cycleMonth	In	long	The cycle month
i_cycleYear	In	long	The cycle year

setAgreementID

This method is useful when querying for PIs of a specific agreement. Using this method is optional and when it is used, it adds an additional constraint to existing methods. Using this method more than once overrides the previous use.

Syntax

```
void setAgreementID (const UtString& i_agreementID)
```

Parameters

Name	Direction	Type	Description
i_agreementID	In	const UtString&	The agreement ID

setPricingItemID

This method is useful when querying for performance indicators (PIs) of a specific pricing item. Using this method is optional and when it is used, it adds an additional constraint to existing methods. Using this method more than once overrides the previous use.

Syntax

```
void setPricingItemID (const UtString& i_pricingItemID)
```

Parameters

Name	Direction	Type	Description
i_pricingItemID	In	const UtString&	The Pricing Item ID

setCustomerID

This method is useful when querying for performance indicators (PIs) of a specific customer. Using this method is optional and when it is used, it adds an additional constraint to existing methods. Using this method more than once overrides the previous use.

Syntax

```
void setCustomerID (const UtString& i_customerID)
```

Parameters

Name	Direction	Type	Description
i_customerID	In	const UtString&	The Customer ID

setOfferInstance

This method is useful when querying for performance indicators (PIs) of a specific offer instance. Using this method is optional and when it is used, it adds an additional constraint to existing methods. Using this method more than once overrides the previous use.

Syntax

```
void setOfferInstance (const UtString&
i_piOfferInstanceNumber)
```

Parameters

Name	Direction	Type	Description
i_piOfferInstanceNumber	In	const UtString&	The Offer instance

setIsCrossCycleFlag

This method is optional. When it is used, it adds an additional Is Cross Cycle constraint to the existing Where clause. Each invocation of this method overrides the previous invocation.

Syntax

```
void setIsCrossCycleFlag (long i_isCrossCycleFlag)
```

Parameters

Name	Direction	Type	Description
i_isCrossCycleFlag	In	long	Is cross cycle flag

CrossCyclePIQueryParameters

An object of this class is used to transfer query parameters with a request to query performance indicators (PIs) that are defined as cross cycle. This class is derived from the PIQueryParameters class.

CrossCyclePIQueryParameters

The constructor of this class sets the mandatory query parameters for the PI query. The result of this query is reduced to only those PIs belonging to the specified cycle information that were defined as cross cycle.

Syntax

```
CrossCyclePIQueryParameters(long i_cycleCode, long
i_cycleMonth, long i_cycleYear)
```

Parameters

Name	Direction	Type	Description
i_cycleCode	In	long	The cycle code
i_cycleMonth	In	long	The cycle month
i_cycleYear	In	long	The cycle year

CrossCyclePIForRerateQueryParameters

It may happen that a cross cycle performance indicator (PI), having been created on the basis of the PI from the previous cycle month, was created based on data (the data of the PI in the previous cycle) that has since been changed. In such a situation, the subscriber/customer of that PI should be marked for re-rate.

This query parameters object creates the proper query for retrieving those PIs. The query fetches PIs that match any of the following cases:

- The cross cycle PI was initialized by a PI that has since been updated.
- The cross cycle PI's initialization was not based on any PI of the previous cycle (because, for example, there was none in the database), but a PI of the previous cycle does exist in the database (having been created after the cross cycle PI was initialized).
- The cross cycle PI's initialization was based a PI that no longer exists (because the PI was removed by a rerate).

This query is executed as the final step of the rerate, to mark subscribers or customers for rerate in the currently-open cycle.

CrossCyclePIForRerateQueryParameters

The constructor of this class sets the mandatory query parameters for the PI query. The result of this query is reduced to only those PIs matching the specified cycle information that were defined as cross cycle.

Syntax

```
CrossCyclePIForRerateQueryParameters (long i_cycleCode,
long i_cycleMonth, long i_cycleYear)
```

Parameters

Name	Direction	Type	Description
i_cycleCode	In	long	The cycle code
i_cycleMonth	In	long	The cycle month of the currently open cycle.
i_cycleYear	In	long	The cycle year of the currently open cycle.

EventQueryParameters

An object of this class is used in order to transfer query parameters with a request to query events (rated events).

The Event query parameters are divided into two groups:

- Mandatory parameters
    - Cycle code: Query for events of a specific cycle code.
    - Cycle month: Query for events of a specific cycle month.
    - Cycle year: Query for events of a specific cycle year.
- Optional Parameters
    - Subscriber ID: If specified, query for events of a specific subscriber.
    - Customer ID: If specified, query for events of a specific customer.
    - Event ID: If specified, query for an event with the specific event ID.
    - Event Type ID: If specified, query for events with the specific event Type ID.
    - Date from: If specified, query for events with event start date greater or equal than the specified date.
    - Date to: If specified, query for events with event start date less than the specified date.
    - Subpartition ID: If specified, query for events with the specified Subpartition ID.

A developer, performing an events query, needs to:

1. Create an object of this type (EventQueryParameters) providing the mandatory parameters for the query.
2. Populate the optional parameters as necessary.
3. Activate the query method of the Pricing Engine.

The following sections describe the methods of the EventQueryParameters class.

EventQueryParameters

The constructor of this class sets the mandatory query parameters for the events query. The result of this query will be reduced to the events belonging to the specified cycle information.

Syntax

```
EventQueryParameters(long i_cycleCode, long i_cycleMonth,
long i_cycleYear)
```

Parameters

Name	Direction	Type	Description
i_cycleCode	In	long	The cycle code
i_cycleMonth	In	long	The cycle month

i_cycleYear	In	long	The cycle year

setSubscriberID

This method is useful when querying for events of a specific subscriber. Using this method is optional and when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setSubscriberID (const UtString& i_subscriberID)
```

Parameters

Name	Direction	Type	Description
i_subscriberID	In	const UtString&	The subscriber ID

setCustomerID

This method is useful when querying for events of a specific customer. Using this method is optional and when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use. When setting the customer ID, the subpartition ID is resolved internally to optimize the query.

Syntax

```
void setCustomerID (const UtString& i_customerID)
```

Parameters

Name	Direction	Type	Description
i_customerID	In	const UtString&	The customer ID

setEventID

This method is useful when querying for an event with a specific ID. Using this method is optional and when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setEventID (const UtString& i_eventID)
```

Parameters

Name	Direction	Type	Description
i_eventID	In	const UtString&	The event ID

setEventTypeID

This method is useful when querying for events of a specific type (derivations of the type are not included). Using this method is optional and when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setEventTypeID (const UtString& i_eventTypeID)
```

Parameters

Name	Direction	Type	Description
i_eventTypeID	In	const UtString&	The event type ID

setDateRange(dateFrom, dateTo)

This method is useful when querying for events occurred at the specified date interval. Using this method is optional and when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setDateRange (const DateType& i_startingDate, const
DateType& i_dateUntil)
```

Parameters

Name	Direction	Type	Description
i_startingDate	In	DateType	The starting date
i_dateUntil	In	DateType	The ending date

setSubPartitionID

This method is useful when querying for events within a specified subpartition. Using this method is optional, and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use. The specified subpartition ID also overrides the subpartition ID calculated by the customer ID if specified.

The subpartition ID was mainly designed to optimize concurrent event queries for the extract processes.

Syntax

```
void setSubPartitionID (const UtString& i_subPartitionID)
```

Parameters

Name	Direction	Type	Description
i_subPartitionID	In	const UtString&	The Subpartition ID

RejectedEventQueryParameters

An object of this class is used in order to transfer query parameters with a request to query rejected events. The RejectedEventQueryParameters class is derived from the base class EventQueryParameters.

The Rejected Event query parameters are divided into two groups:

- Mandatory parameters
  - Cycle code: Query for rejected events of a specific cycle code.
  - Cycle month: Query for rejected events of a specific cycle month.
  - Cycle year: Query for rejected events of a specific cycle year.

- Optional parameters (in addition to the optional parameters of the event query)
    - Error code: If specified, query for rejected events of a specific error code.
    - Error category: If specified, query for rejected events of a specific error category.
    - Processing status: If specified, query for rejected events of a specific processing status.
    - Resolution code: If specified, query for rejected events of a specific resolution code.

The developer, performing a Rejected Events query, needs to:

- Create an object of this type (RejectedEventQueryParameters) providing the mandatory parameters for the query.
- Populate the optional parameters as necessary.
- Activate the query method of the Pricing Engine.

The following sections describe the methods of the RejectedEventQueryParameters.

RejectedEventQueryParameters

The constructor of this class sets values for the mandatory query parameters for the basic Rejected Events query parameters. The result of this query is narrowed to all rejected events belonging to the specified cycle information.

Syntax

```
RejectedEventQueryParameters(long i_cycleCode, long
i_cycleMonth = -1, long i_cycleYear = -1)
```

Parameters

Name	Direction	Type	Description
i_cycleCode	In	Long	The cycle code value
i_cycleMonth	In	long	The cycle month value
i_cycleYear	In	long	The cycle year value

setErrorCode

This method is useful when querying for rejected events of a specific error code. Using this method is optional and when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setErrorCode (const UtString& i_errorCode)
```

Parameters

Name	Direction	Type	Description
i_errorCode	In	const UtString&	The error code

setErrorCategory

This method is useful when querying for rejected events of a specific error category. Using this method is optional and when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setErrorCategory (const UtString& i_errorCategory)
```

Parameters

Name	Direction	Type	Description
i_errorCategory	In	const UtString&	The error category

setProcessingStatus

This method is useful when querying for rejected events of a specific processing status. Using this method is optional and when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setProcessingStatus(const UtString&
i_processingStatus)
```

Parameters

Name	Direction	Type	Description
i_processingStatus	In	const UtString&	The processing status

setResolutionCode

This method is useful in querying for rejected events of a specific resolution code. The method is optional and when it is used, it adds an additional constraint to the existing ones. If used more than once, this method overrides its previous results each time.

Syntax

```
void setResolutionCode(const UtString& i_resolutionCode)
```

Parameters

Name	Direction	Type	Description
i_resolutionCode	In	const UtString&	The resolution code

OutCollectQueryParameters

An object of this class is used in order to transfer query parameters with a request to query outcollect events. The outcollect event query parameter does not receive parameters by itself (but inherits all parameters from its base class). It queries all outcollect events (belonging to cycle 99). The OutcollectQueryParameters class is derived from the base class EventQueryParameters. Users may custom the *where* clause in order to only query the not handled population.

A developer performing an outcollect event query needs to:

- Create an object of this type (OutcollectQueryParameters).
- Populate the optional parameters as necessary.
- Activate the query method of the Pricing Engine.

The following section describes the methods of the OutcollectQueryParameters.

OutcollectQueryParameters

The constructor of this class does not receive parameters.

Syntax

```
OutCollectQueryParameters(const long i_cycleCode =
OUTCOLLECT_CYCLE_CODE)
```

Parameters

Name	Direction	Type	Description
`i_cycleCode`	In	const long	The cycle code value

6. Subscriber Rerate

Querying SubscriberRerate Records

The following section describes the API to the SUBSCRIBER_RERATE table which stores the customers and subscribers of customers needed to be rerated.

subscriberRerateQuery

This method is a Pricing Engine method. It is used to query subscriber rerate records according to the query parameters provided. The SUBSCRIBER_RERATE table contains all the customers or subscribers that are marked for rerating.

The following query parameter class was designed for querying subscriber rerate records:

- SubscriberRerateQueryParameters

  An instance of this class is used to transfer the parameters for querying the SUBSCRIBER_RERATE table.

Syntax

```
ISubscriberRerateCursor*
PricingEngine::subscriberRerateQuery(IDBContext*
i_pUsageDBContext, const SubscriberRerateQueryParameters&
i_queryParameters)
```

Parameters

Name	Direction	Type	Description
i_pUsageDB Context	In	IDBContext *	The Usage database context. The database context in which the subscriber rerate records are stored.
i_query Parameters	In	const Subscriber RerateQuery Parameters&	A reference to the subscriber rerate query parameters object.
	Return Value	ISubscriber RerateCursor *	An interface to a subscriber rerate cursor that enables fetching the result one by one.

note

The Pricing Engine allocates the subscriber rerate cursor internally, and it is the caller's responsibility to free it.

ISubscriberRerateCursor

The interface to the subscriber rerate cursor that represents the result of a subscriber rerate query. The following sections describe the methods of the ISubscriberRerateCursor interface

open

This method prepares the cursor for fetching data (must be called prior to the other methods).

Syntax

```
bool open()
```

Parameters

Name	Direction	Type	Description
	Return Value	bool	Returns FALSE only if the open request failed.

fetch

Fetch the next subscriber rerate cursor.

The method returns a status, which can be:

- FETCH_SUCCESS – The fetch operation finished successfully. The records were populated with the next records data.
- FATCH_FAILURE – The fetch operation failed. The record was left unchanged.
- FETCH_NO_MORE_ROWS – The end of the cursor. The record was left unchanged.

Syntax

```
FetchStatus fetch(SubscriberRerateRecord&
o_subscriberRerateRecord)
```

Parameters

Name	Direction	Type	Description
o_subscriber RerateRecord	Out	Subscriber RerateRecord	The subscriber rerate record to be populated.
	Return Value	FetchStatus	The status of the fetch operation: ▪ FETCH_SUCCESS ▪ FETCH_FAILURE ▪ FETCH_NO_MORE_ ROWS

close

This method closes an opened cursor. The method should be called:

- At the end of the iteration
- When an error occurs during the fetch

Syntax

```
bool close()
```

Parameters

Name	Direction	Type	Description
	Return Value	Bool	Returns FALSE only if the close request failed

SubscriberRerateRecord

This class holds the attributes of SUBSCRIBER_RERATE table and all its accessories.

The following sections describe the methods of the SubscriberRerateRecord.

getCycleCode

This method returns the cycle code.

Syntax

```
long getCycleCode () const
```

Parameters

Name	Direction	Type	Description
	Return Value	long	Returns the cycle code

setCycleCode

This method sets the cycle code to the subscriber rerate record.

Syntax

```
void setCycleCode (long i_cycleCode)
```

Parameters

Name	Direction	Type	Description
i_cycleCode	In	long	The cycle code

getCycleMonth

This method returns the cycle month.

Syntax

```
long getCycleMonth () const
```

Parameters

Name	Direction	Type	Description
	Return Value	long	Returns the cycle month

setCycleMonth

This method sets the cycle month to the subscriber rerate record.

Syntax

```
void setCycleMonth (long i_cycleMonth)
```

Parameters

Name	Direction	Type	Description
i_cycleMonth	In	Long	The cycle month

getCycleYear

This method returns the cycle year.

Syntax

```
long getCycleYear () const
```

Parameters

Name	Direction	Type	Description
	Return Value	long	Returns the cycle year

setCycleYear

This method sets the cycle year to the subscriber rerate record.

Syntax

```
void setCycleYear (long i_cycleYear)
```

Parameters

Name	Direction	Type	Description
i_cycleYear	In	long	The cycle year

getSubscriberID

This method returns the subscriber ID.

Syntax

```
long getSubscriberID () const
```

Parameters

Name	Direction	Type	Description
	Return Value	Long	Returns the subscriber ID

setSubscriberID

This method sets the subscriber ID to the subscriber rerate record.

Syntax

```
void setSubscriberID (long i_subscriberID);
```

Parameters

Name	Direction	Type	Description
i_subscriberID	In	long	The subscriber ID

getCustomerID

This method returns the customer ID.

Syntax

```
long getCustomerID ()
```

Parameters

Name	Direction	Type	Description
	Return Value	Long	Returns the customer ID

getActivityCode

This method returns the activity code.

Syntax

```
const UtString & getActivityCode () const
```

Parameters

Name	Direction	Type	Description
	Return Value	const UtString &	Returns the activity code

setActivityCode

This method sets the activity code to the subscriber rerate record.

Syntax

```
void setActivityCode (const UtString & i_activityCode)
```

Parameters

Name	Direction	Type	Description
i_activityCode	In	const UtString &	The activity code.

getActivitySource

This method returns the activity source.

Syntax

```
const UtString & getActivitySource () const
```

Parameters

Name	Direction	Type	Description
	Return Value	const UtString &	Returns the activity source

setActivitySource

This method sets the activity source to the subscriber rerate record.

Syntax

```
void setActivitySource (const UtString &
i_activitySource)
```

Parameters

Name	Direction	Type	Description
i_activitySource	In	const UtString &	The activity source

getTargetSystem

This method returns the target system.

Syntax

```
const UtString & getTargetSystem () const
```

Parameters

Name	Direction	Type	Description
	Return Value	const UtString &	Returns the target system

setTargetSystem

This method sets the target system to the subscriber rerate record.

Syntax

```
void setTargetSystem (const UtString & i_targetSystem)
```

Parameters

Name	Direction	Type	Description
i_targetSystem	In	const UtString &	The target system

getApplicationID

This method returns the application ID.

Syntax

```
const UtString & getApplicationID () const
```

Parameters

Name	Direction	Type	Description
	Return Value	const UtString &	Returns the application ID

setApplicationID

This method sets the application ID to the subscriber rerate record.

Syntax

```
void setApplicationID (const UtString & i_applicationID);
```

Parameters

Name	Direction	Type	Description
i_applicationID	In	const UtString &	The application ID

getSysCreationDate

This method returns the system creation date.

Syntax

```
const DateType & getSysCreationDate () const
```

Parameters

Name	Direction	Type	Description
	Return Value	const DateType &	Returns the system creation date

setSysCreationDate

This method sets the system creation date to the subscriber rerate record.

Syntax

```
void setSysCreationDate (const DateType &
i_sysCreationDate)
```

Parameters

Name	Direction	Type	Description
i_sysCreation Date	In	const DateType &	The system creation date

getSysUpdateDate

This method returns the system update date.

Syntax

```
const DateType & getSysUpdateDate () const
```

Parameters

Name	Direction	Type	Description
	Return Value	const DateType &	Returns the system update date

setSysUpdateDate

This method sets the system update date to the subscriber rerate record.

Syntax

```
void setSysUpdateDate (const DateType & i_sysUpdateDate)
```

Parameters

Name	Direction	Type	Description
i_sysUpdateDate	In	const DateType &	The system update date

SubscriberRerate Query Parameters

An object of this class is used in order to transfer query parameters with a request to query subscriber rerate records

The subscriber rerate query parameters are divided into two groups:

- Mandatory Parameters
  - Cycle code: Query for subscriber rerate of a specific cycle code.
  - Cycle month: Query for subscriber rerate of a specific cycle month.
  - Cycle year: Query for subscriber rerate of a specific cycle year.
- Optional Parameters
  - Activity code: The activity that occurred that required rerating.
  - Date from: Query for subscriber rerate records, which were updated later than the specified date.

A developer performing a subscriber rerate query must:

- Create an object of this type (SubscriberRerateQueryParameters) providing the mandatory parameters for the query.
- Populate the optional parameters as necessary.
- Activate the subscriberRerateQuery method of the Pricing Engine.

The following sections describe the methods of the SubscriberRerateQueryParameters class.

SubscriberRerateQueryParameters

The constructor of this class sets the mandatory query parameters for the subscriber rerate query. The result of this query is narrowed to the events belonging to the specified cycle information.

Syntax

```
SubscriberRerateQueryParameters(long i_cycleCode, long
i_cycleMonth, long i_cycleYear)
```

Parameters

Name	Direction	Type	Description
i_cycleCode	In	long	The cycle code
i_cycleMonth	In	long	The cycle month
i_cycleYear	In	Long	The cycle year

setActivityCode

This method is useful when querying for subscriber rerate with a specific activity code. Using this method is optional and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setActivityCode(const UtString& i_activityCode);
```

Parameters

Name	Direction	Type	Description
i_activityCode	In	const UtString&	The activity code

setDateFrom

This method is useful when querying for subscriber rerate records that were updated later than the **date from**. Using this method is optional, and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setDateFrom(const DateType& i_dateFrom);
```

Parameters

Name	Direction	Type	Description
i_dateFrom	In	const UtString&	The date from

addAdditionalWhereClause

This method is useful when it is required to customize the **Where** clause of the query. The method joins the given SQL expression to the query's **Where** clause with an **and** and sets it as the new query's **Where** clause.

Syntax

```
void addAdditionalWhereClause(const UtString&
i_additionalWhereClause)
```

Parameters

Name	Direction	Type	Description
i_additionalWhereClause	In	const UtString&	The additional **Where** clause

DeleteSubscriberRerate

This method is a Pricing Engine method. It is used to delete subscriber rerate records from data storage, according to the subscriber rerate query parameters provided.

The following query parameter class was designed for deleting subscriber rerate records.

- SubscriberRerateQueryParameters

  An instance of this class is used to transfer the parameters for deleting subscriber rerate records from the SUBSCRIBER_RERATE table.

The method returns a status, which can be:

- DELETE_SUCCESS – The delete operation finished successfully.
- DELETE_FAILURE – An error occurred during the delete operation.
- NO_DATA_FOUND – The delete operation did not find any record that matches the delete criterion – no records were deleted.

Syntax

```
DeleteStatus
PricingEngine::deleteSubscriberRerate(IDBContext*
i_pUsageDBContext, const SubscriberRerateQueryParameters&
i_queryParameters)
```

Parameters

Name	Direction	Type	Description
i_pUsageDBContext	In	IDBContext *	The usage database. The database context in which the subscriber rerate records are stored.
i_queryParameters	In	const Subscriber RerateQuery Parameters&	A reference to the subscriber rerate query parameters object.

Name	Direction	Type	Description
	Return Value	DeleteStatus	The status of the delete operation: ■ DELETE_SUCCESS ■ DELETE_FAILURE ■ NO_DATA_FOUND

markSubscriberForRerate

This method is a Pricing Engine method. It is used to write subscriber rerate records to the SUBSCRIBER_RERATE table according to the subscriber parameters provided.

The following method was used to check if the subscriber has an agreement level offer:

```
SubOfferLevel::IsCustomerLevel(subscriberID,i_cycleCode,i
_cycleInstance,i_cycleYear);
```

Syntax

```
bool PricingEngine::markSubscriberForRerate(long
i_subscriberID, long i_customerID, long i_cycleCode, long
i_cycleYear, long i_cycleInstance, const UtString&
i_targetSystem, const UtString& i_activityCode, const
UtString& i_activitySource, const UtString&
i_applicationID, IDBContext* i_customerContext,
IDBContext* i_usageContext)
```

Parameters

Name	Direction	Type	Description
i_subscriberID	In	long	The subscriber ID
i_customerID	In	long	Customer ID
i_cycleCode	In	long	Cycle code
i_cycleYear	In	long	Cycle year
i_cycleInstance	In	long	Cycle Instance
i_targetSystem	In	const UtString&	Target system
i_activityCode	In	const UtString&	Activity code
i_activitySource	In	const UtString&	Activity source
i_applicationID	In	const UtString&	Application ID
i_customerContext	In	IDBContext*	The customer database context used for taking customer offers
i_usageContext	In	IDBContext*	The usage database context used for marking the subscriber for Rerate

Name	Direction	Type	Description
	Return Value	bool	True if successfully written to the database, otherwise False.

7. ETRIEVING EXTENDED ENTITY INFORMATION

Extended entity information is information that is stored and retrieved from the data store in addition to the entities data (attributes). The extended information is stored always in columns, as opposed to dynamic data.

The following sections describe the extended information of rated event, rejected event, outcollect event and performance indicator entities.

Control Fields Extended Information

The rated event, rejected event, outcollect event and performance indicator are all extended with control fields.

Control fields include:

- System creation date – The time stamp the record was created in the data store.
- System update date – The time stamp the record was last updated. For new records, the time stamp of the system update date is the system update date.
- Application ID – The ID of the application which last updated the record (e.g., Rater, Rerater).

The Pricing Engine is responsible to set values to the control fields on the records when it inserts or updates the data store.

The ControlFieldsExtendedInfo class provides the methods required to access the values of its fields. A ControlFieldsExtendedInfo object can be retrieved from each of the above entities.

ControlFieldsExtendedInfo

An instance of this class can be constructed according to an already constructed object of the type (C++ copy constructor) so that the values of the control fields are set only by the Pricing Engine to ensure correctness of Pricing Engine APIs that rely on this information.

getSysCreationDate

This method is useful to retrieve the time stamp in which the record represented by the entity was created.

Syntax

```
void getSysCreationDate(UtString& o_sysCreationDate)
```

Parameters

Name	Direction	Type	Description
o_sysCreationDate	Out	UtString	The system creation date

getSysUpdateDate

This method is useful to retrieve the time stamp in which the record represented by the entity was last updated.

Syntax

```
void getSysUpdateDate(UtString& o_sysUpdateDate)
```

Parameters

Name	Direction	Type	Description
o_sysUpdateDate	Out	UtString	The system update date

getApplicationID

This method is useful to retrieve the ID of the application which last updated the record.

Syntax

```
void getApplicationID(UtString& o_applicationID)
```

Parameters

Name	Direction	Type	Description
o_applicationID	Out	UtString	The application ID

EventExtendedInfo

The EventExtendedInfo class represents the extended information of the rated events. It inherits the ControlFieldsExtendedInfo class from where it receives the control fields extended information.

In addition to the control fields extended information, it includes:

- Target ID
- Sub-partition ID

EventExtendedInfo class provides the methods required to access the values of its fields. An EventExtendedInfo object can be retrieved from entities representing rated events. It is impossible to generate an instance of this class.

getSubPartitionID

This method is useful to retrieve the rated event sub-partition ID.

Syntax

```
const UtString& getSubPartitionID() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The sub-partition ID

getDispatchingCaseTargetIdentifier

This method is useful to set the rated Event Target ID. Using this method is optional. When it is used, it adds an additional constraint to the existing ones. Each invocation of this method overrides the previous invocation.

Syntax

```
const UtString& getDispatchingCaseTargetIdentifier ()
const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The Target ID

OutCollectEventExtendedInfo

The OutCollectEventExtendedInfo class represents the extended information of the OutCollect events. It includes the Status parametr, which is to indicate whether the event was delivered.

OutCollectEventExtendedInfo class provides methods required to access the values of its fields. An OutCollectEventExtendedInfo objected can be retrieved from entities representing outcollect events and cannot be generated by a C++ constructor.

getStatus

This method is useful to retrieve the indication whether the outcollect event was delivered.

Syntax

```
char getStatus() const
```

Parameters

None

setStatus

This method is useful to set outcollect events as delivered.

Syntax

```
void setStatus(char i_status) const
```

Parameters

Name	Direction	Type	Description
i_status	In	char	The outcollect event status

RejectedEventExtendedInfo

The RejectedEventExtendedInfo class represents the extended information of the rejected events. It inherits the ControlFieldsExtendedInfo class from where it receives the control fields extended information.

In addition to the control fields extended information, it includes:

- Error date
- Error code
- Error category
- Additional error information
- Processing status
- Processing status date
- Format
- Resolution code
- Resolution date
- Resolution Resource value
- Resource type
- Source ID
- Target ID

RejectedEventExtendedInfo class provides the methods required to access the values of its fields. A RejectedEventExtendedInfo objected can be retrieved from entities representing rejected events.

For the list of error codes, see Appendix A: “Reasons for Rejects.”

RejectedEventExtendedInfo

This method is a copy constructor. It is useful to create an instance of this class initialized with an object derived from ControlFieldsExtendedInfo from which it initializes its control fields.

An object of this class is created when the Pricing Engine’s envelope (host process) wishes to reject an event with its own reasons.

Syntax

```
RejectedEventExtendedInfo(const
ControlFieldsExtendedInfo& i_other)
```

Parameters

Name	Direction	Type	Description
i_other	In	const UtString&	The error date

getErrorDate

- This method is useful to retrieve the date on which the event is rejected. The date is returned in the YYYY-MM-DD 24HH:MI:SS format.

Syntax

```
const UtString& getErrorDate() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The error date

setErrorDate

This method is useful to set the date on which the event is rejected. Using this method is optional and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

The date must have the YYYY-MM-DD 24HH:MI:SS format.

Syntax

```
void setErrorDate(const UtString& i_errorDate)
```

Parameters

Name	Direction	Type	Description
i_errorDate	In	const UtString&	The error date

getErrorCode

This method retrieves the error code representing the reason for a reject.

Various reasons that can cause a reject. The RejectedEventExtendedInfo class provides an error code field that can be tested in order to identify the cause of the reject. The table in Appendix A provides the codes and the reasons that they represent.

note *The error codes are provided by an enumeration called ErrorCode.*

Syntax

```
const UtString& getErrorCode() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The error code, represented as a string

setErrorCode

This method is useful to set the error code representing the reject reason. Using this method is optional and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setErrorCode(const UtString& i_errorCode)
```

Parameters

Name	Direction	Type	Description
i_errorCode	In	const UtString&	The error code

getErrorCategory

This method is useful to retrieve the error category of the error code.

Syntax

```
const UtString& getErrorCategory() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The error category

setErrorCategory

This method is useful to set the error category error code. Using this method is optional and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setErrorCategory(const UtString& i_errorCategory)
```

Parameters

Name	Direction	Type	Description
i_errorCategory	In	const UtString&	The error category

getAddErrorInfo

This method is useful to retrieve the additional error information on the reject reason.

Syntax

```
const UtString& getAddErrorInfo() const
```

Parameters

Name	Direction	Type	Description
	Return value	Const UtString&	The additional error information

setAddErrorInfo

This method is useful to set additional error information on the reject reason. Using this method is optional and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setAddErrorInfo(const UtString& i_addErrorInfo)
```

Parameters

Name	Direction	Type	Description
i_addErrorInfo	In	const UtString&	The additional error information

getProcessingStatus

This method is useful to retrieve the processing status of the rejected event.

Syntax

```
const UtString& getProcessingStatus() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The processing status

setProcessingStatus

This method is useful to set the processing status of the rejected event. Using this method is optional and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

void setProcessingStatus(const UtString& i_processingStatus)

Parameters

Name	Direction	Type	Description
i_processingStatus	In	const UtString&	The processing status

getProcessingStatusDate

This method is useful to retrieve the date on which the processing status was last modified on the rejected event. The date is returned in the YYYY-MM-DD 24HH:MI:SS format.

Syntax

```
const UtString& getProcessingStatusDate() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The processing status date

setProcessingStatusDate

This method is useful to set the date on which the processing status of the rejected event is modified. Using this method is optional and, when used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use. The date must have the YYYY-MM-DD 24HH:MI:SS format.

Syntax

```
void setProcessingStatusDate(const UtString&
i_processingStatusDate)
```

Parameters

Name	Direction	Type	Description
i_processingStatusDate	In	const UtString&	The processing status date

getFormat

This method is useful to retrieve the format of the rejected event.

Syntax

```
const UtString& getFormat() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The rejected event format

setFormat

This method is useful to set the format of the rejected event. Using this method is optional and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setFormat(const UtString& i_format)
```

Parameters

Name	Direction	Type	Description
i_format	In	const UtString&	The rejected event format

getResolutionCode

This method is useful to retrieve the rejected event resolution code.

Syntax

```
const UtString& getResolutionCode() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The rejected event resolution code

setResolutionCode

This method is useful to set the rejected event resolution code. Using this method is optional and, when it is used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setResolutionCode(const UtString& i_resolutionCode)
```

Parameters

Name	Direction	Type	Description
i_resolutionCode	In	const UtString&	The rejected event resolution code

getResolutionDate

This method is useful to retrieve the rejected Event resolution date. The date is returned in the YYYY-MM-DD 24HH:MI:SS format.

Syntax

```
const UtString& getResolutionDate() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The rejected event resolution date

setResolutionDate

This method is useful to set the rejected event resolution date. Using this method is optional and, when used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use. The date must be in the YYYY-MM-DD 24HH:MI:SS format.

Syntax

```
void setResolutionDate(const UtString& i_resolutionDate)
```

Parameters

Name	Direction	Type	Description
i_resolutionDate	In	const UtString&	The rejected event resolution date

getResolutionDesc

This method is useful to retrieve the rejected event resolution description.

Syntax

```
const UtString& getResolutionDesc() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The rejected event resolution description

setResolutionDesc

This method is useful to set the rejected event resolution description. Using this method is optional and, when used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setResolutionDesc (const UtString&
i_resolutionDescription)
```

Parameters

Name	Direction	Type	Description
i_resolution Description	In	const UtString&	The rejected event resolution description

getResourceValue

This method is useful to retrieve the rejected event resource value.

Syntax

```
const UtString& getResourceValue() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The rejected event resource value

setResourceValue

This method is useful to set the rejected event resource value. Using this method is optional, and, when used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setResourceValue(const UtString& i_resourceValue)
```

Parameters

Name	Direction	Type	Description
i_resourceValue	In	const UtString&	The rejected event resource value

getResourceType

This method is useful to retrieve the rejected event resource type.

Syntax

```
const UtString& getResourceType() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The rejected event resource type

setResourceType

This method is useful to set the rejected event resource type. Using this method is optional and, when used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setResourceType(const UtString& i_resourceType)
```

Parameters

Name	Direction	Type	Description
i_resourceType	In	const UtString&	The rejected event resource type

getSourceID

This method is useful to retrieve the rejected event source ID.

Syntax

```
const UtString& getSourceId() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The rejected event source ID

setSourceID

This method is useful to set the rejected event source ID. Using this method is optional and, when used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setSourceId(const UtString& i_sourceID)
```

Parameters

Name	Direction	Type	Description
i_sourceID	In	const UtString&	The rejected event source ID

getTargetID

This method is useful to retrieve the rejected event target ID.

Syntax

```
const UtString& getTargetID() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The rejected event target ID

setTargetID

This method is useful to set the rejected event target ID. Using this method is optional, and, when used, it adds an additional constraint to the existing ones. Using this method more than once overrides the previous use.

Syntax

```
void setTargetID(const UtString& i_targetID)
```

Parameters

Name	Direction	Type	Description
i_targetID	In	const UtString&	The rejected event target ID

8. Entity APIs

From the point of view of a processing system, an entity is a record containing data – for example, an event containing a report of subscriber activity.

Different types of records are represented by different types of entities. For example, different types of events represent different types of subscriber activities.

Each entity has a list of attributes. Each attribute has a name (unique in the scope of the specific entity) and a type definition. For example, an event might have the **start time** attribute defined using the DateTime elementary type and the **charge** attribute defined using the Amount elementary type.

The entity might also have complex attributes. While a simple attribute can hold a single value, a complex attribute is an array of values. For example, the event might have the Duration complex attribute defined using Duration elementary type.

When an entity object is created, its identifier must be specified. The Pricing Engine (PE) has an interface for the creation of entity objects for all supported types — events, customer parameters, and so on – and that interface is the only way to create entity objects. (Entity objects cannot be created by a C++ constructor.) Pricing Engine clients determine the life cycle of the objects, and it is the clients' responsibility to free these objects or to recycle them.

Creation of an entity is a time-consuming operation. The Pricing Engine allows for recycling of entities as a way of reducing the number of creation operations. The Pricing Engine maintains a pool of entities which stores the recycled entities. When an entity is required for the envelope, or internally by the Pricing Engine, first the pool is scanned for the required entity. If, and only if, the required entity does not exist in the pool, a new instance is created.

The entity stores its data in an internal structure, which, should never be accessed directly but only through the API methods of the entity class designed for this purpose.

The following sections describe the entity class methods classified by their specific purpose.

Accessing Entity Attributes

Attributes of an entity can be identified in two ways:

- By their names – as defined in the pricing program. These names are case sensitive (unless the case insensitive mode was specified in Pricing Engine initialization, in which case the names must be provided according to the case insensitive rules).
- By their index – calculated by the Pricing Engine to gain an optimized access.

An attribute may be indicated as **Has no value**, which means that no value is set for the attribute. This indicator should be checked using the entity methods designed for this purpose prior to **Get** operations. Any attempt to retrieve a value of an attribute that has no value will produce an error.

This section describes the entity methods that were designed to extract and modify event data.

set (Name,String,UOM)

This method is used to set new values to attributes of any type. The string-formatted value must be convertible to the attribute type (e.g., the value for a numeric type attribute can include digits, a negative sign and a decimal point). The following describes the format of the string value and the conversion performed by the operation if such is required.

- String type attribute – The string value is set exactly as provided (no conversion).
- Date type attributes – The string value must have the YYYY-MM-DD 24HH:MI:SS format. The year of the date must be greater than 1900.
- Boolean type attributes – The string value must be either TRUE or FALSE.
- Numeric type attributes – The string value must be a valid numeric value and can include a negative sign and decimal point. The decimal point implicitly defines the precision of the value.
    - Precision conversion – The string value is converted to a numeric value with the precision defined for the attribute. Attributes that were not defined with precision are treated as if they were defined with precision zero (no precision).
    - Unit of measure (UOM) conversion – Attributes, which are defined with UOM, assume that the string value is in the defined UOM unless a different UOM is explicitly provided as parameter. When a different UOM is provided, the value is converted from the provided UOM to the attribute's UOM. UOM can be provided only when setting a value to an attribute defined with UOM.

In addition to the above, the method sets the entity change indicator to Changed and the has no value indicator to Has value.

Syntax

```
void set(const UtString& i_attrName, const char
*i_pValue, const UtString* i_pUOM = NULL)
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The name of the attribute to be set.
i_pValue	In	const char *	The value to set the attribute within a string format. The character string must be terminated with NULL.

Name	Direction	Type	Description
i_attrName	In	const UtString&	The name of the attribute to be set.
i_pUOM	In	const UtString *	The UOM of the value. This parameter can be provided only when setting a value to numeric attributes associated with UOMs. By default, the parameter value is NULL, and, in this case, the value is handled as it is in the attribute's UOM.

set (Name,Numeric,UOM)

This method is used to set a new value to attributes of numeric and date types. The following describes the format of the numeric value and the conversion performed by the operation if it is required.

- Date type attributes – The value is a valid date in milliseconds (counting from zero since the year 1900). The time portion can be accurate up to units of second (not milliseconds).
- Numeric type attributes – The value must be a valid numeric value and can include a negative sign. If the attribute is defined with precision, the value has a coded value in the {**v,p**} format, where the **p** least significant digits of the value are the precision digits and the remaining **v** digits are the significant digits.
  - Precision conversion – The value is converted to the precision of the attribute according the {v,p} format. Attributes not defined with precision are treated as if they were defined with precision zero (no precision).
  - Unit of measure (UOM) conversion – Attributes defined with UOM, assume that the value is in the defined UOM unless a different UOM is explicitly provided as a parameter. When a different UOM is provided, the value is converted from the provided UOM to the attribute UOM. UOM can be provided only when setting a value to an attribute defined with UOM.

In addition to the above, the method sets the entity change indicator to Changed and the has no value indicator to Has value.

Syntax

```
void set(const UtString& i_attrName, const _int64
i_value, const UtString* i_pUOM = NULL)
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The name of the attribute to be set.
i_value	In	const _int64	The numeric value to set for the attribute.

Name	Direction	Type	Description
i_pUOM	In	const UtString*	The UOM of the value. This parameter can be provided only when setting a value to numeric attributes associated with UOMs. By default, the parameter value is NULL, and, in this case, the value is handled as it is in the attribute's UOM.

set (Name,Boolean)

This method is used to set new values to Boolean type attributes. It does not perform any data conversion and values are set exactly as given.

In addition to the above, the method sets the entity change indicator to Changed and the has no value indicator to Has Value.

Syntax

```
void set(const UtString& i_attrName, const BooleanType&
i_value)
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The name of the attribute to be set
i_value	In	const BooleanType&	The value to set for the attribute

get (Name,String,UOM)

This method is used to retrieve a string containing the value of attributes of any type. The following describes the format of the return value and the conversion performed by the operation if it is required:

- Date type attributes – The return is a string containing a date in the YYYY-MM-DD 24HH:MI:SS format.
- Boolean type attributes – The return value is a string containing either TRUE or FALSE.
- String type attributes – The return value is a string containing the value of the attributes.
- Numeric type attributes – The return value is a string containing a valid numeric value with a negative sign for negative values. If the attribute is defined with precision, the return value has a decimal point according to this precision.
  - Unit of measure (UOM) conversion – Attributes that are associated with UOM assume that the return value should have the defined UOM unless a different UOM is explicitly provided as parameter. When a different UOM is provided, the return value is converted from the attribute UOM to the provided UOM. A UOM can be

provided only when retrieving the value of an attribute that is defined with a UOM.

The method can be applied only to attributes, which Have value. Any attempt to use this method on an attribute indicated as Has no value produces an error.

Syntax

```
void get(const UtString& i_attrName, UtString& o_value,
const UTString* i_pUOM = NULL) const
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The name of the attribute.
o_value	Out	UtString&	The attribute's value.
i_pUOM	In	const UtString*	The UOM of the return value. This parameter can be provided only when getting the value of numeric attributes associated with UOMs. By default, the parameter value is NULL, and, in this case, the value is handled as it is in the attribute's UOM.

get (Name,Numeric,UOM)

This method is used to retrieve a numeric value of the numeric and date type attributes or derivations of these types. The following describes the format of the return value and the conversion performed by the operation if it is required.

- Date type attributes – The return value is a numeric value that represents a date in milliseconds (counting from zero since the year 1900).
- Numeric type attributes – The value must be a valid numeric value and can include a negative sign. If the attribute is defined with precision, the return value has a coded value in {**v,p**} format, where the **p** least significant digits of the value are the precision digits and the remaining **v** digits are the significant digits.
- Unit of measure (UOM) conversion – Attributes defined with UOM. It is assumed that the return value should have the defined UOM, unless a different UOM is explicitly provided as parameter. When a different UOM is provided, the return value is converted from the attribute UOM to the provided UOM. A UOM can be provided only when retrieving the value of an attribute defined with a UOM.

The method can be applied only to attributes that Have value. Any attempt to use this method on an attribute indicated as Has no value produces an error.

Syntax

```
void get(const UtString& i_attrName, NumericType&
o_value, const UtString* i_pUOM = NULL) const
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The name of the attribute.
o_value	Out	NumericType&	The attribute's value.
i_pUOM	In	const UtString&	The UOM of the return value. This parameter can be provided only when getting the value of numeric attributes associated with UOMs. By default, the parameter value is NULL, and, in this case, the value is handled as it is in the attribute's UOM.

get (Name, Boolean)

This method is used to retrieve a Boolean value of a Boolean type attribute.

The method can be applied only to attributes that Have value. Any attempt to use this method on an attribute indicated as Has no value, produces an error.

Syntax

```
void get(const UtString& i_attrName, BooleanType&
o_value) const
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The name of the attribute
o_value	Out	BooleanType&	The attribute's value

get (Name, Complex)

This method is used to retrieve an interface object for the complex attribute specified. The complex attribute interface provides an API to access (retrieve, set, add, etc.) its data.

Syntax

```
IComplexAttribute get(const UtString& i_attrName) const
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The name of the attribute
	Return Value	IComplexAttribute	The complex attribute

isAttributeNoValue

This method checks whether an attribute has the indication of Has no value, specifying that a value was not set to the attribute.

Syntax

```
bool isAttrNoValue(const UtString& i_attrName) const;
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The attribute name.
	Return Value	Bool	Returns TRUE if the attribute is not set with value.

setAttributeNoValue

This method sets the attribute Has no value indicator; that is, the attribute is not set with a value.

Syntax

```
void setAttrNoValue(const UtString& I_attrName) const;
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The attribute name

Archiving Data

Entity data is stored in an internal structure. This internal structure provides numerous benefits, such as:

- Performance

  Initialization of entity object with data in this structure (dynamic data) is extremely fast.

- Dynamic Storage

  Entities are stored in the database. Dynamic data can be stored in a single database column. Modifying the entity structure does not affect the database table structure.

- Storage Space Economy

  The internal structure is optimized and requires minimal storage space.

The following sections describe the methods that handle archived data.

Init

This method initializes the entity object with data buffer. The data is expected to be in the internal structure of the entity type (the event type which the entity object represents). The data should include all the entity attributes (i.e., static attributes and dynamic attributes). It is recommended that the entity object representing the event be created with the DON'T_INITIALIZE_EVENT initialization mode to eliminate unnecessary memory allocation. The method copies the transferred data as a parameter and does not assume responsibility for the provided data buffer.

To set data buffers that were encoded on a big-endian platform, a reverse action must be specified on the little-endian platform. The reverse performs a conversion from little-endian to big-endian and vice versa (e.g., Windows to UNIX).

Syntax

```
void init (long i_BLOBSize, char *i_pDynamicData, bool
i_reverse)
```

Parameters

Name	Direction	Type	Description
i_BLOBSize	In	long	The size of the data provided. This value must specify exactly the size of the data portion (not the size of the buffer).
i_pDynamicData	In	char *	A pointer to the buffer containing the data in the internal structure.
i_reverse	In	Bool	The reverse indication that specifies whether a conversion from little-endian to big-endian (or vice versa) is required. By default, no conversion is done.

InitReversed

This method initializes the entity object with the data buffer. The data is expected to be in the internal structure of the entity type (the event type which the entity object represents). The data should include all the entity attributes (i.e., static attributes and dynamic attributes). It is recommended that the entity object representing the event be created with the DON’T_INITIALIZE_EVENT initialization mode to eliminate unnecessary memory allocation. The method copies the transferred data as a parameter and does not assume responsibility of the provided data buffer.

The method performs conversion from little-endian to big-endian and vice versa (e.g., from Windows to UNIX) on the given data buffer.

Syntax

```
void initReversed (long i_BLOBSize, char *i_pDynamicData)
```

Parameters

Name	Direction	Type	Description
i_BLOBSize	In	long	The size of the data provided. This value must specify exactly the size of the data portion (not the size of the buffer).
i_pDynamicData	In	char *	A pointer to the buffer containing the data in the internal structure.

InitDynamic

This method initializes the entity object with a data buffer. The data is expected to be in the internal structure of the entity type (the event type represented by the entity object). The data should include the dynamic attributes. The method copies the transferred data as a parameter and does not take charge of the provided data buffer.

To set data buffers that were encoded on a big-endian platform, a reverse action must be specified on the little-endian platform. The reverse performs a conversion from little-endian to big-endian and vice versa (e.g., from Windows to UNIX).

Syntax

```
void initDynamic(long i_BLOBSize, char *i_pDynamicData,
bool i_reverse)
```

Parameters

Name	Direction	Type	Description
i_BLOBSize	In	long	The size of the data provided. This value must specify exactly the size of the portion of data (not the size of the buffer).
i_pDynamicData	In	char *	A pointer to the buffer containing the data in the internal structure.
i_reverse	In	Bool	The reverse indication that specifies whether a conversion from little-endian to big-endian (or vice versa) is required. By default no conversion is made.

InitDynamicReversed

This method initializes the entity object with a data buffer. The data is expected to be in the internal structure of the entity type (which is the event type represented by the entity object). The data should include the dynamic attributes. The method copies the transferred data as a parameter and does not take charge of the provided data buffer.

The method performs conversion from little-endian to big-endian and vice versa (e.g., from Windows to UNIX) on the given data buffer.

Syntax

```
void initDynamicReversed(long i_BLOBSize, char
*i_pDynamicData
```

Parameters

Name	Direction	Type	Description
i_BLOBSize	In	long	The size of the provided data. This value must exactly specify the size of the portion of data (not the size of the buffer).
i_pDynamicData	In	char *	A pointer to the buffer containing the data in the internal structure.

InitHybrid

This method initializes the Entity object with a data buffer. The data is expected to be in the internal structure of the Entity type (the Event type represented by the entity object). The data should include the dynamic attributes that are not marked as fielded in the implementation repository. The method copies the transferred data as a parameter and does not take charge of the provided data buffer.

Syntax

```
void initHybrid(long i_BLOBSize, char *i_pDynamicData)
```

Parameters

Name	Direction	Type	Description
i_BLOBSize	In	long	The size of the data provided. This value must exactly specify the size of the portion of data (not the size of the buffer).
i_pHybridBLOB	In	char *	A pointer to the buffer containing the data in the internal structure.

getBLOB

This method returns the entity's data encoded in the internal structure of the entity type and the size of the data. The data includes all the entity attributes (i.e., static and dynamic attributes).

The method allocates memory internally and assigns it to the pointer provided by the caller (the caller should not allocate memory to store the data). It is the caller's responsibility to free the memory allocated internally in order to store the entity's data (using delete [ ] operator).

To get data from a big-endian platform to a little-endian platform or vice versa (a standard PC, for example, is little-endian), a conversion by reversal must be specified. A reversal indicator specifies whether such a conversion is necessary.

Syntax

```
void getBLOB(char*& o_BLOB,long& o_BLOBSize, bool
i_reverse) const
```

Parameters

Name	Direction	Type	Description
o_BLOB	Out	char*&	A pointer for the dynamic data buffer which is internally allocated.
o_BLOBSize	Out	long&	The size of the dynamic data in the buffer.
i_reverse	In	Bool	The reversal indication, specifying whether to compensate for the difference between big-endian and little-endian by converting data. By default, no conversion is made.

getBLOB (External Memory Allocation)

This method returns the entity's data encoded in the internal structure of the entity type, and the size of the data. The data includes all the entity attributes (both static and dynamic attributes).

The method receives a data buffer, which was allocated by the invoker, and the buffer size. The method sets its data in the provided buffer. If the buffer is too small to contain the encoded data, the method returns a failure status, and the required size is returned to the invoker.

To get data from a big-endian platform to a little-endian platform or vice versa (a standard PC, for example, is little-endian), a conversion by reversal must be specified. A reversal indicator specifies whether such a conversion is necessary.

Syntax

```
bool getBLOB(char* o_BLOB,long i_bufferSize, long&
o_BLOBSize, bool i_reverse) const
```

Parameters

Name	Direction	Type	Description
o_BLOB	Out	char*	A pointer for the data buffer allocated by the invoker.
i_bufferSize	In		The buffer size
o_BLOBSize	Out	long&	The size of the dynamic data in the buffer. When the buffer is too small to contain the encoded data, the required size is set in this parameter.
i_reverse	In	bool	The reversal indication, specifying whether to compensate for the difference between big-endian and little-endian by converting data. By default, no conversion is made.
	Return Value	bool	Buffer adequacy. The method returns FALSE if the data buffer is too small to contain the encoded data, otherwise TRUE.

getBLOBReversed

This method returns the entity’s data encoded in the internal structure of the entity type and the size of the data. The data includes all the entity attributes (i.e., static and dynamic attributes).

The method allocates memory internally and assigns it to the pointer provided by the caller (the caller should not allocate memory to store the data). It is the caller’s responsibility to free the memory allocated internally in order to store the entity’s data (using delete [ ] operator).

The method performs conversion from little-endian to big-endian and vice versa (e.g., from Windows to UNIX) on the given data buffer.

Syntax

```
void getBLOBReversed(char*& o_BLOB,long& o_BLOBSize)
const
```

Parameters

Name	Direction	Type	Description
o_BLOB	Out	char*&	A pointer for the dynamic data buffer, which is internally allocated.
o_BLOBSize	Out	long&	The size of the dynamic data in the buffer.

getDynamic

This method returns the entity's dynamic attributes data encoded in the internal structure of the entity type and the size of the data.

The method allocates memory internally and assigns it to the pointer provided by the caller (the caller should not allocate memory to store the data). It is the caller's responsibility to free the memory allocated internally to store the entity's data (using delete [ ] operator).

To get data from a big-endian platform to a little-endian platform, a reverse action must be specified. The reverse performs a conversion from little-endian to big-endian and vice versa (e.g., from Windows to UNIX).

Syntax

```
void getDynamic(char*& o_BLOB,long& o_BLOBSize, bool
i_reverse=false) const
```

Parameters

Name	Direction	Type	Description
o_BLOB	Out	char*&	A pointer for the dynamic data buffer, which is internally allocated.
o_BLOBSize	Out	long&	The size of the dynamic data in the buffer.
i_reverse	In	bool	The reverse indication that specifies whether a conversion from little-endian to big-endian (or vice versa) is required. By default no conversion is made.

getDynamic (External Memory Allocation)

This method returns the entity's dynamic attributes data encoded in the internal structure of the entity type, and the size of the data.

The method receives a data buffer, which was allocated by the invoker, and the buffer size. The method sets its data in the provided buffer. If the buffer is too small to contain the encoded data, the method returns a failure status, and the required size is returned to the invoker.

To get data from a big-endian platform to a little-endian platform, a reverse action must be specified. The reverse performs a conversion from little-endian to big-endian and vice versa (e.g., from Windows to UNIX).

Syntax

```
bool getDynamic (char* o_pBuffer, long i_bufferSize,
long& o_dynamicDataSize, bool i_reverse=false) const
```

Parameters

Name	Direction	Type	Description
o_pBuffer	Out	char*	A pointer for the data buffer, which contains the entity's dynamic data.
i_bufferSize	In	long	The buffer size.
o_dynamicDataSize	Out	long&	The size of the dynamic data in the buffer.
i_reverse	In	bool	The reverse indication that specifies whether a conversion from little-endian to big-endian (or vice versa) is required. By default, no conversion is made.
	Return Value	bool	The method returns FALSE when the data buffer is too small to contain the encoded data, otherwise TRUE.

getDynamicReversed

This method returns the entity's dynamic attributes data encoded in the internal structure of the entity type and the size of the data.

The method allocates memory internally and assigns it to the pointer provided by the caller (the caller should not allocate memory to store the data). It is the caller's responsibility to free the memory allocated internally to store the entity's data (delete []).

The method performs conversion from little-endian to big-endian and vice versa (e.g., from Windows to UNIX) on the given data buffer.

Syntax

```
void getDynamicRerversed(char*& o_BLOB,long& o_BLOBSize)
const
```

Parameters

Name	Direction	Type	Description
o_BLOB	Out	char*&	A pointer for the dynamic data buffer, which is internally allocated.
o_BLOBSize	Out	long&	The size of the dynamic data in the buffer.

getDynamicReversed(External Memory Allocation)

This method returns the entity's dynamic attributes data encoded in the internal structure of the entity type and the size of the data.

The method receives a data buffer which was allocated by the invoker and the buffer size. The method sets its data on the provided buffer. An error is produced when the given buffer is too small to contain the entity's dynamic data.

The method performs conversion from little-endian to big-endian and vice versa (e.g., from Windows to UNIX) on the given data buffer.

Syntax

```
void getDynamicReversed(char* o_pBuffer, long
i_bufferSize, long& o_dynamicDataSize) const
```

Parameters

Name	Direction	Type	Description
o_pBuffer	Out	char*	A pointer for the data buffer, which contains the entity's dynamic data.
i_bufferSize	In	long	The buffer size.
o_dynamicDataSize	Out	long&	The size of the dynamic data in the buffer.

getHybrid

This method returns the entity's dynamic attributes that are not defined as Fielded, encoded in the internal structure of the Entity type, and the size of the data.

The method allocates memory internally, and assigns it to the pointer provided by the caller (the caller should not allocate memory to store the data). It is the caller's responsibility to free the memory allocated internally to store the entity's data (delete []).

Syntax

```
void getHybrid(char*& o_BLOB,long& o_BLOBSize) const
```

Parameters

Name	Direction	Type	Description
o_BLOB	Out	char*&	A pointer to the dynamic data buffer, which is allocated internally.
o_BLOBSize	Out	long&	The size of the dynamic data in the buffer.

getHybrid

This method returns the entity's dynamic attributes that are not defined as Fielded, encoded in the internal structure of the Entity type, and the size of the data.

The method receives an already allocated buffer, and assigns the encoded BLOB to it.

Syntax

```
void getHybrid(char* io_BLOB, long i_allocatedSize, long&
o_BLOBSize) const
```

Parameters

Name	Direction	Type	Description
io_BLOB	In/Out	char*	A pointer to the data buffer, which contains the entity's dynamic data.
i_allocatedSize	In	long	The buffer size.

Name	Direction	Type	Description
o_BLOBSize	Out	long&	The size of the dynamic data in the buffer.
	Return Value	bool	FALSE when the data buffer is too small to contain the encoded data; otherwise, TRUE.

Entity Life Cycle

The entity life cycle is controlled by the envelope (the host process that activated the Pricing Engine and that controls it). When the envelope has finished using an entity, the envelope has two options:

- Release the entity – Give back, to the system, the resources that were acquired by the entity.
- Recycle the entity – Give the entity back to the Pricing Engine so the entity can be used again later.

The Pricing Engine (PE) maintains an internal pool of entities. When an entity is required by the envelope or by the PE itself, the PE checks whether the pool contains an instance of the required entity. If and only if a required instance of an entity does not exist in the pool, a new instance is created.

It is advisable to recycle entities – adding them to the pool – because the creation of an entity is time-consuming and recycling can dramatically reduce the number of entity creation operations.

recycle

This static method recycled an entity. The specified entity is returned to the Pricing Engine’s internal pool of entities.

After this operation it is **illegal** to access the entity, and as a safety measure, the entity pointer is set to Null.

Syntax

```
static void Entity::recycle(Entity &*i_pEntity)
```

Parameters

Name	Direction	Type	Description
	In/Out	Entity &*	The entity to recycle. The passed pointer is set to Null by the operation

Accessing the Entity Metadata

Following are the methods provided by the entity class for access to the metadata of an entity instance.

getVersionEffectiveDate

This method returns the effective date of the version that the entity is based on.

Syntax

```
DateType getVersionEffectiveDate() const
```

Parameters

Name	Direction	Type	Description
	Return value	DateType	The effective date of the version that the entity is based on.

getEntityType

This method returns the entity type of the entity.

Syntax

```
EntityType getEntityType() const
```

Parameters

Name	Direction	Type	Description
	Return value	EntityType	The entity type of the entity, which can be: ■ Customer (CustomerParameters) ■ CustomerOffers ■ Event ■ PerformanceIndicator ■ ExternalRecord

getType

This method returns the type of the entity as defined in the PC. (For example, the type of a **GSM Voice** event is **GSM Voice**.)

Syntax

```
UtString getEntityType() const
```

Parameters

Name	Direction	Type	Description
	Return value	UtString	The entity type.

getEntityDescriptor

This method is useful for exploring an entity’s metadata. It returns an object that contains the metadata of the entity (For more information see Chapter 10, “Entity Metadata”).

Syntax

```
const EntityDescriptor* getEntityDescriptor() const
```

Parameters

Name	Direction	Type	Description
	Return value	const EntityDescriptor*	The entity metadata

Entity Extended Information

The entity provides methods to store and access (called, accessors) a single object derived from the EntityExtendedInfo base class.

An entity is in charge of the extended information object it stores, and, during entity destruction, the stored extended information is freed.

The following section describes these accessors.

getExtendedInfo

This method is used to retrieve the extended info object, which the entity stores. An object of type NoExtendedInfo is returned when the entity does not store extended information.

Syntax

```
EntityExtendedInfo& getExtendedInfo() const
```

Parameters

Name	Direction	Type	Description
	Return value	EntityExtendedInfo&	A reference to the Extended Info object the entity stores.

setExtendedInfo

This method is used to store the extended information object in the entity. To override an Extended Info object stored by an entity, the **force set** parameter must be set.

The entity takes charge of the given extended information object; the extended information object is freed when:

- It is overridden by a new extended information object.
- The entity is freed.

Syntax

```
void setExtendedInfo(EntityExtendedInfo *i_pExtendedInfo,
bool i_bForceSet = false)
```

Parameters

Name	Direction	Type	Description
i_extendedInfo	In	EntityExtendedInfo*	A pointer to the extended information object.
i_bForceSet	In	Bool	True value specifies to override current ended information object with new one, releasing the old object. By default (False value), does not override an existing ended information object.

9. DATA TYPES

The Pricing Engine core provides a class for each basic type it supports (i.e., string, numeric, date and Boolean types). These classes are designed to provide additional functionality required for each basic type (e.g., string manipulation for string types) and serve as parameter types of the event API. The elementary types are translated to these basic types.

The following sections describe the basic data classes, their methods and their functionality.

Numeric Type

The NumericType class represents any fixed numeric data type (i.e., natural, integer and decimal with fixed point numbers). A NumericType object can be created according to any unit of measurement (UOM) defined in the pricing program. It converts values from one unit to another unit of the same UOM type. It supports the traditional arithmetical operations.

The heart of the numeric type is a simple fraction. However, when retrieving a value from the numeric type, the value is represented as an integer value of a certain precision (that is, the simple fraction 3/2 if requested with precision of 1 will be 15, which is 1.5). If the requested precision is not enough to represent the simple fraction accurately, the Pricing Engine performs mathematical rounding automatically.

Objects of this type are used to retrieve and set values to the entities' numeric type attributes.

The following sections describe the methods of the NumericType class.

constructor (_int64, NumericTypeInfo)

The method creates an instance of the class according to an integer value, and its numeric type information. The value is expected to be encoded as a pair of value and precision (i.e., a value of 32445 can be equivalent to any number of the following:

- 0.32445
- 3.2445
- 32.445
- 324.45
- 3244.5
- 32445

The former is encoded as [0,5] while the later is encoded as [5,0]). The precision on the numeric type information determines the actual number. If the precision is 0, then the number is 32445; if the precision is 3, then the number is 32.445.

When numeric type information is not specified for the value, the value is treated as if it has zero precision.

Syntax

```
NumericType(_int64 i_value, const NumericTypeInfo&
i_typeInfo = numericTypeInfo())
```

Parameters

Name	Direction	Type	Description
i_value	In	_int64	The value as integer in the precision and UOM as specified by the numeric type information (i.e., if the value is 567 and the precision is 2, then the number is equivalent to 5.67).
i_typeInfo	In	const NumericTypeInfo&	The numeric type information associated with the value.

copy constructor

Creates a NumericType object with value identical to the value of the given NumericType object (the numeric type information is copied to from the given NumericType object).

Syntax

```
NumericType(const NumericType& i_numeric)
```

Parameters

Name	Direction	Type	Description
i_numeric	In	const NumericType&	The value to copy from

constructor (NumericTypeBase, NumericTypeInfo)

Creates an object of this class with the value of the given by NumericTypeBase object. The type, UOM, and precision of the value are specified by the NumericTypeInfo parameter.

When numeric type information is not specified for the value, the value is set without UOM and with zero precision.

Syntax

```
NumericType(const NumericTypeBase& i_numeric,

const NumericTypeInfo& i_typeInfo = numericTypeInfo())
```

Parameters

Name	Direction	Type	Description
i_numeric	In	const NumericTypeBase&	The value to copy from
i_typeInfo	In	const NumericTypeInfo&	The numeric type information associated with the value

constructor (UtString)

Creates an object of this class with the value of the given string. If the value has a decimal point, the created object will be set to the precision of the number of the digits after the decimal point. The string may have negative sign and in that case the value of the object will be negative.

Syntax

```
NumericType(const UtString& i_value)
```

Parameters

Name	Direction	Type	Description
i_value	In	const UtString&	The value

operator = (NumericType)

Assigns a value to the object (the numeric type information is assigned together with the value).

Syntax

```
NumericType& operator=
(const NumericType& i_value)
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to copy from.
	Return value	NumericType&	The object containing the value it was set with.

operator == (NumericType)

Tests whether the object on the left side of the operator is equal to the object on the right side of the operator.

Object can be checked for equality only if they are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric).

When the UOM of the left operand and the right operand are not the same, one of the operand values will be converted to meet the other UOM.

Syntax

```
bool operator== (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to compare to.
	Return value	bool	▪ TRUE if equal ▪ FALSE if not equal

operator != (NumericType)

Tests whether the object on the left side of the operator is NOT equal to the object on the right side of the operator.

Object can be checked for equality only if they are of the same elementary type or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric).

When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
bool operator!= (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to compare to
	Return value	bool	▪ TRUE if NOT equal ▪ FALSE if equal

operator <(NumericType)

Tests whether the on the left side of the operator is smaller than the object on the right side of the operator.

Object can be compared only if they are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric).

When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
bool operator< (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to compare to.
	Return value	bool	▪ TRUE if the left operand's value is less than the right operands. ▪ FALSE if the left operand's value is greater or equal to the right operand's value.

operator <= (NumericType)

Tests whether the object on the left side of the operator is smaller than or equal to the object on the right side of the operator.

Objects can be compared only if they are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric).

When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
bool operator<= (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to compare to.
	Return value	bool	▪ TRUE if the left operand's value is less than or equal to the right operand's value. ▪ FALSE if the left operand's value is greater than the right operand's value.

operator > (NumericType)

Tests whether the object on the left side of the operator is greater than the object on the right side of the operator.

Object can be compared only if they are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric).

When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
bool operator> (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to compare to.
	Return value	bool	▪ TRUE if the left operand's value is greater than the right operands. ▪ FALSE if the left operand's value is less than or equal to the right operand's value.

operator >= (NumericType)

Tests whether the object on the left side of the operator is greater than or equal to the object on the right side of the operator.

Object can be compared only if they are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric).

When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
bool operator>= (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to compare to.
	Return value	bool	▪ TRUE if the left operand's value is greater than or equal to the right operand's value. ▪ FALSE if the left operand's value is less than the right operand's value.

operator += (NumericType)

Adds the value to this numeric object. The operation changes the left operand.

Addition operation is legal only if the operands are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric).

When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
NumericType& operator+= (const NumericType& i_value)
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to add to the numeric object
	Return value	NumericType&	The result of the addition

operator – = (NumericType)

Subtracts the value from the numeric object. The operation changes the left operand.

Subtraction operation is legal only if the operands are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric). When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
NumericType& operator-= (const NumericType& i_value)
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to subtract from the numeric object
	Return value	NumericType&	The result of the subtraction

operator *= (NumericType)

Multiplies the value of the numeric object by another numeric object. The operation changes the left operand.

Multiplication operation is legal only under the following conditions:

- The operands are not based on any elementary type (the basic elementary type for numeric values – Numeric). In this case, the result is without any elementary type.
- One of the operands is of a certain elementary type and the other is not. In this case, the result is in that elementary type.

Syntax

```
NumericType& operator*= (const NumericType& i_value)
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to multiply the numeric object by
	Return value	NumericType&	The result of the multiplication

operator /= (NumericType)

Divides the value of the numeric object by another numeric object. The operation changes the left operand.

Division operation is legal only under the following conditions:

- The operands are not based on any elementary type (the basic elementary type for numeric values – Numeric). In this case, the result is without any elementary type.
- One of the operands is of a certain elementary type, and the other is not. In this case, the result is in that elementary type and UOM.
- Both operands are of the same elementary type, and have UOMs. In this case, the result is without UOMs.

Any attempt to divide a numeric object by zero will produce an error.

Syntax

```
NumericType& operator/= (const NumericType& i_value)
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to divide the numeric object by
	Return value	NumericType&	The result of the division

operator + (NumericType)

Adds a numeric value to another numeric value and returns the result.

Addition operation is legal only if the operands are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric). When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
NumericType operator+ (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to add to the numeric object
	Return value	NumericType	The result of the addition

operator – (NumericType)

Subtracts the value from the numeric object and returns the result.

Subtraction operation is legal only if the operands are of the same elementary type, or if they are not based on any elementary type (the basic elementary type for numeric values – Numeric).

When the UOM of the left operand and the right operand are not the same, one of the operand values is converted to meet the other UOM.

Syntax

```
NumericType operator- (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to subtract from the numeric object
	Return value	NumericType	The result of the subtraction

operator * (NumericType)

Multiplies the value of a numeric object by another numeric object.

Multiplication operation is legal only under the following conditions:

- The operands are not based on any elementary type (the basic elementary type for numeric values – Numeric). In this case, the result is without any elementary type.
- One of the operands is of a certain elementary type, and the other is not. In this case, the result is in that elementary type.

Syntax

```
NumericType operator* (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to multiply the numeric object by.
	Return value	NumericType	The result of the multiplication.

operator / (NumericType)

Divides the value of a numeric object by another numeric object.

Division operation is legal only under the following conditions:

- The operands are not based on any elementary type (the basic elementary type for numeric values – Numeric). In this case, the result is without any elementary type.
- One of the operands is of a certain elementary type, and the other is not. In this case, the result is in that elementary type and UOMs.
- Both operands are of the same elementary type, and have UOMs. In this case, the result is without UOMs.

note Any attempt to divide a numeric object by zero will produce an error.

Syntax

```
NumericType operator/ (const NumericType& i_value) const
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to divide the numeric object by.
	Return value	NumericType	The result of the division.

setValue (NumericType)

Assigns a value to the object (the numeric type information is assigned together with the value).

Syntax

```
void setValue (const NumericType& i_value)
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericType&	The value to copy from

setValue (NumericTypeBase, NumericTypeInfo)

Assigns a value to the object. The value is given by a NumericTypeBase object. The precision and the UOM of the value are given by the NumericTypeInfo object.

When numeric type information is not specified for the value, the value is treated as it has zero precision.

Syntax

```
void setValue (const NumericTypeBase& i_numeric,
const NumericTypeInfo& i_typeInfo = NumericTypeInfo())
```

Parameters

Name	Direction	Type	Description
i_numeric	In	const NumericTypeBase&	The value to copy from
I_typeInfo	In	const NumericTypeInfo&	The numeric type information associated with the value

setValue (_int64, NumericTypeInfo)

The method sets value to the object according to an integer value, and its numeric type information. The value is expected to be encoded as a pair of [value,precision] (i.e., a value of 32445 can be any number of the following 0.32445, 3.2445, 32.445, 324.45, 3244.5 or 32445. The former is encoded as [0,5] while the later is encoded as [5,0]). The precision on the numeric type info determines the actual number. If the precision is 0 then the number is 32445, if the precision is 3 then the number is 32.445.

When numeric type information is not specified for the value, the value is treated as it has zero precision.

Syntax

```
void setValue(_int64 i_value,
const NumericTypeInfo& i_typeInfo = numericTypeInfo())
```

Parameters

Name	Direction	Type	Description
i_value	In	_int64	The value as integer in the type, UOM and precision given by the numeric type information (i.e., if the value is 567 and the precision is 2 then the number is 5.67).
i_typeInfo	In	Const NumericTypeInfo&	Holds the Type, UOM information and the precision of the value.

setValue (UtString, NumericTypeInfo)

Creates an object with the value of the UtString input Parameter. The precision of the number is set according to the numeric type info.

Syntax

```
void setValue(const UtString& i_value,
const NumericTypeInfo& I_typeInfo = numericTypeInfo())
```

Parameters

Name	Direction	Type	Description
i_value	In	const UtString&	The value to create the object with
i_typeInfo	In	const NumericTypeInfo&	The numeric type information

getTypeInfo

Returns the NumericTypeInfo object of the numeric type object.

Syntax

```
const NumericTypeInfo& getTypeInfo() const
```

Parameters

Name	Direction	Type	Description
	Return value	const NumericTypeInfo&	The numeric type information of the numeric object

Date Type

The DateType class represents a date-time data type. It is inherited from the NumericType. Currently, its methods that receive a date in string format support only a single mask, which is “YYYY-MM-DD 24HH:MI:SS” (when setting a DateType object by a string, its contents must be formatted according to a specified mask. When receiving a string containing the date from a DateType object, it is formatted according to this mask).

Objects of this type are used to retrieve and set values to an entity date type attributes.

Date type object can store dates since the year 1900 up to 2049 (i.e., ‘1900-01-01 00:00:00’ to ‘2049-31-12 23:59:59’).

The date-time stored in a DateType object is in milliseconds but its precision is of seconds. Therefore, all methods that retrieve or receive parameters in milliseconds should be provided with values in milliseconds in precision of seconds.

The following sections describe the DateType class methods.

constructor(Default)

This method creates an empty DateType object. The emptiness of a DateType object can be tested by the isNull() method. Any attempt to perform operations on an empty DateType Object will produce an error.

Syntax

```
DateType ();
```

Parameters

None

setValue(String)

This method sets a new value to a DateType object. The parameter containing the new value is expected to be in the ‘YYYY-MM-DD 24HH:MI:SS’ format and in the supported date range. Any attempt to set an invalid date-time will produce an error.

Syntax

```
void setValue(const UtString& i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	const UtString&	The date-time value.

setValue(Numeric)

This method sets a value to a DateType object. The parameter containing the new value is expected to be in milliseconds starting from 1900-01-01 00:00:00 and in the supported date range. Any attempt to set an invalid date-time will produce an error.

Syntax

```
void setValue(_int64 i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	_int64	The date-time value

setStartDayTime

This method sets the time portion of the date-time of the DateType object to ’00:00:00.’ Any attempt to set an invalid date-time will produce an error.

Syntax

void setEndDayTime();

Parameters

None

setEndDayTime

This method sets the time portion of the date-time of the DateType object to '23:59:59.' Any attempt to set an invalid date-time will produce an error.

Syntax

void setEndDayTime();

Parameters

None

setDate

This method sets a new date value to a DateType object (the time portion is initialized to '00:00:00'). The parameter containing the new value is expected to be in the 'YYYY-MM-DD' format and in the supported date range. Any attempt to set an invalid date-time will produce an error.

Syntax

```
void setDate(const UtString& i_date);
```

Parameters

Name	Direction	Type	Description
i_date	In	const UtString&	The date value

getValue(String)

This method retrieves the value of the DateType object in the 'YYYY-MM-DD 24HH:MI:SS' format. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
void getValue(UtString& i_value) const;
```

Parameters

Name	Direction	Type	Description
i_value	Out	UtString&	The string to be populated with the date-time value

getValue(Numeric)

This method retrieves the value of the DateType object in milliseconds since the year 1900. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getValue() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The date-time value in milliseconds

getSeconds

This method retrieves the seconds in the time portion of the date-time stored in the DateType object. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getSeconds() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The seconds in the time portion of the date-time

getMinutes

This method retrieves the minutes in the time portion of the date-time stored in the DateType object. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getMinutes() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The minutes in the time portion of the date-time

getHour

This method retrieves the hour in the time portion of the date-time stored in the DateType object. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getHour() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The hour in the time portion of the date-time

getTime

This method retrieves the time of the date-time stored in the DateType object in milliseconds. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getTime() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The time portion of the date-time

getMonth

This method retrieves the month in the date of the date-time stored in the DateType object. For example, January is represented by value of 1. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getMonth() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The month in the date in the date-time

getYear

This method retrieves the year in the date of the date-time stored in the DateType object. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getYear() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The year in the date in the date-time

getDayOfMonth

This method retrieves the day in the date of the date-time stored in the DateType object (value's range 1-31). Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getDayOfMonth() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The day in the date in the date-time

dayOfWeek

This method retrieves the day of week in the date of the date-time stored in the DateType object (value's range 0-6. 0 – Sunday … 6 – Saturday). Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
int dayOfWeek() const;
```

Parameters

Name	Direction	Type	Description
	Return value	int	The day of week in the date in the date-time (0 – Sunday … 6 – Saturday)

getDayOfYear

This method retrieves the day in the year of the date-time stored in the DateType object (value's range 1-365). Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
int getDayOfYear() const;
```

Parameters

Name	Direction	Type	Description
	Return value	Int	The day in the year of the date-time (1-365)

getDayInYear

This method retrieves the number of days in the year of the date-time stored in the DateType object. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
int getDayInYear() const;
```

Parameters

Name	Direction	Type	Description
	Return value	int	The number of days in the year

getMillisecondsToEndOfDay

This method is useful to calculate the time left until the end of the day (23:59:59) in milliseconds. Any attempt to retrieve a value of an empty object will produce an error.

Syntax

```
_int64 getMillisecondsToEndOfDay() const
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The time left till the end of the day

getCurrentDate

This static method retrieves the system's current date.

Syntax

```
Static DateType getCurrentDate();
```

Parameters

Name	Direction	Type	Description
	Return value	DateType	The system’s current date

getIntervalsByDay

This method is useful to retrieve the number of days between the date-time stored by the DateTime object and the parameter (the number of times the midnight is passed). When the object’s date-time is prior to the parameter’s date-time, the result is negative. The operation considers the time portion of the date-time value. Any attempt to apply this method on an empty DateType object or on an empty date-time parameter will produce an error.

Syntax

```
_int64 getIntervalsByDay() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The number of days between the two dates

getIntervalsByWeek

This method is useful to retrieve the number of weeks elapsed between the date-time stored by the DateTime object and the parameter (the number of times a week begins ‘Sunday 00:00:00’). When the object’s date-time is prior to the parameter’s date-time, the result is negative. The operation considers the time portion of the date-time value. Any attempt to apply this method on an empty DateType object or on an empty date-time parameter will produce an error.

Syntax

```
_int64 getIntervalsByWeek() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The number of weeks elapsed between the two dates

getIntervalsByMonth

This method is useful to retrieve the number of month elapsed between the date-time stored by the DateTime object and the parameter (the number of times a month begins ‘1/Month 00:00:00’). When the object’s date-time is prior to the parameter’s date-time, the result is negative. The operation considers the time portion of the date-time value. Any attempt to apply this method on an empty DateType object or on an empty date-time parameter will produce an error.

Syntax

```
_int64 getIntervalsByMonth() const;
```

Parameters

Name	Direction	Type	Description
	Return value	_int64	The number of month between the two dates

addMilliseconds

This method adds a number of milliseconds to the date-time of the DateType object. A negative value will reduce the number of milliseconds from the date-time of the DateType object. Any attempt to apply this method on an empty DateType object will produce an error.

Syntax

```
void addMilliseconds (_int64 i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	_int64	The number of milliseconds to add

addSeconds

This method adds a number of seconds to the date-time of the DateType object. A negative value will reduce the number of seconds from the date-time of the DateType object. Any attempt to apply this method on an empty DateType object will produce an error.

Syntax

```
void addSeconds (_int64 i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	_int64	The number of seconds to add

addMinutes

This method adds a number of minutes to the date-time of the DateType object. A negative value will reduce the number of minutes from the date-time of the DateType object. Any attempt to apply this method on an empty DateType object will produce an error.

Syntax

void addMinutes (int i_value);

Parameters

Name	Direction	Type	Description
i_value	In	int	The number of minutes to add

addHours

This method adds a number of hours to the date-time of the DateType object. A negative value will reduce the number of hours from the date-time of the DateType object. Any attempt to apply this method on an empty DateType object will produce an error.

Syntax

```
void addHours (_int64 i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	_int64	The number of hours to add

addDays

This method adds a number of days to the date-time of the DateType object. A negative value will reduce the number of days from the date-time of the DateType object. Any attempt to apply this method on an empty DateType object will produce an error.

Syntax

```
void addDays (int i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	int	The number of days to add

addMonths

This method adds a number of months to the date-time of the DateType object. A negative value will reduce the number of months from the date-time of the DateType object. Any attempt to apply this method on an empty DateType object will produce an error.

Syntax

```
void addMonths (int i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	int	The number of months to add

isBetweenDates

The method tests whether the date-time of the DateTime object is between the dates interval specified by the parameters. The parameters and the object on which the method is activated must not be empty, otherwise an error is produced. The method returns true if the following formula is satisfied: start date <= this date <= end date.

Syntax

```
bool isBetweenDates(const DateType& i_startDate, const
DateType& i_endDate) const;
```

Parameters

Name	Direction	Type	Description
i_startDate	In	const DateType &	The start date
i_endDate	In	const DateType &	The end date
	Return value	bool	True if the object date is in the dates interval

getDaysBetweenDates

The static method returns the number of days between the two specified dates. The return value is negative when the start date is prior to the end date. Any attempt to pass to this method empty date type objects will produce an error.

Syntax

```
_int64 getDaysBetweenDates(const DateType& i_startDate,
const DateType& i_endDate);
```

Parameters

Name	Direction	Type	Description
i_startDate	In	const DateType &	The start date
i_endDate	In	const DateType &	The end date
	Return value	bool	The days difference

Boolean Type

The Boolean type data class represents a Boolean data type. Objects of this type are used to retrieve and set values to the entity’s Boolean type attributes.

The following sections describe the Boolean type class methods.

Constructor()

This method is the default constructor of the Boolean type class, it creates a Boolean type object and initializes it to false.

Syntax

```
BooleanType ();
```

Parameters

None

Constructor(Boolean)

This method creates a Boolean type object and initializes it according the Boolean value it receives.

Syntax

```
BooleanType (bool i_value);
```

Parameters

Name	Direction	Type	Description
i_value	in	bool	The Boolean value

setValue(BooleanType)

This method sets the Boolean type object with the value of the given parameter.

Syntax

```
void setValue (const BooleanType& i_value);
```

Parameters

Name	Direction	Type	Description
i_value	in	const BooleanType&	The Boolean value

setValue(bool)

This method sets the Boolean type object to the given parameter.

Syntax

```
void setValue (bool i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	bool	The Boolean value

setValue(String)

This method sets the Boolean type object with the value of the given parameter. The operator is case insensitive with respect to the parameters value. The following values represent a true value:

- ‘true’
- ‘yes’
- ‘1’
- ‘y’

The following values represent a false value

- ‘false’
- ‘no’
- ‘0’
- ‘n’

Any attempt to set other values will produce an error.

Syntax

```
void setValue (const UtString& i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	const UtString&	The Boolean value

getValue(boolean)

This method returns the value of the Boolean type object as a bool type value.

Syntax

```
bool getValue () const
```

Parameters

Name	Direction	Type	Description
	Return Value	bool	The value of the object

getValue(UtString)

This method returns the value of the Boolean type object as a string value. The value returns Yes for a true value and No for false value.

Syntax

```
void getValue (UtString& o_value) const
```

Parameters

Name	Direction	Type	Description
o_value	Out	UtString	The value of the object

operator bool()

This is a bool casting operator. It returns the objects value.

Syntax

```
bool operator! () const;
```

Parameters

Name	Direction	Type	Description
	Return value	bool	The value of the object

operator !

The method returns the negation of the object’s value. It returns true when the object’s value is false, otherwise false.

Syntax

```
bool operator! () const;
```

Parameters

Name	Direction	Type	Description
	Return value	bool	The negation of the object’s value

operator ==

The method compares two Boolean type objects. It returns 1 when the two objects are equal, otherwise 0.

Syntax

```
bool operator== (const BooleanType& i_value) const;
```

Parameters

Name	Direction	Type	Description
i_value	In	const BooleanType&	The object to compare with
	Return value	bool	The comparison result

operator !=

The method checks whether two Boolean type objects have different values. It returns True when the two objects have different values, otherwise, False.

Syntax

```
bool operator!= (const BooleanType& i_value) const;
```

Parameters

Name	Direction	Type	Description
i_value	In	const BooleanType&	The object to compare with
	Return value	bool	The comparison result

operator = (BooleanType)

This operator sets the Boolean type object with the value of the given parameter.

Syntax

```
BooleanType& operator= (const BooleanType& i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	const BooleanType&	The value to be assigned
	Return value	BooleanType&	The assigned object

operator = (String)

This operator sets the Boolean type object with the value of the given parameter. The operator is case insensitive with respect to the parameters value. The following values represent a true value:

- ‘true’
- ‘yes’
- ‘1’
- ‘y’

The following values represent a false value

- 'false'
- 'no'
- '0'
- 'n'

Any attempt to set other values will produce an error.

Syntax

```
BooleanType& operator= (const BooleanType& i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	const UtString&	The value to be assigned
	Return value	BooleanType&	The assigned object

operator = (bool)

This operator sets the Boolean type object with the value of the given parameter.

Syntax

```
BooleanType& operator= (bool i_value);
```

Parameters

Name	Direction	Type	Description
i_value	In	bool	The value to be assigned
	Return value	BooleanType&	The assigned object

operator &&

This operator performs a logical 'AND' between the two Boolean type objects.

Syntax

```
bool operator&& (const BooleanType& i_value) const;
```

Parameters

Name	Direction	Type	Description
i_value	In	bool	The right operand for the logical 'AND'
	Return value	bool	The result of the logical 'AND'

operator ||

This operator performs a logical OR between the two Boolean type objects.

Syntax

```
bool operator|| (const BooleanType& i_value) const;
```

Parameters

Name	Direction	Type	Description
i_value	In	bool	The right operand for the logical OR
	Return value	bool	The result of the logical OR

operator ^

This operator performs a logical XOR between the two Boolean type objects.

Syntax

```
bool operator^ (const BooleanType& i_value) const;
```

Parameters

Name	Direction	Type	Description
i_value	In	bool	The right operand for the logical XOR
	Return value	bool	The result of the logical XOR

UtString

The UtString type represents a string of characters. Objects of this type are used to retrieve and set values to the event's string type attributes.

The UtString uses an expansion algorithm to reduce resizing during string manipulations. It expands by the minimum between the required size and the actual size multiplied by 2.

The string length is stored internally to reduce unnecessary calculations to get the string length.

The following sections describe the methods of this class.

UtString(const char*,StringUsage)

This constructor creates UtString object from the character string parameter. The character string has to be a null terminated string.

The constructor also receives a directive that controls the way the character string is treated so the creation of UtString can be more efficient. The directive is of type StringUsage and can have the following values:

- COPY – This is the default value. This is also the most expensive way to create UtString. The UtString allocates a buffer to hold the character string (in the string's length plus a place for the null terminator character). It then copies the character string parameter into this buffer.
- ADOPT – Specifies that the new instance is taking full ownership of the incoming character string parameter. The new instance does not perform memory allocation and copy. It treats the buffer it received as its own, and also has the responsibility of deleting it. Therefore, the character string parameter must be dynamically allocated. Creation of UtString with

this directive is much faster then creation with the COPY directive. The user must remember that after the completion of the constructor, the given character string parameter is considered as an internal member of the UtString and should not be deleted.

- LEASE – Similar to the ADOPT directive, but unlike it, it does not treat the incoming buffer as its own. This means that the UtString uses the buffer but does not own it. It will not delete it or change it. A copy of its contents is made on first attempt to change it (e.g., operator+=). This directive is useful when the user needs to pass a string wrapped as UtString, without the overhead of dynamic memory allocation and copy operation.

Syntax

```
UtString(const char * i_pStr = 0, StringUsage i_usage =
COPY);
```

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	Character string to initialize the UtString
i_usage	In	StringUsage	Determines how character string parameter is treated: ▪ COPY– Allocates a buffer and copies the character string parameter into it. ▪ ADOPT – points to the incoming character string parameter and treats is as its own. The instance is responsible for this buffer, including deleting it. ▪ LEASE – Points to the incoming character string parameter but does not take any responsibility for it.

UtString(size_t, const char*,StringUsage)

This constructor creates a UtString object from the character string parameter. The character string has to be null-terminated.

The constructor also receives a directive that controls the way the character string is treated so that the creation of UtString can be more efficient. The directive is of type StringUsage. It can have the following values:

- COPY – This is the default value. This is also the most expensive way to create UtString. UtString allocates a buffer to hold the character string (the string’s length plus a place for the null terminator character). It then copies the character string parameter into this buffer.

- ADOPT – Specifies that the new instance is taking full ownership of the incoming character string parameter. The new instance does not perform memory allocation and copy. It treats the buffer it received as its own, and also has the responsibility for deleting it. Therefore, the character string parameter must be dynamically allocated. Creation of UtString with this directive is much faster than the creation with the COPY directive. The user must remember that after the completion of the constructor, the given character string parameter is considered as an internal member of UtString, and should not be deleted.
- LEASE – Similar to the ADOPT directive. The difference is that it does not treat the incoming buffer as its own. This means that UtString uses the buffer but does not own it. It will not delete it or change it. A copy of its contents is made on first attempt to change it (e.g., operator+=). This directive is useful when the user needs to pass a string wrapped as UtString without the overhead of dynamic memory allocation and copy operation.

Syntax

```
UtString(size_t i_len, const char * i_pStr = 0,
StringUsage i_usage = COPY);
```

Parameters

Name	Direction	Type	Description
i_len	in	Size_t	The string length without the Null terminator.
i_pStr	In	const char *	Character string to initialize UtString.
i_usage	In	StringUsage	Determines how the character string parameter is treated: ▪ COPY– Allocates a buffer and copies the character string parameter into it. ▪ ADOPT – Points to the incoming character string parameter and treats is as its own. The instance is responsible for this buffer, including its deletion. ▪ LEASE – Points to the incoming character string parameter but takes no responsibility for it.

UtString(const char*,bool)

This constructor creates UtString object from the given character string parameter. The character string parameter has to be a null terminated string.

This constructor is for comparability with old code. Developers should use the former constructor UtString(const char*,StringUsage).

The constructor also receives a bool indication that controls the way the character string parameter is treated so the creation of UtString can be more efficient. This bool indication specifies:

- True – This is the default value. Specifies to the UtString to allocate a buffer to hold the character string (in the string's length plus a place for the null terminator character) and then to copy the character string parameter into this buffer.
- False – Specifies to the UtString to use the string parameter without taking an ownership on it. It will not delete it or change it. A copy of its contents is made on first attempt to change it (e.g., operator+=). This option is useful when the user needs to pass a string wrapped as UtString, without the overhead of dynamic memory allocation and copy operation.

Syntax

UtString(const char * i_pStr, bool i_copyString = true);

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	Character string to initialize the UtString
i_copyString	In	bool	If to copy or not the first argument

UtString (UtString)

The copy constructor creates UtString object based on other UtString object. The new UtString object is constructed with ownership on its contents (i.e., it allocates memory and copies the character string of the other object into it).

Syntax

```
UtString(const UtString& i_other);
```

Parameters

Name	Direction	Type	Description
i_other	In	const UtString&	The other UtString object

UtString(char)

This constructor converts the given character into a character string (e.g., 'a' to "a," etc.) and then initializes the new created object with it.

Syntax

```
UtString(char i_char);
```

Parameters

Name	Direction	Type	Description
i_char	In	char	The character to construct with

UtString(long)

This constructor converts the given long into a character string (e.g., 12345678 to “12345678”) and then initializes the new created object with it.

Syntax

```
UtString(long i_long);
```

Parameters

Name	Direction	Type	Description
i_long	In	long	The long to be constructed with

UtString(short)

This constructor converts the given short into a character string (e.g., 1234 to “1234,” etc.) and then initializes the new created object with it.

Syntax

```
UtString(short i_short);
```

Parameters

Name	Direction	Type	Description
i_short	In	short	The short to be converted

UtString(double)

This constructor converts the given double into a character string (e.g., 2.0 to “2.000000e+000,” etc.) and then initializes the new created object with it.

Syntax

```
UtString(double i_double);
```

Parameters

Name	Direction	Type	Description
i_double	In	double	The double to be constructed with

UtString(float)

This constructor converts the given float into a character string (e.g., 2.0 to “2.000000,” etc.) and then initializes the new created object with it.

Syntax

```
UtString(float i_float);
```

Parameters

Name	Direction	Type	Description
i_float	In	float	The float to be constructed with

UtString(_int64)

This constructor converts the given double into a character string (e.g., 123456789012345 to “123456789012345,” etc.) and then initializes the new created object with it.

Syntax

```
UtString(_int64 i_int64);
```

Parameters

Name	Direction	Type	Description
i_int64	In	_int64	The int64 to be constructed with

UtString(bool)

This constructor converts the given bool into a character string and then initializes the new created object with it. True value is converted into "true" and false value is converted into "false."

Syntax

```
UtString(bool i_bool);
```

Parameters

Name	Direction	Type	Description
i_bool	In	bool	The bool to be constructed with

operator = (UtString)

This operator assigns the data of the other UtString object into it, thus making both of the strings identical in their contents.

Syntax

```
UtString& operator= (const UtString& i_other);
```

Parameters

Name	Direction	Type	Description
i_other	in	const UtString&	The UtString to assign
	Return value	UtString&	The assigned object.

operator = (char *)

This operator assigns the data of the character string into it. The character string parameter must be null terminated.

Syntax

```
UtString& operator= (const char * i_pStr);
```

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	The character string to assign
	Return value	UtString&	The assigned object

operator = (char)

This operator converts the given character into a character string (e.g., 'a' to "a," etc.) and then assigns the converted character string into it.

Syntax

```
UtString& operator= (char i_char);
```

Parameters

Name	Direction	Type	Description
i_char	In	char	The char to assign
	Return value	UtString&	The assigned object

operator = (long)

This operator converts the given long into a character string (e.g., 12345678 to “12345678,” etc.) and then assigns the converted character string into it.

Syntax

```
UtString& operator= (long i_long);
```

Parameters

Name	Direction	Type	Description
i_long	In	long	The long to assign
	Return value	UtString&	The assigned object

operator = (short)

This operator converts the given short into a character string (e.g., 1234 to “1234,” etc.) and then assigns the converted character string into it.

Syntax

```
UtString& operator= (short i_short);
```

Parameters

Name	Direction	Type	Description
i_short	In	short	The short to assign
	Return value	UtString&	The assigned object

operator = (double)

This operator converts the given double into a character string (e.g., 2.0 to “2.000000e+000,” etc.) and then assigns the converted character string into it.

Syntax

```
UtString& operator= (double i_double);
```

Parameters

Name	Direction	Type	Description
i_double	In	double	The double to assign
	Return value	UtString&	The assigned object

operator = (float)

This operator converts the given float into a character string (e.g., 2.0 to “2.000000,” etc.) and then assigns the converted character string into it.

Syntax

```
UtString& operator= (float i_float);
```

Parameters

Name	Direction	Type	Description
i_float	In	float	The float to assign
	Return value	UtString&	The assigned object

operator = (_int64)

This operator converts the given int64 into a character string (e.g., 123456789012345 to “123456789012345,” etc.) and then assigns the converted character string into it.

Syntax

```
UtString& operator= (_int64 i_number);
```

Parameters

Name	Direction	Type	Description
i_number	In	_int64	The _int64 to assign
	Return value	UtString&	The assigned object

size

This method returns the string’s length. The length is measured by the number of characters in the string (e.g., the size of “abc” is 3, the string null terminator is ignored). The size() method and the length() method returns the same result.

Syntax

```
long size() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	long	The string size

length

This method returns the string’s length. The length is measured by the number of characters in the string (e.g., the size of “abc” is 3, the string null terminator is ignored). The size() method and the length() method returns the same result.

Syntax

```
long length() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	long	The string size

capacity

This method returns the size of the internal buffer (number of characters) allocated by the UtString to store its contents. Note that the capacity and length are not always the same.

Syntax

```
long capacity() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	long	The internal buffer size

UtChar operator []

This method returns a UtChar that holds the data of the ith element in the string. The given index should be between 0 to length of the string (not included). Otherwise, an error is produced. Note that this operator cannot be applied on const UtString object.

Syntax

```
UtChar operator[](int i_index);
```

Parameters

Name	Direction	Type	Description
I_index	In	Int	The index of the requested char
	Return value	UtChar	Contains information about the char

const UtChar operator []

This method returns a UtChar that holds the data of the ith element in the string. The given index should be between 0 to length of the string (not included); otherwise an error is produced. Note that this operator can be applied on const UtString object.

Syntax

```
const UtChar operator[] (int i_index) const;
```

Parameters

Name	Direction	Type	Description
I_index	In	int	The index of the requested char
	Return value	const UtChar	Contains information about the char

empty

This method tests the emptiness of a UtString object. An empty UtString is a string containing “” (i.e., its first character is back slash zero) or a string with capacity equal to zero.

Syntax

bool empty() const;

Parameters

Name	Direction	Type	Description
	Return Value	bool	True if string is empty; otherwise false.

isEqual

This method tests whether two UtString objects have the same contents.

Syntax

```
bool isEqual(const UtString * i_pOther) const;
```

Parameters

Name	Direction	Type	Description
i_pOther	In	const UtString *	The string to be compared with
	Return value	bool	True if both strings have the same contents; otherwise false

operator ==(UtString)

This operator tests whether two UtString objects have the same contents.

Syntax

```
bool operator==(const UtString& i_str) const;
```

Parameters

Name	Direction	Type	Description
i_str	In	const UtString&	The string to be compared with
	Return value	bool	True if both strings has the same contents; otherwise false

operator ==(char *)

This operator tests whether the contents of the UtString is identical to the contents of the character string parameter. Note that the character string has to be null terminated.

Syntax

```
bool operator ==(const char * i_pStr) const;
```

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	The character string to be compared with
	Return value	Bool	True if both strings have the same contents; otherwise false

operator != (char *)

This operator tests whether the contents of the UtString are different from the contents of the character string parameter. Note that the character string has to be null terminated.

Syntax

```
bool operator !=(const char * i_pStr) const;
```

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	The character string to be compared with
	Return value	bool	True if both strings have different contents; otherwise false

operator > (char *)

The operator tests whether the contents of the UtString object is greater (string comparison) than the contents of the character string parameter. Note that the character string parameter has to be null terminated.

Syntax

```
bool operator >(const char * i_pStr) const;
```

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	The character string to be compared with
	Return value	bool	True if the contents of the UtString object are greater than the contents of the character string parameter; otherwise false

operator < (char *)

This operator tests whether the contents of the UtString object is less (string comparison) than the contents of the character string parameter. Note that the character string parameter has to be null terminated.

Syntax

```
bool operator <(const char * i_pStr) const;
```

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	The string to be compared with
	Return value	bool	True if the contents of the UtString object are less than the contents of the character string parameter; otherwise false

operator >= (char *)

This operator tests whether the contents of the UtString object is greater or equal (string comparison) to the contents of the character string parameter. Note that the character string parameter has to be null terminated.

Syntax

```
bool operator >=(const char * i_pStr) const;
```

Parameters

Name	Direction	Type	Description
i_pStr	in	const char *	The string to be compared with
	Return value	bool	True if the contents of the UtString object are greater or equal to the contents of the character string parameter; otherwise false

operator <= (char *)

This operator tests whether the contents of the UtString object is less or equal (string comparison) to the contents of the character string parameter. Note that the character string parameter has to be null terminated.

Syntax

bool operator <=(const char * i_pStr) const;

Parameters

Name	Direction	Type	Description
	Return Value	bool	True if the contents of the UtString object are less or equal to the contents of the character string parameter; otherwise false.

compareCStr

This method compares the contents of UtString objects to the contents of a character string parameter. The method returns an integer representing the relations among the strings. Note that the character string parameter has to be null terminated.

Syntax

```
int compareCStr(const char* i_pStr) const;
```

Parameters

Name	Direction	Type	Description
i_pother	In	const char*	The string to compare to

Name	Direction	Type	Description
	Return Value	int	The comparison result: ■ 0 – the strings are equal ■ 1 – the UtString contents are greater then the contents of the character string parameters ■ -1 – the UtString contents are greater then the contents of the character string parameters

operator += (UtString)

This method appends one UtString to another.

Syntax

```
UtString& operator+=(const UtString& i_other);
```

Parameters

Name	Direction	Type	Description
i_other	In	const UtString&	The string to be appended
	Return Value	UtString&	The appended object

operator += (char *)

This method appends the contents of a character string to UtString object. Note that the character string parameter has to be null terminated.

Syntax

UtString& operator +=(const char * i_pStr);

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	The character string to be appended
	Return Value	UtString&	The appended object

operator += (char)

This method appends one char to UtString object.

Syntax

UtString& operator +=(char i_char);

Parameters

Name	Direction	Type	Description
i_char	In	char	The char to be appended
	Return Value	UtString&	The appended object

operator + (UtString)

This method appends two UtString objects. The result is placed in a new UtString object. The two UtStrings that combined to create the result are not changed.

Syntax

```
const UtString operator +(const UtString& i_str) const;
```

Parameters

Name	Direction	Type	Description
i_str	In	const UtString&	The UtString
	Return Value	const UtString	A new UtString object containing the result

operator +(char *)

This method appends a character string to UtString object. The result is placed in a new UtString object. The UtString object and the character string that combined to create the result are not changed. Note that the character string parameter has to be null terminated.

Syntax

```
const UtString operator +(char * i_pStr) const;
```

Parameters

Name	Direction	Type	Description
i_pStr	In	char *	The string that is appended
	Return Value	const UtString&	A new UtString object containing the result

operator +(char)

This method appends a character to UtString object. The result is placed in a new UtString. The UtString and the character that are combined to create the result are not changed.

Syntax

const UtString operator+(char i_char) const;

Parameters

Name	Direction	Type	Description
i_char	In	char	The char to be appended
	Return Value	const UtString&	A new UtString object containing the result

c_str

This method returns a const pointer to the UtString internal buffer. User of this method should remember that this pointer may change when operations require memory reallocation to be made on the UtString object.

Syntax

```
const char* c_str() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	const char *	Pointer to the UtString's internal buffer

abandon

This method tells the UtString to return a pointer to its internal buffer and than to resets itself to an empty string (""). By doing so, the UtString gives up any responsibility for this buffer, and the responsibility becomes the caller's, including the responsibility for deleting it.

If the UtString does not have ownership on the data (i.e., the UtString was created using the LEASE directive – see constructor) it returns NULL.

Syntax

```
char * abandon();
```

Parameters

Name	Direction	Type	Description
	Return Value	char *	Pointer to the internal buffer

operator const char *

This operator returns a const pointer to the ut string internal buffer. User of this method should remember that this pointer may change when operations requires memory reallocation are made on the ut string object. The functionality of this operator is the same as the c_str() method.

Syntax

```
operator const char*() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	const char *	Pointer to the internal buffer

operator short

This operator converts the contents of the ut string object into a short.

When the conversion fails – the string cannot be converted into a short – an error is produced.

Syntax

```
operator short() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	short	The string value converted to short

operator int

This operator converts the contents of the UtString object into an integer.

When the conversion fails – the string cannot be converted into an integer – an error is produced.

Syntax

```
operator int() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	int	The string value converted to int

operator long

This operator converts the contents of the UtString object into a long.

When the conversion fails – the string cannot be converted into a long – an error is produced.

Syntax

```
operator long() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	long	The string value converted to long

operator float

This operator converts the contents of the UtString object into a float.

When the conversion fails – the string cannot be converted into a float – an error is produced.

Syntax

```
operator float() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	float	The string value converted to float

operator double

This operator converts the contents of the UtString object into a double.

When the conversion fails – the string cannot be converted into a double – an error is produced.

Syntax

```
operator double() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	double	The string value converted to double

operator int64

This operator converts the contents of the UtString object into an _int64.

When the conversion fails – the string cannot be converted into a n_int64 – an error is produced.

Syntax

```
operator int64() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	_int64	The string value converted to _int64

extractFirst

Extracts the part of this UtString appearing before the first appearance of the delimiter. The delimiter parameter can be one or more characters that form a null terminated string. An empty UtString object is returned when the delimiter is not found.

Syntax

```
UtString extractFirst(const char* i_pDelimiter);
```

Parameters

Name	Direction	Type	Description
i_pDelimiter	In	const char *	The delimiter
	Return Value	UtString	The result

extractFirst

Extracts the part of this UtString appearing before the first appearance of the delimiter. The delimiter is a single char. An empty UtString object is returned when the delimiter is not found.

Syntax

```
UtString extractFirst(char i_delimiter);
```

Parameters

Name	Direction	Type	Description
i_delimiter	In	char	The delimiter
	Return Value	UtString	The result

extractLast

Extracts the part of this UtString appearing after last appearance of delimiter. The delimiter parameter is one or more characters that form a null terminated string. An empty UtString object is returned when the delimiter is not found.

Syntax

```
UtString extractLast(const char* i_pDelimiter);
```

Parameters

Name	Direction	Type	Description
i_pDelimiter	In	const char *	The delimiter
	Return Value	UtString	The result

concat

This method appends the contents of a character string to UtString object. Note that the character string parameter has to be null terminated.

Syntax

```
long concat(const char* i_pStr);
```

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	The string to append
	Return Value	long	The number of chars that were appended

concat

This method appends a specific amount of characters from a character string to UtString object. Note that the character string parameter has to be null terminated. Any attempt to append an amount of characters greater the length of the character string will produce an error.

Syntax

```
long concat(const char * i_pStr, size_t i_len);
```

Parameters

Name	Direction	Type	Description
i_pStr	In	Const char *	The string to be appended
i_len	In	size_t	The number of characters to append
	Return Value	Long	The number of chars that were appended

concat

This method appends a specific amount of characters starting from a specified position from a character string to UtString object. Any attempt to append an amount of characters greater than the length of the character string or starting from a position out of range will produce an error. Note that the character string parameter has to be null terminated.

Syntax

```
long concat(const char * i_pStr, size_t i_from, size_t
i_len);
```

Parameters

Name	Direction	Type	Description
i_pStr	In	const char *	The string to be appended
i_from	In	size_t	The position to start the appending
i_len	In	size_t	The number of characters to append
	Return Value	long	The number of characters that were appended

insert

This method inserts UtString object in front of another.

Syntax

```
UtString& insert(const UtString& i_pre);
```

Parameters

Name	Direction	Type	Description
i_pre	In	const UtString&	The string to insert
	Return Value	UtString&	The result

insert

This method inserts character string in front of UtString object. Note that the character string has to be null terminated.

Syntax

```
UtString& insert(const char * i_pPreStr);
```

Parameters

Name	Direction	Type	Description
i_pPreStr	In	Const char *	The character string to insert
	Return Value	UtString&	The result

insert

This method inserts an amount of characters from a character string in front of UtString object. Any attempt to insert an amount of characters greater than the length of the character string will produce an error. Note that the character string has to be null terminated.

Syntax

```
UtString& insert(const char * i_preStr, int i_len);
```

Parameters

Name	Direction	Type	Description
i_preStr	In	const char *	The string to be appended
i_len	In	size_t	The number of characters to insert
	Return Value	UtString&	The result

toUpper

This method transforms all lower case characters in the UtString object into upper case characters.

Syntax

```
void toUpper();
```

Parameters

None

toLower

This method transforms all upper case characters in the UtString object into lower case characters.

Syntax

```
void toLower();
```

Parameters

None

find

This method returns the position of the first occurrence of a character in the UtString object (starting from zero).

Syntax

```
long find(char i_char) const;
```

Parameters

Name	Direction	Type	Description
i_char	In	char	The requested char.
	Return value	long	The index of the requested char. If the character is not found –1 is returned.

find_last_of

This method returns the position of the last occurrence of a character in the UtString object.

Syntax

```
long find_last_of(char i_char) const;
```

Parameters

Name	Direction	Type	Description
i_char	In	char	The requested char.
	Return value	long	The POSITION of the requested character. If the char is not found –1 is returned.

substr

This method returns UtString object that holds a sub-string from the current UtString. The sub-string is cut from the UtString starting from a specified position in the specified length. Note that the start position has to be between 0 to the length of the UtString object and the length has to be less or equal to the string length minus the start position.

Syntax

```
UtString substr(int i_nStart, int i_nLen) const;
```

Parameters

Name	Direction	Type	Description
I_nStart	in	int	The position to start in
I_nLen	in	int	The length of the string
	Return Value	UtString	The subset of the original string

reset

The method set the UtString object to be empty.

Syntax

```
void reset();
```

Parameters

Name	Direction	Type	Description
I_flag	in	enum	Deletes the internal buffer or not

reserve

This method resizes the internal buffer of the UtString object at least to the requested size. This method is used to reduce memory allocations performed by the UtString object to store it contents during string manipulations (e.g., the storage required by a UtString object to store the result of an append operation is larger the contents it stored before the operation so its buffer should be reallocated to a larger size).

Syntax

```
long reserve(long i_size);
```

Parameters

Name	Direction	Type	Description
i_size	In	long	Requested size of internal buffer
	Return Value	long	The size that was really allocated

Complex Data Type

The complex data type is a multi-dimensional data type. It can also be described as a multi-dimensional vector. It can have as many dimensions as required, but it must have at least one. The complex data type is homogeneous (i.e., objects of this type always store elements from the same type).

An element in a complex data object comprises two parts:

- Value
- ID

The ID uniquely identifies the value in the complex data object.

The value of an element can be retrieved and set by specifying its ID. An example of usage of the complex data type can be a television weekly program guide. In this example there are three dimensions:

- The day of week
- The channel
- The hour of the day

The three dimensions form a unique ID; that is, there cannot be two programs that occur on the same day, on the same channel, and at the same time. The value of all of the elements is the program.

An event attribute can be defined as complex (i.e., its type is a complex data type). A complex attribute can be accessed only through the IComplexAttribute interface. The following sections describe the IComplexAttribute interface methods.

createNew

This method creates a new empty instance of the complex attribute with the same definitions (type and dimensions).

Syntax

```
IComplexAttribute createNew() const
```

Parameters

Name	Direction	Type	Description
	Return value	IComplexAttribute	The new object

sort

Sorts the data resides in the complex attribute according to its values. The sort order can be determined by the caller.

Syntax

```
void sort(bool i_ascending=true)
```

Parameters

Name	Direction	Type	Description
i_ascending	In	bool	Indicates whether to sort the complex attribute in ascending or descending order.

set(String)

This method is used to set the value of an element of a complex attribute using the element’s ID. If an element with the specified key already exists in the complex attribute, its value is replaced by the new value. Otherwise, a new element with the specified parameters is added to the complex attribute. This method should be used only on a string type complex attribute. Any attempt to use this method on a complex attribute of a type other than String will produce an error.

Syntax

```
void set(const ComplexElementID& i_path, const UtString&
i_value);
```

Parameters

Name	Direction	Type	Description
i_path	In	const ComplexElementID&	The element ID
i_value	In	const UtString&	The value to set the element with

To create a ComplexElementID type attribute, another CodedComplexElementID class, which is inherited from the ComplexElementID attribute is used. An example of creating and using a two dimensional ComplexElementID attribute is given below:

IComplexAttribute IComplex = pEntity->get("Rate quantity");

CodedComplexElementID c (0,0);

IComplex.set(c,NumericType(12));

set(Numeric, UOM)

This method is used to set the value of an element of a complex attribute using the element’s ID. If an element with the specified key already exists in the complex attribute, its value is replaced by the new value. Otherwise, a new element with the specified parameters is added to the complex attribute.

This method should be used only on a numeric complex attribute. Any attempt to use this method on a complex attribute of a type other than numeric will produce an error.

Syntax

```
void set(const ComplexElementID& i_path,
const NumericType& i_value,
const UtString* i_pUOMString = NULL);
```

Parameters

Name	Direction	Type	Description
i_path	In	const ComplexElementID&	The element ID.
i_value	In	const NumericType&	The value to set the element with.
i_pUOMString	In	const UtString*	The UOM of the value. If not specified then the UOM are assumed to be the UOM of the attribute itself.

set(char*, UOM)

This method is used to set the value of an element of a complex attribute using the element's ID. If an element with the specified key already exists in the complex attribute, its value is replaced by the new value. Otherwise, a new element with the specified parameters is added to the complex attribute. This method should be used only on a numeric complex attribute. Any attempt to use this method on a complex attribute of a type other than numeric will produce an error.

Syntax

```
void set(const ComplexElementID& i_path,
const char* i_value,
const UtString* i_pUOMString = NULL);
```

Parameters

Name	Direction	Type	Description
i_path	In	const ComplexElementID&	The element ID.
i_value	In	const char*	The value to set the element with. The value can be a number with a decimal point.
i_pUOMString	In	const UtString*	The UOM of the value. If not specified, then the UOM are assumed to be the UOM of the attribute itself.

get(String)

This method is used to retrieve the value of an element from a complex attribute by its ID. The method returns FALSE if the Complex storage does not contain an element with the specified ID. This method should be used only on a String type complex attribute. Any attempt to use this method on a complex attribute of a type other than String will produce an error.

Syntax

```
bool get(const ComplexElementID& i_path, UtString&
o_value) const;
```

Parameters

Name	Direction	Type	Description
i_path	In	const ComplexElementID&	The element ID.
o_value	Out	UtString& o_value	The value of the element.
	Return Value	Bool	Returns FALSE if the element was not found in the complex attribute.

get(Numeric)

This method is used to retrieve the value of an element from a complex attribute by its ID. The method returns FALSE if the complex storage does not contain an element with the specified ID. This method should be used only on numeric type and date type complex attributes. Any attempt to use this method on a complex attribute of a type other than numeric type and date type will produce an error.

Syntax

```
bool get(const ComplexElementID& i_path,
NumericType& o_value,
const UtString i_pUOMString = 0) const;
```

Parameters

Name	Direction	Type	Description
i_path	In	const ComplexElementID&	The element's ID.
o_value	Out	NumericType& o_value	The value of the element.
i_pUOMString	In	const UtString*	The UOM of the result. If the UOM are not the same the UOM of the attribute then conversion is done. If no UOM are specified then the UOM are assumed to be the UOM of the attribute itself.
	Return Value	bool	Returns FALSE if the element was not found in the complex attribute.

In(Numeric)

This method is used for searching a value in the complex attribute. It returns TRUE if the value is found, otherwise it returns FALSE.

Syntax

```
bool in(const NumericType& i_value) const;
```

Parameters

Name	Direction	Type	Description
i_value	in	const NumericType&	The value to search for
	Return Value	bool	■ TRUE –Value found ■ FALSE – Value not found

In(String)

This method is used for searching a value in the complex attribute. It returns TRUE if the value is found, otherwise it returns FALSE.

Syntax

```
bool in(const UtString& i_value) const;
```

Parameters

Name	Direction	Type	Description
i_value	in	const UtString&	The value to search for
	Return Value	bool	■ TRUE –Value found ■ FALSE – Value not found

begin

This method returns an iterator to the beginning of the complex attribute. The iterator is used to iterate over the complex attribute elements.

Syntax

```
ComplexAttributeIterator begin() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	ComplexAttributeIterator	An iterator to the beginning of the complex attribute

end

This method returns an iterator to the end of the complex attribute. The iterator is used to iterate over the complex attribute elements.

Syntax

```
ComplexAttributeIterator end() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	ComplexAttributeIterator	An iterator to the end of the complex attribute

count

This method returns the number of the elements in the complex attribute.

Syntax

```
long count() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	Long	The number of the elements in the complex attribute

10. ENTITY METADATA

During its initialization, the Pricing Engine reads and analyzes the Implementation Repository (IR) released by the Product Catalog. The IR contains the definition of the entities involved in the pricing process (such as events, performance indicators, and customer offers).

The Pricing Engine uses this metadata from the Implementation Repository as a guide to the defined entities, their attributes, the types of the attributes, and more.

The metadata of an entity can be explored in runtime.

This section describes the metadata classes, which are entity descriptor and attribute descriptor.

EntityDescriptor

This class is for storing the metadata of a single entity. It provides methods for accessing the attribute's entity.

getAttributeDescriptor(Name)

Given an attribute name, this method returns the object that stores the attribute's metadata. If an attribute with the specified name cannot be found, a Null pointer is returned.

Syntax

```
AttributeDescriptor *getAttributeDescriptor(const
UtString& i_attrName)
```

Parameters

Name	Direction	Type	Description
i_attrName	In	const UtString&	The attribute name
	Return value	AttributeDescriptor	The attribute's metadata if found, otherwise Null

getAttributeDescriptor(Index)

Given the index of an attribute, this method returns the object that stores the attribute's metadata. If an attribute with the specified index cannot be found, a Null pointer is returned

Syntax

```
AttributeDescriptor *getAttributeDescriptor(const
unsigned int i_attrIndex)
```

Parameters

Name	Direction	Type	Description
i_attrIndex	In	const unsigned int	The attribute's metadata if found, otherwise Null

startAttrIterator

This method is useful in iterating over the metadata of an entity's attributes. It returns an iterator to the beginning of the entity's sequence of attribute metadata.

Syntax

```
AttrDescriptorIterator startAttrIterator() const
```

note *The AttrDescriptorIterator is a typedef for a CollectionIterator<AttributeDescriptor *>.*

Parameters

Name	Direction	Type	Description
	Return value	AttrDescriptorIterator	An iterator to the beginning of the sequence of attribute metadata

endAttrIterator

This method is useful in iterating over the metadata of an entity's attributes. It returns an iterator to the end of the entity's sequence of attribute metadata. For more information, see Chapter 12, "Utility Classes."

Syntax

```
AttrDescriptorIterator endAttrIterator() const
```

note *The AttrDescriptorIterator is a typedef for a CollectionIterator<AttributeDescriptor *>.*

Parameters

Name	Direction	Type	Description
	Return value	AttrDescriptorIterator	An iterator to the end of the sequence of attribute metadata

startStaticAttrIterator

This method is useful in iterating over the metadata of an entity's static attributes. It returns an iterator to the beginning of the entity's sequence of attribute metadata. For more information, see Chapter 12, "Utility Classes."

Syntax

```
AttrDescriptorIterator startStaticAttrIterator() const
```

note *The AttrDescriptorIterator is a typedef for a CollectionIterator<AttributeDescriptor *>.*

Parameters

Name	Direction	Type	Description
	Return value	AttrDescriptorIterator	An iterator to the beginning of the sequence of attribute metadata

endStaticAttrIterator

This method is useful in iterating over the metadata of an entity's static attributes. It returns an iterator to the end of the entity's sequence of attribute metadata. For more information, see Chapter 12, "Utility Classes."

Syntax

```
AttrDescriptorIterator endStaticAttrIterator() const
```

note *The AttrDescriptorIterator is a typedef for a CollectionIterator<AttributeDescriptor *>.*

Parameters

Name	Direction	Type	Description
	Return value	AttrDescriptorIterator	An iterator to the end of the sequence of attribute metadata

AttributeDescriptor

This class is for storing the metadata of a single attribute. It provides methods for accessing the attribute's metadata.

getType

This method returns the type of the attribute as defined in the Product Catalog. An attribute can be

- Numeric
- String
- Boolean
- Date

Syntax

```
const INTERNALTYPE getType() const
```

Parameters

Name	Direction	Type	Description
	Return value	INTERNALTYPE	The type. Valid values: ■ INTERNALTYPE_String ■ INTERNALTYPE_Datetime ■ INTERNALTYPE_Number ■ INTERNALTYPE_Boolean

getAttrName

The method returns the attribute name as defined in the Product Catalog.

Syntax

```
const UtString& getAttrName() const
```

Parameters

Name	Direction	Type	Description
	Return value	const UtString&	The attribute name

getAttrIndex

This method returns a numeric value (index) that can be used to access the attribute in an optimized manner. (The index is calculated during the Pricing Engine initialization.)

Syntax

```
long getAttrIndex() const
```

Parameters

Name	Direction	Type	Description
	Return value	long	The attribute index

isComplex

This method checks whether an attribute is a complex attribute.

Syntax

```
bool isComplex() const
```

Parameters

Name	Direction	Type	Description
	Return value	bool	TRUE if the attribute is a complex attribute. Otherwise FALSE.

isPseudo

This method checks whether an attribute is a pseudo attribute.

Syntax

```
bool isPseudo() const
```

Parameters

Name	Direction	Type	Description
	Return value	bool	TRUE if the attribute is a pseudo attribute. Otherwise FALSE

11. Error Handling

The Pricing Engine APIs are divided into two categories:

- Business oriented APIs – References business entities or activates business scenarios (e.g., event processing, event creation, Pricing Engine initialization, etc.)
- Technical oriented APIs (low level) – Accessing infrastructure data structures or services (entity methods, complex attribute methods, etc.)

Business Oriented APIs

The business oriented APIs returns a status which specifies exactly what happened during the execution. It is the caller's responsibility to check the return status and act accordingly.

In addition to the status returned by the APIs, the errors are also reported to the Amdocs error reporting mechanism. These errors can be extracted and pour to files or any other required output.

Technical Oriented APIs

The technical oriented APIs use the standard C++ exception mechanism to report errors that occur during runtime, either internally or externally through a third party software it uses (e.g., databases, XML parser, etc.) Business oriented APIs use technical oriented APIs only so that all the errors produced are trapped and converted to understandable statuses.

In addition the errors are also reported to the Amdocs error reporting mechanism. These errors can be extracted and poured to files or any other required output.

An error can have a severity of: Error or a severity of: Fatal.

- Error, as an error's severity, means that the method's activation failed because of an error detected by the Pricing Engine. (For example, when an attribute has no value, an attempt to access the attribute causes an error with severity Error.)
- Fatal, as an error's severity, means that the error occurred in a third-party program and cannot be handled by the Pricing Engine. (For example, when the connection is invalid, an attempt to perform a database operation causes an error with severity Fatal.)

note

Currently, the Pricing Engine does not provide any additional information on the error as a status code or in any similar way.

The following sections describe the type of exceptions the Pricing Engine may generate when an error is detected.

GeneralException

This class is the base class for all the Pricing Engine exception classes. Exceptions that are generated by the Pricing Engine are always derived from this class. This class exception, however, is never generated; the Pricing Engine only generates exceptions of inherited classes.

ErrorException

This class is derived from the general Exception class. It states that an error occurred internally in the Pricing Engine (not in the third party software it uses). Depending on the business the process performs, at times this exception can be handled (e.g., Reject an Event) and the process can continue, and at other times, it cannot.

FatalException

This class is derived from the general Exception class. It states both that an error occurred externally to the Pricing Engine (in the third party software) and that this error cannot be handled. When such an exception is generated, the process will abort.

12. Utility Classes

In the present version, the NumericTypeInfo class is the only utility class.

NumericTypeInfo

For storing numeric-type information, the NumericTypeInfo class is provided. The numeric type information includes:

- Type – A numeric elementary type. When a type is not specified, the numeric elementary type (the base of all numeric elementary types) is assumed.
- UOM – One of the elementary type UOMs.
- Precision – A number between 0 and 16.

An object of this class is used to create a numeric type object with the desired value.

The following section describes the methods of the NumericTypeInfo class.

constructor

Default construct. It is used to create an object of with numeric elementary type which is the base of all numeric elementary types. It has no UOM. The precision it represents is zero.

Syntax

```
NumericTypeInfo()
```

Parameters

None

Constructor(Short)

Create a NumericTypeInfo object initialized with the given precision. Its type is the numeric elementary type which is the base of all numeric elementary types. It has no UOM.

Syntax

```
NumericTypeInfo(short i_precision)
```

Parameters

Name	Direction	Type	Description
i_precision	In	short	The precision of the numeric value

operator =

This operator sets the NumericTypeInfo object with the value of the given parameter.

Syntax

```
NumericTypeInfo& operator=(const NumericTypeInfo&
i_value)
```

Parameters

Name	Direction	Type	Description
i_value	In	const NumericTypeInfo&	The value to be assigned with
	Return Value	NumericTypeInfo&	The new assigned object

getPrecision

This method returns the precision stored in a NumericTypeInfo object.

Syntax

```
short getPrecision() const
```

Parameters

Name	Direction	Type	Description
	Return Value	short	The precision

setPrecision

This method is used for setting a new precision to a NumericTypeInfo object.

Syntax

```
void setPrecision(short i_precision)
```

Parameters

Name	Direction	Type	Description
i_precision	In	short	The precision to set the object with

13. OPERATORS

This chapter describes the methods used for generic iterators (collection iterators) and for complex attribute iterators.

Generic Iterator (Collection Iterator)

The generic iterator is a template class that provides iteration services on ICollection types. The template parameter specifies the type of the objects.

The collection iterator expects that the template parameter will be a type of a pointer.

The following sections describe the methods of the generic iterator.

operator ++()

This method increments the iterator to the next object in the sequence. The iterator can be incremented up to one position past the last object in the sequence.

Syntax

```
CollectionIterator<TYPE> operator ++()
```

Parameters

Name	Direction	Type	Description
	Return Value	CollectionIterator<TYPE>	The iterator

operator *()

This method returns a pointer to the object described by the iterator. Any attempt to apply this method on an iterator that describes the end iterator (which is positioned on the last object in the sequence) produces an error.

Syntax

```
TYPE operator *()
```

Parameters

Name	Direction	Type	Description
	Return Value	TYPE	The object described by the iterator

operator ==()

This method checks whether two iterators from the same sequence are equal.

Syntax

```
bool operator==(const CollectionIterator<TYPE>& i_other)
```

Parameters

Name	Direction	Type	Description
i_other	In	const CollectionIterator< TYPE>	The iterator to compare with
	Return Value	bool	The comparison result: TRUE when equal; otherwise FALSE

operator !=()

This method checks whether two iterators from the same sequence are unequal.

Syntax

```
bool operator!= (const CollectionIterator<TYPE>& i_other)
```

Parameters

Name	Direction	Type	Description
i_other	In	const CollectionIterator<TYPE>	The iterator to compare with
	Return Value	bool	The comparison result: TRUE when equal; otherwise FALSE

Complex Attribute Iterator

This type of iterator describes a complex attribute element that can be randomly accessed in the complex attribute element sequence.

Iteration over complex attribute elements is done using two iterators of this type. The first iterator describes the beginning of the sequence of the complex attribute elements, and the second describes the end of the complex attribute sequence. The iteration is performed over complex attribute elements in this range. A complex attribute iterator is obtained from the IComplexAttribute interface of the complex attribute to be iterated.

The following sections describe the methods of the complex attribute iterator.

operator ++()

This method increments the iterator to the next complex attribute element in the sequence. The iterator can be incremented up to one position over the last element in the sequence.

Syntax

```
ComplexAttributeIterator operator ++()
```

Parameters

Name	Direction	Type	Description
	Return Value	ComplexAttributeIterator	Increments a complex attribute iterator to the next element in the sequence

operator *()

This method returns the complex element object described by the iterator. Any attempt to apply this method on an iterator that describes the end iterator (which is positioned on the last element in the sequence) produces an error.

Syntax

```
ComplexElement operator *()
```

Parameters

Name	Direction	Type	Description
	Return Value	ComplexElement	The complex element described by the iterator

operator ==()

This method checks whether two iterators from the same sequence are equal.

Syntax

```
bool operator==(const ComplexAttributeIterator& i_other)
```

Parameters

Name	Direction	Type	Description
i_other	In	ComplexAttributeIt erator	The complex attribute iterator to compare with
	Return Value	bool	The comparison result: TRUE when Equal; otherwise FALSE

operator !=()

This method checks whether two iterators from the same sequence are not equal.

Syntax

```
bool operator!=(const ComplexAttributeIterator& i_other)
```

Parameters

Name	Direction	Type	Description
i_other	In	ComplexAttributeIterator	The complex attribute iterator to compare with
	Return Value	bool	The comparison result: TRUE when Equal; otherwise FALSE

14. ARCHIVING API

closePIForCycle

This static method is used to close a performance indicator (PI) for a given cycle. The method is activated according to the selected close mode and the cycle data of the given PI. The close method can be one of the following:

- PURGE – The given PI is deleted from the database. This mode should be used only after the PIs were archived. Once a PI was deleted there is no way to restore it.
- RECONSTRUCT – A new PI record is created for the next cycle month/year. The creation process includes copying the data from the input PI parameter, changing the cycle month and year to the next, and activating the initialization handlers of the relevant pricing item type (PIT). Reconstruction of a PI can be performed only on cross cycle PIs, and is performed only in case that the offer from which the PI was created is still active in the required cycle month and year.
- PURGE_AND_RECONSTRUCT – A combination of the purge and reconstruct options.

Parameters

Name	Direction	Type	Description
I_pUsageContext	In	IDBContext*	Database context of the PI where all the changes in the PI should take place.
i_pCustomerContext	In	IDBContext*	Database context of the customer. Customer data is retrieved from this database.
i_PIEntity	In	const PIEntity&	The PI on which the changes takes place. Primary key data from this PI is used to retrieve the PI from the database. The cycle month and year of i_PIEntity is considered as the requested cycle month and year for performing changes.

Name	Direction	Type	Description
i_PICloseMode	In	PICloseMode	Operation modes for closing the PI: PURGE –delete the PI. RECONSTRUCT – Create a PI in the next cycle. PURGE_AND_ RECONSTRUCT: a combination of the two previous modes.

15. CUSTOMIZATIONS

Compression

Pricing Engine provides and exit point to support compression of event binary data store in Oracle. To enable compression, the user must supply a dynamic library that implements the compress and uncompress of the data.

The library name must be written in PM1_CONF_SECTION_PARAM with the following data:

Colum Name	Value	Description
PARAM_CLASS	PE	The application.
SECTION_NAME	PE.Compression	The compression section.
PARAM_NAME	libraryName	The compression library.
PARAM_VALUE	*Library name*	The library name without prefix or suffix. For example, if the library name on an HP machine is libMyComp.sl, then the name in this entry must be MyComp. If the library name in Windows is MyComp.dll, the name in this entry must be MyComp.

The customization library must have two C-type methods for compression and decompression. Prototypes for these methods can be found in the CompressionPrototype.h file under the pub folder.

blobCompress

This method compresses the entity binary data. The method gets two buffers. The first buffer contains the uncompressed data. The second buffer contains space for the compressed data. The method returns TRUE if the compress finishes successfully, FALSE if there was an error.

note

If the method did not compress the data (not because of an error), it must return TRUE. E.g., the method decided not to compress the data – the return value will be TRUE, and o_compressID will be zero).

Syntax

```
bool blobCompress(const char* i_pUncompressBuffer, _int64
i_uncompressBufferSize, char* io_pCompressBuffer, _int64*
io_compressBufferSize, _int64* o_compressID)
```

Parameters

Name	Direction	Type	Description
i_ pUncompressBuffer	in	const char*	Pointer to the BLOB data to be compressed.

Name	Direction	Type	Description
i_ uncompressBufferSize	in	_int64	The size of the uncompressed BLOB buffer.
io_ pCompressBuffer	in/out	char*	the buffer for the compressed data. The caller defines the buffer size.
io_ compressBufferSize	in/out	_int64*	The size of the data in the buffer. The caller defines the max size of the compressed buffer (PE), Tthe method must provide the actual size used.
o_ compressID	out	_int64*	The compress ID. The uncompressing methos uses this value to decide how to uncompress the data. Zero stands for "no compress".
	Return Value	bool	■ TRUE – Success ■ FALSE – Error encountered

blobUncompress

This method uncompresses the entity binary data. The method gets two buffers. The first buffer contains the compressed data. The second buffer contains the space for the uncompressed data. The method returns TRUE if the uncompress finishes successfully, FALSE if there was an error.

note *if the method did not uncompress (not because of an error), it returns TRUE.*

Syntax

```
bool blobUncompress(_int64 i_compressID, const char*
i_pCompressBuffer, _int64 i_compressBufferSize, char*
io_pUncompressBuffer, _int64* io_uncompressBufferSize)
```

Parameters

Name	Direction	Type	Description
i_compressID	in	_int64	The compress ID. Zero stands for "no compress".
i_pCompressBuffer	in	const char*	Pointer to the compressed BLOB data.

Name	Direction	Type	Description
i_compressBufferSize	in	_int64	The size of the compressed BLOB buffer.
io_pUncompressBuffer	in/out	char*	Buffer for the uncompressed data. The caller defines the buffer size.
io_uncompressBufferSize	in/out	_int64*	The data size in the buffer. The caller defines the max size of the uncompressed buffer (PE). The method provides the actual size used.
	Return Value	bool	■ TRUE – Success ■ FALSE – Error encountered

Appendix A. Reasons for Rejects

An event can be rejected for any of several reasons. The following table shows error codes and their corresponding reasons.

note *The error codes are provided by an enumeration called* ErrorCode.

Error Code	Description
MissingCustomerData	Either customer offer data or customer parameter data was not found for the subscriber associated with the event.
MissingProductCatalogData	Product Catalog reference data was not found for the evaluation of an event.
QualificationFailure	For some reason, guiding to service failed for the event.
NoServiceQualified	No service was qualified for the event during guiding to service.
ViolationOfQualificationPolicy	The services qualified for the event violate the qualification policy.
LockedPIs	The performance indicators required for event processing are locked.
ImplementationReject	The event has been rejected by an implementation request.
ExtensionFailure	The execution of an extension function failed as a result of a signal or unhandled exception.
ComputationFailure	For some reason, event computation failed.
VersionOutOfScope	The event is rejected because its version is outside the acceptable range of versions of the implementation (the versions coverage interval).

Document Release Information

Case No.	Service Pack No.	Description of Change
A115992, 300089	SP7	The markSubscriberForRerate method was added to the SubscriberRerate Query Parameters section of Chapter 6. A new API process that returns o_numberOfDispatchedRecords and supports reservations was added.