amdocs rating

Rating 6.0

Extension Functions Programming Guide

amdocs

Document Information

Software Version:	**6.0**
Publication Date:	**January 2005; Updated May 2005 for SP 3**
Catalog Number:	**212247**

Contents

1. INTRODUCTION

Extension functions provide an easy way to extend the core Pricing Engine functionality. New functionality is introduced by implementing an extension function along with the required auxiliary objects.

Scope of This Document

This document describes the Pricing Engine extension function mechanism, including:

- General description of the extension function mechanism
- Introducing a new extension function into the system
- Implementing extension functions correctly and efficiently

Terminology

The following terms and acronyms are used in this document.

Term	Definition
IR	Implementation Repository – A container of definitions, part of the Product Catalog. The repository includes structural definitions of business objects, together with business rules defining all possible pricing item types. IR also includes type definitions, event types, customer parameters, and external record layout definitions. The type system and the event type definitions are hierarchical and extensible.
PI	Performance Indicators – Counters for all accumulated rated events for a subscriber or group of subscribers. Any attribute and amount of the event can be accumulated. Usually, the chargeable attributes, as well as event rates, are accumulated. PIs are used for retaining data associated with allowances, discounts, and budget control schemes.
PIT	Pricing Item Type
PE	Pricing Engine – The "brain" of the Rating component. It performs all the business logic processing for Rating and other pricing-related components, such as the Online Charging authorization function and non-usage rating calculations (recurring and one-time charges). It allows flexible pricing definitions with many options, because most of the business logic is implementation-dependent rather than embedded in the Rating core code.
Pricing Program	Pricing program, in the Rating context, refers to all the Product Catalog objects used for pricing, e.g., data dictionary (containing elementary types, event types, etc.), Implementation Repository, etc.
Product Catalog	Product Catalog contains the pricing definitions of offers and customer packages, including the wide range of variables that comprise them.

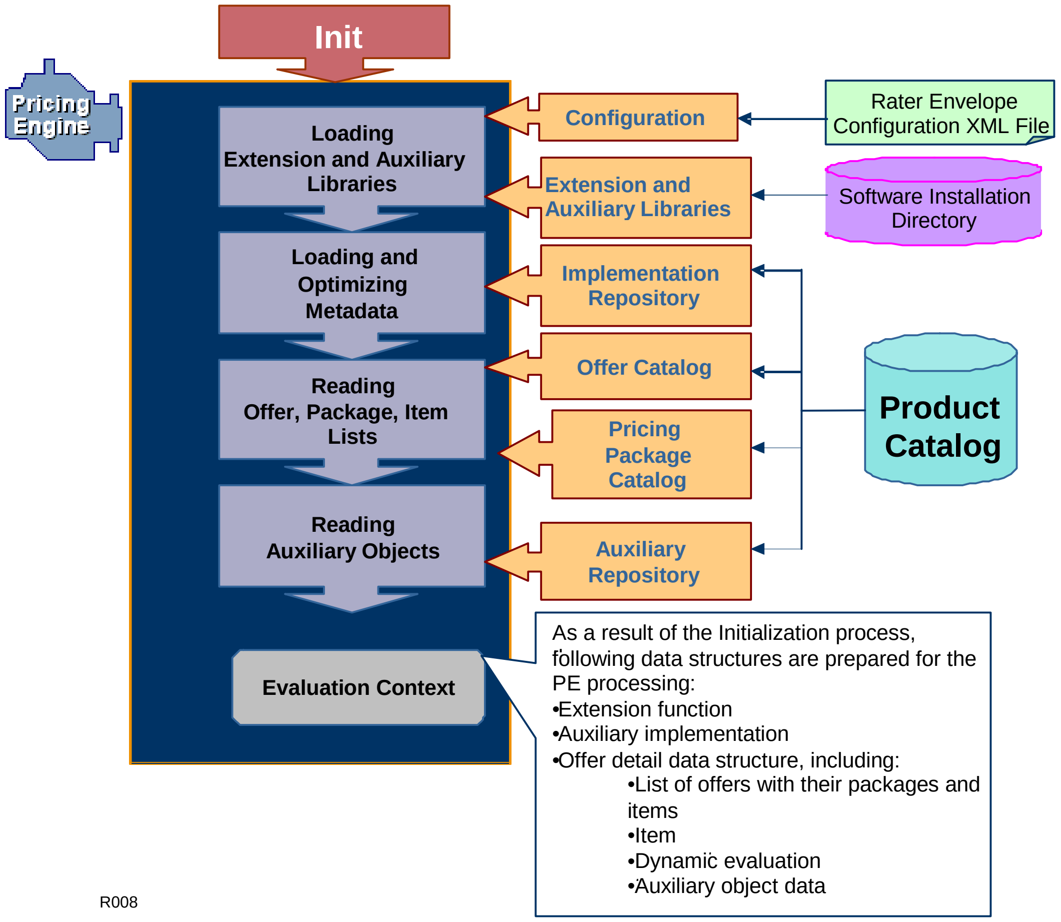
2. Extension Functions – Overview

Extension functions are pre-tested building blocks, provided for use in Product Catalog qualification criteria, qualification cases, and mapping case handlers. Extension functions improve performance of frequently repeated sequences of rating calculations. They can perform calculations based on Pricing Engine entities, entity attributes, and auxiliary objects.

Extension functions are categorized into extension libraries according to their purpose, such as Core, Utility, Control, etc.

Extension Functions and Pricing Engine

The figure below depicts the initialization of Pricing Engine and the mapping of extension functions on initialization.


Init
Pricing Engine
Loading Extension and Auxiliary Libraries
Loading and Optimizing Metadata
Reading Offer, Package, Item Lists
Reading Auxiliary Objects
Evaluation Context
Configuration
Extension and Auxiliary Libraries
Implementation Repository
Offer Catalog
Pricing Package Catalog
Auxiliary Repository
Rater Envelope Configuration XML File
Software Installation Directory
Product Catalog
R008

As a result of the Initialization process, following data structures are prepared for the PE processing:
- Extension function
- Auxiliary implementation
- Offer detail data structure, including:
    - List of offers with their packages and items
    - Item
    - Dynamic evaluation
    - Auxiliary object data

Figure 2-1: Pricing Engine Initialization Phase

Pricing Engine loads the extension libraries during its initialization, and performs all the preparations required to activate the extension functions when they are called by qualification criteria or qualification case handlers.

The diagram below depicts the point at which Pricing Engine activates the extension functions.

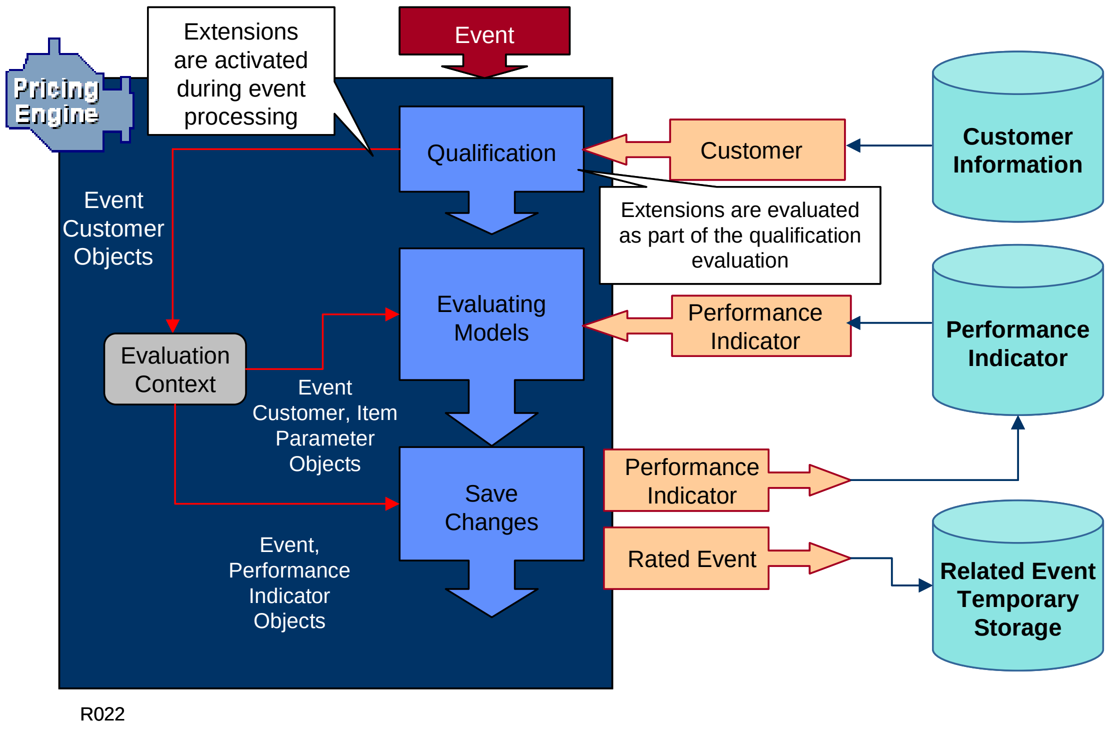

Pricing Engine
Extensions are activated during event processing
Event
Qualification
Customer
Customer Information
Event Customer Objects
Extensions are evaluated as part of the qualification evaluation
Evaluating Models
Performance Indicator
Performance Indicator
Evaluation Context
Event Customer, Item Parameter Objects
Save Changes
Performance Indicator
Rated Event
Related Event Temporary Storage
Event, Performance Indicator Objects
R022

Figure 2-2: Pricing Engine Event Processing Phase

During the Pricing Engine event processing phase, the extension functions are called in order to perform the pricing computation.

Extension functions are well defined in the Product Catalog environment, but their implementation is not validated against definitions. It is the programmer's responsibility to ensure that the function definition adheres to its implementation.

Extension Function Types

Extension functions can be:

- Elementary types (complex types and auxiliary types are regarded as elementary types)
- Entity types
- Entity subroutines
- Pseudo-attributes

Each extension function has a name that is unique within the scope of the type for which it is defined.

Elementary Type Extension Functions

The context of an elementary type extension function is limited to the attribute's value and the parameters it receives. An elementary type extension function can be invoked on an attribute of the elementary type or on an attribute of an inherited elementary type. It receives the value of the attribute on which it was invoked as an implicit parameter. The attribute parameter is passed by value (changing the passed value does not change the attribute's value), except for complex values, which are passed by reference.

An elementary type extension function is required to within the following contexts:

- Logical expression
- Arithmetic expression
- Assignment statement source
- Parameter of an extension function

Entity Type Extension Functions

The context of an entity type extension function is the entity's attributes and the parameters it receives. An entity type extension function can be invoked on an instance of the same entity type or on an instance of an inherited type. The entity extension functions can modify the state of the entity. Some of the entities are updateable (non-persistent: PI, Event, External Record, and Variables) and some are not (persistent: Customer Parameters and Item Parameters). It is illegal to modify the persistent entities at runtime. However, there is no protection against such modifications.

The entity type extension function can be invoked from within the following contexts:

- Logical expression
- Arithmetical expression
- Right operand of an assignment statement
- Parameter of an extension function

Entity Subroutine Extension Functions

Entity subroutine extension functions are very similar to the entity type extensions from the implementation point of view. The only major difference is that this type of extension functions does not return a value. As a result, extension functions of this type can be used only as statements, and not as value operands in logical expressions.

Pseudo-attribute Extension Functions

A pseudo-attribute extension function is a unique type of extension representing a computed attribute that has no persistency. The pseudo-attribute behaves as any other attribute, and can be used as an entity attribute. When a pseudo-attribute is accessed, its business logic is activated. The pseudo-attribute must return a value of the elementary type. An example of such an attribute is the Segments attribute of the Event entity.

The pseudo-attribute type functions can be invoked from within the following contexts:

- Logical expression
- Arithmetic expression
- Assignment statement source or target
- Parameter of an extension function

Usage Examples

This section provides examples of extension function usage.

Entity and Elementary Type Usage

Assume that it is necessary to transform a ZIP code to an area code, which will be compared with the area code appearing on an event to determine whether or not the event originated in the subscriber’s home zone. Assume also that the operator maintains a table, which maps ranges of ZIP codes to area codes.

The following steps implement this functionality:

1. Define the “ZipToAreaCode” extension function. This function is defined on the ZIP code; therefore, it is an elementary type extension function returning the area code.
2. Determine whether the function belongs to an existing extension library or to a new one. For defining new extension libraries, see *Appendix B*.
3. Implement the “ZipToAreaCode” class. Implement the algorithm for transforming the ZIP code to area code using the database table.
4. Create a qualification criterion called “Home Zone QC” in Implementation Repository. The qualification criterion uses the function to compute the location area from the ZIP code and compare the results with the area where the event originated. The qualification criterion can be associated with the “Home Zone” PIT in the voice event qualification.

Pseudo-attribute Usage

Assume that a voice call extended over a period of time. The call was broken into time segments according to the pricing of the different periods. The segments were stored in pseudo-attributes to be used for calculating the charge as follows:

$$Charge = \frac{Segment}{Quantity} * Rate$$

The following steps implement the charge calculation:

1. Define the “ChargeableSegments” extension function.
2. Implement the “ChargeableSegments” class. Implement the algorithm handling the segments.
3. Create a statement on a qualification case in Implementation Repository. The statement uses the “Chargeable Segment” pseudo-attribute to receive the segment value in order to calculate the charge.

3. Implementing Extension Functions

There are two stages in creating extension functions: definition and coding. The definition and usage of extension functions is performed in Product Catalog and described in the *Product Catalog Specification* and *Product Catalog User Guide* documents.

The extension function coding involves the following:

- Writing a C++ class
- Registering the new C++ class in the appropriate extension library, to enable its use by Pricing Engine

Extension Function Declaration

Pricing Engine provides several classes that serve as base for all extension functions. Each class is designed for a specific type of extension functions. The extension function developer inherits one of these classes according to the required extension function type (entity, elementary, or pseudo), and to whether the function returns a value.

Some of the Pricing Engine extension function base classes are templates which receive template parameters. Template parameters can be Product Catalog elementary types, auxiliary types, auxiliary value types, or complex types.

The Product Catalog elementary type inherits one of the following Product Catalog core elementary types:

- Numeric
- Date
- String
- Boolean

For each of the above Product Catalog core elementary types, Pricing Engine has a representative class:

- Numeric – NumericType
- Date – DateType
- String – UtString
- Boolean – BooleanType

All other types are mapped as follows:

- Complex type – Represented by the Pricing Engine IComplexAttribute class (Pricing Engine does not support a complex type abstraction)
- Auxiliary type – Represented by the Pricing Engine AuxiliaryObjectType class
- Auxiliary value type – Represented by the Pricing Engine AuxiliaryValueType class (Pricing Engine does not support a complex type abstraction)

note *For more information regarding the above types, see the* Pricing Engine API *document.*

The template parameters must be determined according the above Pricing Engine types; for example, the CatalogID elementary type inherits the Numeric elementary type, and is, therefore, represented by the Pricing Engine NumericType class.

Elementary Type Method Declaration

To implement an elementary type extension function, one should inherit the Pricing Engine ElementaryTypeMethodImplBase class. The ElementaryTypeMethodImplBase class is a template class. Its template parameters specify the elementary type on which the extension function is defined, and the type of the function's return value.

For example, the following class declaration establishes an elementary method extension function, GetYear, defined on the Date elementary type that returns the year as a number.

```
class GetYear: public ElementaryTypeMethodImplBase
<DateType,NumericType>
{
public:
      // CTOR
      GetYear();
      // DTOR
      virtual ~GetYear();
      // Create a new of ‘GetYear’ instance and return it
      virtual IClonableNode* clone() const;
      // Calculate the year and return it.
      virtual NumericType doEval(DateType
i_attributeValue) const;
};
```

Entity Subroutine Declaration

An entity subroutine is an entity method extension function which does not have a return value. To implement an entity subroutine, one should inherit the Pricing Engine EntitySubroutineImplBase class.

For example, the following class declaration establishes an entity subroutine extension function, RejectEvent. The entity on which the function is declared is Event, and the effect on the Event is changing its status to “rejected.”

```
class RejectEvent: public EntitySubroutineImplBase
{
public:
    // CTOR
    RejectEvent();
    // DTOR
    virtual ~RejectEvent();
    // Create a new ‘RejectEvent’ instance and return it.
    virtual IClonableNode* clone() const;
    // Reject the Event.
    virtual void doEval(Entity* i_pEntity) const;
};
```

Entity Method Declaration

To implement an entity method extension function, one should inherit the Pricing Engine EntityMethodImplBase class. The EntityMethodImplBase class is a template class. Its template parameter specifies the type of the function’s return value.

For example, the following class declaration establishes an entity method extension function, GetOfferID. The function is declared on the Performance Indicator (PI) entity. It returns a numeric value representing the Offer ID.

```
    class GetOfferID: public
    EntityMethodImplBase<NumericType>
{
public:
    // CTOR
    GetOfferID();
    // DTOR
    virtual ~GetOfferID();
    // Create a new ‘GetOfferID’ instance and return it.
    virtual IClonableNode* clone() const;
    // Calculate the offer ID and return it.
    virtual NumericType doEval(Entity* i_pEntity) const;
};
```

Pseudo-attribute Declaration

To implement a pseudo-attribute, one has to inherit the Pricing Engine PseudoAttributeMethodImplBase class. The PseudoAttributeMethodImplBase class is a template class. Its template parameter specifies the attribute's elementary type.

For example, the following class declaration establishes a pseudo-attribute, PseudoOfferID, defined on an attribute of the Offer ID type (numeric).

```
class PseudoOfferID: public
PseudoAttrMethodImplBase<NumericType>
{
 public:
// CTOR
      PseudoOfferID();
// DTOR
      virtual ~PseudoOfferID();
// Create a new 'PseudoOfferID' instance and return it.
   virtual IClonableNode* clone() const;
// Returns whether the Pseudo-attribute was initialized
// or not
      virtual bool isNull() const;
      protected:
// Calculate the Offer ID.
virtual void doGet(const Entity* i_pEntity, NumericType&
o_offerID);
virtual NumericType doGet(const Entity* i_pEntity);
};
```

Interface Methods

There are several interface methods that must be implemented by all extension function types except pseudo-attribute type. These interface methods enable common handling mechanism for all extensions.

Subsequent sections describe the common interface methods.

Init

An extension function is initialized during the Pricing Engine initialization. Although different instances of some extension functions can be used many times in different places in Pricing Engine, these functions are initialized only once.

Init is used to load reference data required during extension function execution, in order to shorten the execution time.

The reference data must be stored in C++ static member of any desired type, to be shared among all instances of this class rather than duplicated. Non-static members must not be used because the extension function logic may be executed concurrently in several threads.

Extension initialization errors must be reported to an error mechanism. In addition, a C++ exception of the Pricing Engine FatalException type must be generated. In this case, Pricing Engine will stop its initialization and return an initialization failure status.

Implementation of this method is optional.

Syntax

```
virtual void init();
```

Parameters

None

Example

```
void ExtensionExample::init()
{
  // Load reference data from database
  if(loadReferenceData() != true)
  {
      ERR_Report_0
      (EXTENTION_FAILD_TO_LOAD_REFERENCE_DATA);
      throw FatalException();
  }
}
```

Clone

Pricing Engine generates computation trees for business logic defined in a pricing program. An invocation of an extension function defined in the pricing program is interpreted as a computation node in the tree. Different invocations of the same extension function in the pricing program are interpreted as different instances of the computation node that represents the extension function.

The clone method is invoked during the Pricing Engine initialization phase whenever Pricing Engine needs a new instance of the extension function to build the computation tree.

Implementation of this method is mandatory; it creates a new instance of the class. Memory allocation for the cloned object is the responsibility of the caller (Pricing Engine).

Syntax

```
virtual IClonableNode* clone() const = 0;
```

Parameters

Name	Direction	Type	Description
	Return Value	IclonableNode *	The cloned object

Example

```
IClonableNode* GetStartPeriodValue::clone() const
{
      return new GetStartPeriodValue();
}
```

doEval (Elementary Type)

The doEval Elementary Type method is invoked during Pricing Engine processing (activation of the pricing program logic) that can be part of event processing, event extract, etc. It is invoked on an elementary type extension function whenever it should be computed according to the pricing program. The implementation of this method must be thread-safe; therefore, it can use only static members.

The method implementation must return a value calculated from the attribute's data, the extension function parameters, and the read-only entities in the execution context, i.e., customer offers, customer parameters, and item parameters. These entities do not always exist in the execution context; for example, item parameters do not exist in the event extract.

It is very important to optimize the extension function code because it affects the Pricing Engine performance.

The method receives a value of the attribute on which the extension function was invoked as an implicit parameter, and must return a value of the declared type. Modification of the parameter does not modify the attribute value except for complex attributes.

Syntax

```
virtual <ReturnType> doEval(<AttributesType>
i_attributeValue) const;
```

Parameters

Name	Direction	Type	Description
i_attributeValue	In	The attribute type (defined as a template parameter)	The attribute value
	Return Value	The return value type (defined as a template parameter)	The return value

Example

This is an elementary type function defined on a string attribute that stores date in a string format. It returns the date as a DateType object.

```
DateType ToDate::doEval(UtString i_attributeValue) const
{
      DateType date;
      date.setValue(i_attributeValue);
      return date;
}
```

doEval (Entity)

The doEval Entity method is invoked during Pricing Engine processing (activation of the Pricing Engine logic), which can be part of event processing, event extract, etc. It is invoked on an entity extension function whenever it should be computed according the pricing program. The implementation of this method must be thread-safe; therefore, it can use only static members.

The implementation should return a value calculated from the entity attributes, the extension function parameters, and the read-only entities in the execution context, that is, customer offers, customer parameters, and item parameters. These entities do not always exist in the execution context; for example, item parameters do not exist in the event extract. The implementation can modify the value of the entity's attributes.

It is very important to optimize the extension function code because it affects the Pricing Engine performance.

The method receives the entity on which the extension function was invoked as an implicit parameter, and must return a value of the declared type.

Syntax

```
virtual <ReturnType> doEval(Entity *i_pEntity) const;
```

Parameters

Name	Direction	Type	Description
i_pEntity	In	Entity *	A pointer to the entity object
	Return Value	The return value type (defined as a template parameter)	The return value

Example

This is an entity type method extension function with a defined PI entity. It calculates the effective offer and returns a numeric type result containing the effective Offer ID.

```
NumericType GetOfferID::doEval(Entity* i_pEntity) const
{
      DateType effectiveOndate;
      this->getParameter(DATE_PARAM, effectiveOndate);

//return the offer ID that matching the PI item ID
NumericType effectiveOfferID =
getEffectiveOfferID(i_pEntity, effectiveOnDate);
      return effectiveOfferID;
}
```

doEval (Entity Subroutine)

The doEval Entity Subroutine method is invoked during Pricing Engine processing (activation of the Pricing Engine logic), which can be part of event processing, event extract, etc. It is invoked on an entity extension function whenever it should be computed according the pricing program. The implementation of this method must be thread-safe; therefore, it can use only static members.

The method can access the entity attributes, extension function parameters, and the read-only entities in the execution context (i.e. customer offers, customer parameters, and item parameters). These entities do not always exist in the execution context; for example, item parameters do not exist in the event extract. The implementation can modify the value of the entity's attributes.

It is very important to optimize the extension function code because it affects the Pricing Engine performance.

The method receives, as an implicit parameter, the entity on which the extension function was invoked, and does not return a value.

Syntax

```
virtual void doEval(Entity *i_pEntity) const;
```

Parameters

Name	Direction	Type	Description
i_pEntity	In	Entity *	A pointer to the entity object

Example

This is an entity subroutine extension function defined on the Event entity. This method adds a subscriber to the Rerate list (SUBSCRIBER_RERATE table).

```
void MarkSubscriberForRerate::doEval(Entity* i_pEntity)
const
{
 //get general parameters needed for performing this
 // activity
 …
 //get Event Related parameters
 NumericType cycleCode;
 NumericType customerID;
 NumericType subscriberID;
 i_pEntity->get(RATED_EVENT_CYCLE_CODE_ATTRIBUTE,
cycleCode);
 i_pEntity-
>get(RATED_EVENT_CUSTOMER_ID_ATTRIBUTE,customerID);
 i_pEntity-
>get(RATED_EVENT_SUBSCRIBER_ID_ATTRIBUTE,subscriberID);
 if(SubOfferLevel::
```

```
IsCustomerLevel(subscriberID,cycleCode,cycleMonth,cycleYe
ar))
          subscriberID = 0;
 //Update the Subscriber for rerate record.
 SubscriberRerateRecord subRerateRec;
 subRerateRec.setCycleCode(cycleCode);
 subRerateRec.setCycleMonth(cycleMonth);
 …
 //Get DB connection
 …
 //Insert the subscriber for rerate recorded for the
 // rated event table.
 DLOracleMarkSubscriberForRerate::
   insert(*pDBConnection,subRerateRec);
}
```

Release

Extension functions are released during the Pricing Engine re-initialization and termination. Although different instances of an extension function can be used many times in different places in the pricing program, an extension function is released only once.

The release method releases the resources acquired by the extension function during its initialization.

Implementation of this method is optional.

Syntax

```
virtual void release();
```

Parameters

None.

Example

```
void ExtensionExample::release()
{
  // Release reference data
  if(releaseReferenceData() != true)
  {
      ERR_Report_0
      (EXTENTION_FAILD_TO_RELEASE_REFERENCE_DATA);
      throw FatalException();
  }
}
```

Pseudo-attribute Interface Methods

Pseudo-attribute is a unique type of extension function. Unlike all other extensions, this type represents data calculated at runtime and treated as an attribute, rather than as a pure function. To support these characteristics, the pseudo-attribute interface provides methods such as isNull, get, and set, for querying and updating the attribute. A pseudo-attribute extension function must override the internal implementation of these methods by implementing doGet and doSet.

note

Pseudo-attributes implement the clone method (as do all extension functions). For description of this method, see the Clone section above.

isNull

This method provides an indication whether or not the pseudo-attribute contains a value. It is a mandatory method.

Syntax

```
virtual bool isNull() const
```

Parameters

Name	Direction	Type	Description
	Return Value	bool	False if the pseudo-attribute contains a value

Example

```
bool ChargeableSegments::isNull() const
{
       return m_segmentsStorage.isEmpty();
}
```

doGet

This method retrieves the pseudo-attribute content by performing the appropriate computation. It is a protected method called by the get() interface method.

Implementing this method is optional; although if it is not implemented, the pseudo-attribute cannot be extracted.

Syntax

```
virtual void doGet(const Entity* i_pEntity, BasicType&
o_value);
```

Parameters

Name	Direction	Type	Description
i_pEntity	In	const Entity*	The entity on which the pseudo-attribute is defined
o_value	Out	BaseType&	The pseudo-attribute value (template parameter)

Example

This method calculates the content of a numeric type pseudo-attribute and returns its value.

```
void PseudoOfferID::doGet(const Entity* i_pEntity,
BasicType& o_value)
{
      return getRatingContext().getOfferID();
}
```

doSet

This method sets the content of the pseudo-attribute.

Implementing this method is optional. If it is not implemented, the pseudo-attribute is calculated during a get request.

Syntax

```
virtual void doSet(Entity* i_pEntity, const BasicType&
i_value);
```

Parameters

Name	Direction	Type	Description
i_pEntity	In	const Entity*	The entity on which the pseudo-attribute is defined
i_value	In	BaseType&	The pseudo-attribute value (template parameter)

Example

This method sets the content of the Chargeable pseudo-attribute to contain the current charge.

```
void CurrentChargeableAttribute::doSet(Entity* i_pEntity,
      const UtString& i_CurrentChargeableAttribute)
{
      m_CurrentChargeableAttribute =
i_CurrentChargeableAttribute;
}
```

Service Mechanism

The Service mechanism enables Pricing Engine and external envelopes to provide data and activity services, such as customer database service, to clients within Pricing Engine, that is, the extension functions.

The Service mechanism comprises three major components:

- Services – Elements that provide data and activity services.
- Service Providers –Manage and provide services. Responsible for supplying the desired service to the client extension functions.
- Service Subscription mechanism – Responsible for subscription of client extension functions to a desired service, and for enabling activation of subscribers by services.

The services can be accessed during the following extension operations:

- Initialization – An extension can subscribe to a service in Service Subscriber Manager if it needs to use a service in the computation phase. The initServices method of the extension function is invoked on the extension for each subscribed service to initialize working with that service. For example, the extension can prepare SQL statements on a database service to be used later, when the extension is evaluated.

  The Service mechanism guarantees that the service to which the extension subscribed and initialized during the Initialization phase is delivered in the computation phase (doEval method).
- Computation – An extension can access the services to which it has subscribed, e.g., activate prepared SQLs.
- Release – All services to which an extension function has subscribed must be released, and the resources allocated for the services must be freed, such as prepared statements, bind variables, etc.

Service Base Interface

All services have a common interface enabling the retrieval of basic information about the service identity (service type and name).

getType

This interface gets the service type.

Syntax:

```
ServiceType getType() const;
```

Parameters:

Name	Direction	Type	Description
	Return Value	ServiceType	Enumeration representing the type of service. Service types: ■ DB_SERVICE_TYPE – Database service.

getName

This interface gets the service name.

Syntax:

```
const UtString& getName() const;
```

Parameters:

Name	Direction	Type	Description
	Return Value	const UtString&	The service name

Service Provider Interface

The IServicesProvider class provides an interface for handling services. This interface is used by extension functions during the computation phase to retrieve the required service.

Syntax

```
virtual ServiceBase* getService(const UtString &
i_serviceName,ServiceType i_serviceType)=0;
```

Parameters

Name	Direction	Type	Description
i_serviceName	In	const UtString &	The service name
i_serviceType	In	ServiceType	Enumeration representing the type of service
	Return Value	ServiceBase*	The required service instance

Subscription for a Service

An extension function that needs to use a service has to register for that service with the ServiceSubscribersManager class by activating the ServiceInitSubscriberActivator static interface with the extension details and the required service identifiers (service name and type). The activator performs the the extension function subscription for a service, and is responsible for calling the initService() method. ServiceInitSubscriberActivator is a template class that receives the extension class name as a template parameter.

An extension function that attempts to subscribe to a specific service more than once will fail.

The subscription operation returns “false” if it encounters an error. It is the developer’s responsibility to ensure that this error is reported to the error mechanism, and to generate a C++ exception of the Pricing Engine FatalException type. A subscription error during Pricing Engine initialization stops the initialization and returns an “initialization failure” status.

Syntax

```
static bool ServiceInitSubscriberActivator<typename
ExtensionName>::subscribe(SubscriberObject
*i_pSubscriberObj, ServiceType i_serviceType, const
UtString& i_serviceName)
```

Parameters

Name	Direction	Type	Description
i_pSubscriberObj	In	MethodDelegators *	A pointer to the extension function object to be subscribed for the service
i_serviceType	In	ServiceType	The service type; can be: DB_SERVICE_TYPE
i_serviceName	In	const UtString&	The service name
	Return Value	bool	"False" if the operation fails, "True" otherwise

Example

This function subscribes for a database service for a customer.

```
void MatchCustomerGroupsWithResource::init()
{
// Subscribe for the Customer database service
if(ServiceInitSubscriberActivator<MatchCustomerGroupsWithRes
ource>
::subscribe(this,DB_SERVICE_TYPE,"Customer") == false)
 {
ERR_Report_0 (RATER_FAILED_TO_SUBSCRIBE_FOR_SERVICE);
throw FatalException();
 }
…
}
```

Unsubscribing from a Service

An extension function that has subscribed to services during initialization must unsubscribe from all the services when it is released (when Pricing Engine is re-initialized or terminated). Each subscription requires a separate un-subscription.

Extension functions unsubscribe from a service at the ServiceSubscribersManager class by activating the ServiceInitSubscriberActivator static interface with the extension details and the service identifiers (service name and type). The activator unsubscribes the extension function from the service, and is responsible for calling the releaseService() method. ServiceInitSubscriberActivator is a template class that receives the extension class name as a template parameter.

The unsubscription operation returns “false” if it encounters an error. It is the developer’s responsibility to report this error to the error mechanism, and to generate a C++ exception of the Price Engine type. An unsubscription error during Pricing Engine re-initialization or termination stops the re-initialization or termination and returns an “initialization failure” status.

Syntax

```
static bool ServiceInitSubscriberActivator<typename
ExtensionName>::unsubscribe(SubscriberObject *
i_obj,ServiceType i_serviceType,const UtString&
i_serviceName)
```

Parameters

Name	Direction	Type	Description
i_pSubscribedObj	In	MethodDelegators *	A pointer to the extension function object subscribed for the service
i_serviceType	In	ServiceType	The service type; for example DB_SERVICE_TYPE
i_serviceName	In	const UtString&	The service name
	Return Value	bool	“False” if the operation fails; “True” otherwise

Example

This function unsubscribes a customer from a database service.

```
void MatchCustomerGroupsWithResource:: release()
{
  …

if(ServiceInitSubscriberActivator<MatchCustomerGroupsWith
Resource>
::unsubscribe(this,DB_SERVICE_TYPE,"Customer") == false)
 {
ERR_Report_0 (RATER_FAILED_TO_UNSUBSCRIBE_FROM_SERVICE);
throw FatalException();
 }
   …
}
```

Unsubscription from All Services

An extension function that has subscribed to multiple services during its initialization must unsubscribe from all the subscribed services when it is released (when Pricing Engine is re-initialized or terminated). The unsubscription can be done for all the subscribed services in a single call.

The extension functions unsubscribe from services at the ServiceSubscribersManager class by activating the ServiceInitSubscriberActivator static interface with the extension details. The activator unsubscribes the extension function from all its services, and is responsible for calling the releaseService method for each service from which the extension is unsubscribed. ServiceInitSubscriberActivator is a template class that receives the extension class name as a template parameter.

The unsubscription operation returns “false” if it encounters an error. It is the developer’s responsibility to report this error to the error mechanism, and to generate a C++ exception of the Pricing Engine type. An unsubscription error during Pricing Engine re-initialization or termination will stop the re-initialization or termination and return an “initialization failure” status.

note *When the extension is subscribed to few services, it is more efficient to use the unsubscribe interface than the unsubscribeAll interface.*

Syntax

```
static bool ServiceInitSubscriberActivator<typename
ExtensionName>::unsubscribeAll(SubscriberObject *
i_obj,ServiceType)
```

Parameters

Name	Direction	Type	Description
i_pSubscribedObj	In	MethodDelegators *	A pointer to the extension function object subscribed for the service

Example

This example unsubscribes from all services using the MatchCustomerGroupsWithResource extension function.

```
void MatchCustomerGroupsWithResource:: release()
{
      …
  …

if(ServiceInitSubscriberActivator<MatchCustomerGroupsWithRe
source>
::unsubscribeAll(this) == false)
 {
ERR_Report_0
(RATER_FAILED_TO_UNSUBSCRIBE_FROM_ALL_SERVICES);
throw FatalException();
 }
   …
}
```

Initializing a Service

An extension function is activated by Pricing Engine to initialize the services to which it has subscribed. The extension function must perform the actions required for preparing the service to be used during its computation. This type of initialization is activated only if the extension function has subscribed to a service during the extension initialization.

The method receives the service to be initialized. The extension function is activated separately for each service it has subscribed to. The service can be identified according to its name and type.

An error in service initialization must be reported to the error mechanism to return “False”; otherwise, “True” is returned. When an error occurs during the service initialization, Pricing Engine initialization terminates unsuccessfully.

Syntax

```
virtual bool initService(ServiceBase& i_service) const;
```

Parameters

Name	Direction	Type	Description
i_service	In	ServiceBase&	The service that needs to be initialized
	Return Value	bool	The status of the initialization (“False”= failure)

Example

This example initializes a service using the MatchCustomerGroupsWithResource extension function.

```
bool MatchCustomerGroupsWithResource::
initService(ServiceBase& i_service)
{
  …
 if(i_service.getName() != CUSTOMER_DB_SERVICE_NAME)
 {
return false;
 }
   //Init the customer DB service with the extension
function statements
   return initCustomerDBService(i_service);
}
```

Releasing a Service

An extension function is activated to remove its effect on a service when the it unsubscribes from that service. This type of release is evoked only if the extension function unsubscribes from the service during the re-initialization or termination. During this operation, the extension function must release all resources acquired for the service.

The releaseService method is activated separately for each service to be unsubscribed from. The service is provided as an input parameter to the extension function.

A service initialization error must be reported to the error mechanism to return “False”. When an error in the release service occurs during Pricing Engine re-initialization or termination, the Pricing Engine re-initialization or termination stops unsuccessfully.

Syntax

```
virtual bool releaseService(ServiceBase& i_service)
const;
```

Parameters

Name	Direction	Type	Description
i_service	In	ServiceBase&	The service to be released
	Return Value	bool	The status of the release operation (“False” = failure)

Example

This example releases a service using the MatchCustomerGroupsWithResource extension function.

```
bool MatchCustomerGroupsWithResource::
releaseService(ServiceBase& i_service)
{
  …
 if(i_service.getName() != CUSTOMER_DB_SERVICE_NAME)
 {
return false;
 }
   //Release the customer DB service from the extension
   // function statements
   return releaseCustomerDBService(i_service);
}
```

Using Services in Extension Function Computation

In the extension function computation phase, a service provider containing all the relevant services is delivered to the extension function. The extension function can use the given services in its computation.

Extension functions that require a service in this phase must override this form of the doEval method; otherwise, the previous doEval method is activated (for backwards compatibility with the previous extension function implementations).

doEval Elementary Type Method

Syntax:

```
TYPE doEval(ATTR_TYPE i_attributeValue,
IServicesProvider& i_ServiceProvider) const
```

Parameters:

Name	Direction	Type	Description
i_attributeValue	In	The attribute type (defined as a template parameter)	The attribute value
i_servicesProvider	In	IservicesProvider&	The service provider
	Return Value	The return value type (defined as a template parameter)	The return value

doEval Entity Method

Syntax:

```
TYPE doEval(Entity* i_pEntity, IServicesProvider&
i_serviceProvider) const
```

Parameters:

Name	Direction	Type	Description
i_pEntity	In	Entity *	A pointer to the entity object
i_servicesProvider	In	IservicesProvider&	The service provider
	Return Value	The return value type (defined as a template parameter)	The return value

doEval Entity Subroutine Method

Syntax:

```
virtual void doEval(Entity* i_pEntity, IServicesProvider&
i_serviceProvider) const
```

Parameters:

Name	Direction	Type	Description
i_pEntity	In	Entity *	A pointer to the entity object
i_servicesProvider	In	IservicesProvider&	The service provider

Example

This example uses a service in doEval of an entity method extension function.

```
BooleanType MatchCustomerGroupsWithResource
  ::doEval(Entity* i_pEntity,IServicesProvider&
i_servicesProvider) const
{
      …
      ServiceBase* pService =

i_servicesProvider.getService(“Customer”,DB_SERVICE_TYPE)
;
      …
}
```

4. Extension Function Input Parameters

Extension functions can receive parameters containing information required for calculations. The parameters can be of any elementary type, auxiliary type, or auxiliary value type, for example: numeric, Boolean, auxiliary, etc. They can be ether static or dynamic.

All extension functions inherit the ability to receive parameter values from the MethodDelegators class. This class provides a public interface for this purpose.

Static Parameters

Static parameters are part of the extension function interface. They are explicitly declared in the pricing program; therefore, their type and origin are known. The static parameters are described in the subsequent sections.

In order to receive these parameters, an extension function must know the parameter name and type as it is defined in the pricing program. Then, it can address the parameter contents by using the MethodDelegators methods described in subsequent sections.

Pricing Engine does not convert parameter values from one type to another (e.g., from numeric to string). An attempt to retrieve a parameter value of one type with a method defined for a different type results in an error.

The parameter name provided to the methods must be exactly as defined in the pricing program. Any attempt to access a parameter with an incorrect name results in an error.

GetParameter (Numeric)

This form of the getParameter method is used to retrieve a value of a parameter of Numeric type, according its name.

Syntax

```
void getParameter(const UtString& i_parameterName,
NumericType& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
o_value	Out	NumericType&	The parameter value

Example

This example uses the getParameter method to retrieve a numeric input.

```
void Round::doEval(…)
{
   NumericType roundMethod;
   this->getParameter(METHOD_TYPE_PARAM,roundMethod);
…
}
```

getParameter (String)

This form of the getParameter method is used to retrieve the value of a parameter of String type.

Syntax

```
void getParameter(const UtString& i_parameterName,
UtString& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
o_value	Out	UtString&	The parameter value

Example

This example uses the getParameter method for retrieving a string input.

```
void MapSegmentsToValueListType:: doEval(…)
{
   UtString value;
   this->getParameter(VALUE_PARAM,value);
…
}
```

getParameter (Date)

This form of the getParameter method is used to retrieve the value of a parameter of Date type.

Syntax

```
void getParameter(const UtString& i_parameterName,
DateType& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
o_value	Out	DateType&	The parameter value

Example

This example uses the getParameter method for retrieving an event date of type DateType.

```
void GetDistanceToPeriodEnd:: doEval(…)
{
   DateType eventDate;
   this->getParam(EVENT_DATE_PARAM, eventDate);
…
}
```

getParameter (Boolean)

This form of the getParameter method is used to retrieve the value of a parameter of Boolean type.

Syntax

```
void getParameter(const UtString& i_parameterName,
BooleanType& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
o_value	Out	BooleanType&	The parameter value

Example

This example uses the getParameter method for retrieving a flag of type BooleanType.

```
void ExtensionExample:: doEval(…)
{
BooleanType flag;
this->getParameter(IS_VALID_PARAM, flag);
…
}
```

getParameter (AuxiliaryObjectType)

This form of the getParameter method is used to retrieve the value of a parameter of Auxiliary Object type.

Syntax

```
void getParameter(const UtString& i_parameterName,
AuxiliaryObjectType& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
o_value	Out	AuxiliaryObjectType&	The parameter value

Example

This example uses the getParameter method for retrieving a period set of type AuxiliaryObjectType.

```
void MapSegmentsToStartPeriod:: doEval(…)
{
AuxiliaryObjectType auxiliaryObjectTypePeriods;
this->getParameter(PERIOD_PARAM,
auxiliaryObjectTypePeriods);
…
}
```

getParameter (AuxiliaryValueType)

This form of the getParameter method is used to retrieve the value of a parameter of Auxiliary Value type.

Syntax

```
void getParameter(const UtString& i_parameterName,
AuxiliaryValueType& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
o_value	Out	AuxiliaryValueType&	The parameter value

Example

This example uses the getParameter method for retrieving a value of type AuxiliaryValueType.

```
void ExtensionExample:: doEval(…)
{
AuxiliaryValueType value;
this->getParameter(VALUE_PARAM, value);
…
}
```

getNumericParameter

The getNumericParameter method is used to retrieve the value of a numeric parameter whose contents are a string representation of the numeric value.

Syntax

```
NumericType getNumericParam(const UtString&
i_parameterName) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
	Return Value	NumericType	The parameter value

Example

This example uses the getNumericParameter method for retrieving a string representing a numeric value.

```
void Round:: doEval(…)
{
   NumericType roundMethod =
       this->getNumericParameter(METHOD_TYPE_PARAM);
…
}
```

getStringParameter

The getStringParameter method is used to retrieve the value of a string parameter according to its name.

Syntax

```
UtString getStringParam(const UtString& i_parameterName)
const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
	Return Value	UtString	The parameter value

Example

This example uses the getStringParam method to retrieve a value of type String.

```
void MapSegmentsToValueListType:: doEval(…)
{
   UtString value;
value = this->getStringParam(VALUE_PARAM);
…
}
```

getDateParameter

The getDateParameter method is used to retrieve the value of a date parameter according to its name.

Syntax

```
DateType getDateParam(const UtString& i_parameterName)
const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
	Return Value	DateType	The parameter value

Example

This example uses the getDateParam method to retrieve an event date of type DateType.

```
NumericType GetDistanceToPeriodEnd:: doEval(…)
{
DateType eventDate;
eventDate = this->getDateParam(EVENT_DATE_PARAM);
…
}
```

getBooleanParameter

The getBooleanParameter method is used to retrieve the value of a Boolean parameter according to its name.

Syntax

```
BooleanType getBooleanParam(const UtString&
i_parameterName) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
	Return Value	BooleanType	The parameter value

Example

This example uses the getBooleanParam method to retrieve a flag of type BooleanType.

```
void ExtensionExample:: doEval(…)
{
BooleanType flag;
flag = this->getBooleanParam(IS_VALID_PARAM);
…
}
```

getComplexParameter

The getComplexParameter method is used to retrieve the value of a complex parameter according to its name.

Syntax

```
IComplexAttribute getComplexParameter(const UtString&
i_parameterName) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
	Return Value	IComplexAttribute	The parameter value

Example

This example uses the getComplexParameter method to retrieve an accumulation of type IComplexAttribute.

```
void BreakQuantityToStepsByDimAccordingToAcc:: doEval(…)
{
IComplexAttribute accumulationComplex = this->
getComplexParameter(COUNTER_PARAM);
…
}
```

getAuxiliaryParameter

The getAuxiliaryParameter method is used to retrieve the value of an auxiliary (Auxiliary Object) parameter according to its name.

Syntax

```
AuxiliaryObjectType getAuxiliaryParameter(const UtString&
i_parameterName) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
	Return Value	AuxiliaryObjectType	The parameter value

Example

This example uses the getAuxiliaryParameter method to retrieve a period set of type AuxiliaryObjectType.

```
void MapSegmentsToStartPeriod:: doEval(…)
{
AuxiliaryObjectType auxiliaryObjectTypePeriods;
auxiliaryObjectTypePeriods =
       this->getAuxiliaryParameter(PERIOD_SET_PARAM);
…
}
```

getAuxiliaryValueParameter

The getAuxiliaryValueParameter method is used to retrieve the value of an Auxiliary Value parameter according to its name.

Syntax

```
AuxiliaryValueType getAuxiliaryValueParameter(const
UtString& i_parameterName) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
	Return Value	AuxiliaryValueType	The parameter value

Example

This example uses the getAuxiliaryValueParameter method to retrieve a value of type AuxiliaryValueType.

```
void ExtensionExample:: doEval(…)
{
AuxiliaryValueType value =
        this->getAuxiliaryValueParameter(VALUE_PARAM);
…
}
```

Dynamic Parameters

Dynamic parameters are optional parameters that can be passed to an extension function without being declared in the extension function declaration. The number of the dynamic parameters is not limited. The extension function user can pass multiple values or none, and it is the responsibility of the extension function to handle a correct flow in each case. The dynamic parameters are described in subsequent sections.

Dynamic parameters have a set of access methods different from those used for the static parameters. These methods receive the location of the required parameter in the dynamic parameters list. In addition, the number of parameters in the dynamic parameter list can be queried. An attempt to access a parameter with a location out of the range results in an error (the first location is zero and the last location is the number of dynamic parameters minus one).

Pricing Engine does not convert values from one type to another type (e.g., from numeric to string). An attempt to retrieve a parameter value of a type for which the method is not defined produces an error.

GetDynamicParameter (Numeric)

This form of the getDynamicParameter method is used to retrieve the value of a Numeric dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
void getDynamicParameter(long i_index, NumericType&
o_value) const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
o_value	Out	NumericType&	The parameter value

Example

This example gets the first dynamic parameter, which is of type numeric, and which contains the Group ID.

```
BooleanType MatchTwoUserGroupResource::doEval(…)
{
  if (this->getDynamicParametersCount()>0)
  {
  NumericType groupID;
      this->getDynamicParameter(0, groupID);
      …
  }
}
```

getDynamicParameter (String)

This form of the getDynamicParameter method is used to retrieve the value of a String dynamic parameter according to its location in the dynamic parameter list.

This method performs a conversion to string, unlike most getParameters methods. If the parameter cannot be converted to string, an error is produced.

Syntax

```
void getDynamicParameter(long i_index, UtString& o_value)
const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
o_value	Return Value	UtString&	The parameter value

Example

This example gets the first dynamic parameter, which is of type String, and which contains the diminution value.

```
UtString ApplyMappingTableValue::doEval(…)
{
  if (this->getDynamicParametersCount()>0)
  {
UtString dimValue;
   this->getDynamicParameter(0, dimValue);
  …
  }
}
```

getDynamicParameter (Date)

This form of the getDynamicParameter method is used to retrieve the value of a Date dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
void getDynamicParameter(long i_index, DateType& o_value)
const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
o_value	Out	DateType&	The parameter value

Example

This example gets the first dynamic parameter, which is of type Date, and which contains the group effective date.

```
BooleanType MatchCustomerGroupsWithResource::doEval(…)
{
  if (this->getDynamicParametersCount()>0)
  {
    DateType groupEffectiveDate;
    this->getDynamicParameter(0, groupEffectiveDate);
   …
  }
}
```

getDynamicParameter (Boolean)

This form of the getDynamicParameter method is used to retrieve the value of a Boolean dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
void getDynamicParameter(long i_index, BooleanType&
o_value) const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
o_value	Out	BooleanType&	The parameter value

Example

This example gets the first dynamic parameter, which is of type Boolean, and which determines whether to consider special days in the period calculation.

```
NumericType GetDistanceToPeriodEnd::doEval(…)
{
  if (this->getDynamicParametersCount()>0)
  {
   BooleanType considerSpecialDays;
   this->getDynamicParameter(0, considerSpecialDays);
   …
  }
}
```

getDynamicParameter (AuxiliaryType)

This form of the getDynamicParameter method is used to retrieve the value of an Auxiliary Object type dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
void getDynamicParameter(long i_index,
AuxiliaryObjectType& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
o_value	Out	AuxiliaryObjectType&	The parameter value

Example

This example gets the second dynamic parameter, which is of type Auxiliary Object, and which determines whether special days should be included in the calculation.

```
NumericType GetDistanceToPeriodEnd::doEval(…)
{
  if (this->getDynamicParametersCount()>1)
  {
   AuxiliaryObjectType auxObjectTypeSpecialDays;
   this->getDynamicParameter(1,
auxObjectTypeSpecialDays);
  …
  }
}
```

getDynamicParameter (AuxiliaryValueType)

This form of the getDynamicParameter method is used to retrieve the value of an Auxiliary Value type dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
void getDynamicParameter(long i_index,
AuxiliaryValueType& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
o_value	Out	AuxiliaryValueType&	The parameter value

Example

This example gets the first dynamic parameter, which is of type Auxiliary Value, and which contains the target ID.

```
void CreateNotification::doEval(…)
{
  if (this->getDynamicParametersCount()>1)
  {
   AuxiliaryValueType targetId;
   this->getDynamicParameter(1, targetId);
  …
  }
}
```

getDynamicParameter (IComplexAttribute)

This form of the getDynamicParameter method is used to retrieve the value of a Complex Attribute type dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
void getDynamicParameter(long i_index, IComplexAttribute
& o_value) const;
```

Parameters

Name	Direction	Type	Description
i_parameterName	In	const UtString&	The parameter name
	Return Value	IComplexAttribute	The parameter value

Example

This example uses the getComplexDynamicParameter method to retrieve an accumulation of type IComplexAttribute.

```
void ExtensionExample:: doEval(…)
{
IComplexAttribute accumulationComplex =
this->getDynamicParameter(0);
…
}
```

getNumericDynamicParameter

The getNumericDynamicParameter method is used to retrieve the value of a Numeric dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
NumericType getNumericDynamicParameter(long i_index)
const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
	Return Value	NumericType	The parameter value

Example

This example gets the first dynamic parameter, which is of type numeric, and which contains the group ID.

```
BooleanType MatchTwoUserGroupResource::doEval(…)
{
  if (this->getDynamicParametersCount()>0)
  {
      NumericType groupID = this-
>getNumericDynamicParameter(0);
   …
  }
}
```

getStringDynamicParameter

The getStringDynamicParameter method is used to retrieve the value of a String dynamic parameter according to its location in the dynamic parameter list.

This method performs conversion to string, unlike most other getParameter() methods. If the parameter cannot be converted to a string, an error is produced.

Syntax

```
UtString getStringDynamicParameter(long i_index) const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
	Return Value	UtString	The parameter value

Example

This example gets the first dynamic parameter, which is of type String, and which contains the diminution value.

```
UtString ApplyMappingTableValue::doEval(…)
{
  if (this->getDynamicParametersCount()>0)
  {
UtString dimValue = this->getStringDynamicParameter(0);
   …
  }
}
```

getDateDynamicParameter

The getDateDynamicParameter method is used to retrieve the value of a Date dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
DateType getDateDynamicParameter(long i_index) const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
	Return Value	DateType	The parameter value

Example

This example gets the first dynamic parameter, which is of type Date, and which contains the group effective date.

```
BooleanType MatchCustomerGroupsWithResource::doEval(…)
{
  if (this->getDynamicParametersCount()>0)
  {
    DateType groupEffectiveDate =
          this->getDateDynamicParameter(0);
   …
  }
}
```

getBooleanDynamicParameter

The getBooleanDynamicParameter method is used to retrieve the value of a Boolean dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
BooleanType getBooleanDynamicParameter(long i_index)
const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
	Return Value	BooleanType	The parameter value

Example

This example gets the first dynamic parameter, which is of type Boolean, and which determines whether to consider special days in the period calculation.

```
NumericType
GetDistanceToPeriodEnd::doEval(AuxiliaryObjectType
                              i_periodSet)
{
  if (this->getDynamicParametersCount()>0)
  {
   BooleanType considerSpecialDays =
             this->getBooleanDynamicParameter(0);
  …
  }
}
```

getAuxiliaryDynamicParameter

The getAuxiliaryDynamicParameter method is used to retrieve the value of an Auxiliary Object type dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
AuxiliaryObjectType getAuxiliaryDynamicParameter(long
i_index) const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameter list
	Return Value	AuxiliaryObjectType	The parameter value

Example

This example gets the second dynamic parameter, which is of type Auxiliary Object, and which contains special days that should be included in the calculation.

```
NumericType GetDistanceToPeriodEnd::doEval(…)
{
  if (this->getDynamicParametersCount()>1)
  {
   AuxiliaryObjectType auxObjectTypeSpecialDays =
               this->getAuxiliaryDynamicParameter(1);
  …
  }
}
```

getAuxiliaryValueDynamicParameter

The getAuxiliaryValueDynamicParameter method is used to retrieve the value of an Auxiliary Value type dynamic parameter according to its location in the dynamic parameter list.

Syntax

```
AuxiliaryValueType getAuxiliaryDynamicParameter(long
i_index) const;
```

Parameters

Name	Direction	Type	Description
i_index	In	long	The parameter location in the dynamic parameters list
	Return Value	AuxiliaryValueType	The parameter value

Example

This example gets the first dynamic parameter, which is of type Auxiliary Value, and which contains the target ID.

```
void CreateNotification::doEval(…)
{
  if (this->getDynamicParametersCount()>0)
  {
   AuxiliaryValueType targeted =
     this->getAuxiliaryValueDynamicParameter (0);
  …
  }
}
```

getDynamicParametersCount

The getDynamicParametersCount method is used to query the number of dynamic parameters passed to the extension function.

Syntax

```
long getDynamicParametersCount() const;
```

Parameters

Name	Direction	Type	Description
	Return Value	Long	The number of dynamic parameters passed to the extension function

Example

This example gets the number of dynamic parameters.

```
BooleanType MatchTwoUserGroupResource::doEval(UtString
i_resource_1)
{
  long paramCount = this->getDynamicParametersCount();
  if (paramCount)
  {
  …
  }
}
```

Context Entities

The MethodDelegator interface and the Rating context helper class provide a set of methods that enable extension functions to access entities participating in their context run, for example, events and PIs in event processing, and PIs in PI extract.

In Pricing Engine version 6.0, access methods to context entities from MethodDelegator will not be supported (for more information, see Chapter 8, “Deprecated APIs”).

getCustomer

The getCustomer method is used to retrieve the current Customer Parameters entity residing in the Pricing Engine context.

Syntax

```
const Entity&     getCustomer() const
```

Parameters

Name	Direction	Type	Description
	Return Value	Entity&	The Customer Parameters entity

getCustomerOffers

The getCustomerOffers method is used to retrieve the current Customer Offers entity residing in the Pricing Engine context.

Syntax

```
const Entity&    getCustomerOffers() const
```

Parameters

Name	Direction	Type	Description
	Return Value	Entity&	The Customer Offers entity

getItemParameters

The getItemParameters method is used to retrieve the current Item Parameters entity residing in the Pricing Engine context.

Syntax

```
const Entity&    getItemParameters() const
```

Parameters

Name	Direction	Type	Description
	Return Value	Entity&	The Item Parameters entity

getPI

The getPI method is used to retrieve the current PI entity residing in the Pricing Engine context.

Syntax

```
Entity& getPI() const
```

Parameters

Name	Direction	Type	Description
	Return Value	Entity&	The PI entity

getEvent

The getEvent method is used to retrieve the current Event entity residing in the Pricing Engine context.

Syntax

```
Entity& getEvent() const
```

Parameters

Name	Direction	Type	Description
	Return Value	Entity&	The Event entity

Example

An extension function can use entities to retrieve information for performing a computation. The GetOfferName Entity Type extension function uses the Customer Offers entity to find the offer name.

```
UtString GetOfferName::doEval(NumericType i_itemID) const
{
      UtString offerVersionName;
      //Get the customer offers entity
const Entity& customerOffers =
RatingContext::getCustomerOffers();
…
}
```

5. Extension Function Registration

An extension function is declared in a specific extension library (implemented as a shared library). During Pricing Engine initialization, all the extension libraries are loaded to memory, each as an entry point (C function) used for the extension library initialization. The extension library initialization process, init, is activated after Pricing Engine has successfully loaded the library. Extension functions must be registered in Pricing Engine to be used during the computation.

When Pricing Engine interprets the pricing program and an extension function is invoked, Pricing Engine seeks the extension implementation in the list of registered extension functions. If an extension function does not exist, Pricing Engine checks the Enforce Extension Availability configuration parameter. A value of “true” causes the Pricing Engine initialization to fail. A value of “false” sets a pseudo extension function instead of the real one, so that Pricing Engine initialization can continue.

note *At runtime, an error is generated if the computation tries to invoke the pseudo extension function.*

Entity Method and Pseudo-attribute Function Registration

Entity method and pseudo-attribute extension functions are registered in the extension library entry point. The macro in the example below is used for this purpose.

The entity name and the extension function name must be spelled exactly as they are defined in Product Catalog.

Syntax (Macro)

```
REGISTER_ENTITY_EXTENSION (Entity Name, Function Name,
Implementation Class Name)
```

Parameters

Name	Description
Entity Name	The entity on which the extension is defined
Function Name	The extension function name
Implementation Class Name	The class that implements the extension function

Example

```
REGISTER_ENTITY_EXTENSION (Event, Apply free units,
ApplyFreeUnits);
```

Elementary Type Function Registration

Elementary type extension functions are registered in the extension library entry point. The macro in the example below is used for this purpose.

The elementary type name and extension function name must be spelled exactly as they are defined in Product Catalog.

Syntax (Macro)

```
REGISTER_ELEMENTARY_TYPE_EXTENSION (Elementary Type Name,
Function Name, Implementation Class Name)
```

Parameters

Name	Description
Elementary Type Name	The elementary type on which the extension is defined
Function Name	The extension function name
Implementation Class Name	The class that implements the extension function

Example

```
REGISTER_ELEMENTARY_TYPE_EXTENSION (Scale, Get scale leg,
GetScaleLeg);
```

6. Error Handling

Extension functions must use the C++ exception mechanism to report errors detected during their execution.

Exception Classes

The exceptions an extension function throws must be derived from one of the following exception classes supported by Pricing Engine:

- ErrorException – Use this type of exception when the error detected by an extension function is related to the processing of a specific event (for example, if the subscriber data related to an event does not exist in the data store). On encountering an error exception, Pricing Engine rejects the event (except for predefined error exceptions that Pricing Engine handles).
- LockTimeoutException – Use this exception when an extension function attempts to perform database operations that fail as a result of a timeout.

Pricing Engine interprets other exception types as error exceptions.

Exception Handling for Activities

This section explains how exceptions are handled in the various Rating activities.

Event Processing

In case of a LockTimeoutException, Pricing Engine returns the Retry return code. This makes the envelope start processing again because locking is only temporary.

For any other exception thrown by an extension function, the Reject Event status is returned.

Event and PI Extract

If the Event and PI extract mechanism encounters an exception, the FETCH_FAILURE status is returned from the EntityCursor::fetch method. An error is written to the log file, and the record is disregarded. Calling the fetch method again moves the processing to the next database record.

Usage Queries

For any exception in a usage query, the calling application receives an empty XML file, and a returned status indicating that a problem has occurred.

closePIForCycle

For the Close PI for Cycle activity, LockTimeoutException causes Pricing Engine to return the Retry return code. This indicates to the envelope that the method must be called again because locking is only temporary.

For any other exception thrown by an extension function, the Fatal status is returned, indicating that the process cannot continue.

finalizeCustomerPIs

For the Finalize Customer PI activity, any exception ends the method with a Fatal status, indicating that the process cannot continue.

7. Extension Function Testing Simulator

The extension function Testing Simulator reduces the time and complexity of testing new extension functions or modifying existing ones. It enables testing extension functions in a unit test without dependency on Product Catalog, usage and customer data stores, and with minimal preparation.

To test an extension function without the testing simulator, the developer must declare that function in Product Catalog, insert a call to that function in the correct PIT, release a Product Catalog version, distribute it to the testing environment, and run Rating or the related envelope with the correct event. In many cases, these requirements cannot be fulfilled or take too much time/effort.

Testing simulations described in this chapter enable the developer to write a test program that simulates the initialization, activation, and release of the extension function. The developer first creates all the entities required by the extension function implementation and initializes them, then creates the extension function, sets all the parameters and context entities, and finally activates the extension function.

The testing simulation utility classes enable simple coding of the testing program. They are mostly based on the exiting Pricing Engine interfaces. (These classes are described in the "Testing Simulation Class Definition" section.)

The flow of an extension function testing program comprises the following steps:

1. Initializing Pricing Engine
2. Creating entities and auxiliaries
3. Populating entities
4. Creating and initializing the extension function
5. Running the extension function

Subsequent sections describe these steps in detail.

Initializing Pricing Engine

In the initialization phase, the developer needs to:

- Register an error handler (e.g., logger of error to a text file) so that, when an error occurs during the execution, it can be inspected.
- Initialize Pricing Engine. The assumption is that a reference database with a Product Catalog implementation exists (although it does not have to contain the new extension definition and usage).

For more information about Pricing Engine initialization, see the Pricing Engine APIs *document.*

The following is an example:

```
…
// Register an error handler
lErrorHandler LogFileErrorHandler(“The error log file”);

// Initialize the PE in a full initialization mode
if (!PricingEngine::init(“The PE configuration”))
{
      // The initialization failed, flush errors.
      ERR_FlushErrors

      // abort
      exit(1);
}
…
```

Creating Entities and Auxiliaries

The developer needs to create entities that are accessed from the extension function implementation (e.g., Customer Parameters, Event, etc.). It is assumed that, to simplify the testing, the developer wants to create only the entities required for the extension function execution.

For more information about creating Pricing Engine entities, see the Pricing Engine APIs *document.*

Entities

Entities are created by using the Pricing Engine creation methods.

The following example demonstrates the creation of a GSM Voice type event:

…

```
Entity *pEntity =
PricingEngine::createEntity("Event","GSM voice");

if(!pEntity)
{
  // Report on Event creation error
  GEAR_ERR_Report_2(CREATE_ENTITY_FAILED,
      ("Event","GSM voice")

  // Event creation failed - flush the error stack
      ERR_FlushErrors

      //abort
      exit(1);

}
```

…

Auxiliaries

A reference to an auxiliary object loaded during the Pricing Engine initialization is created by setting up an AuxiliaryObjectType with its identification (type, name). This object can later be passed to the extension function as a parameter.

The following is an example:

…

```
// Create a reference to an auxiliary object of type
// "Period set" and name "Peak of peak"
AuxiliaryObjectType AuxilaryObjectType("Period set",
               "Peak off peak");
```

…

Populating Entities

The testing simulator’s basic requirement is that the entity attributes used by the extension function must be initialized.

The entities have been created in the preceding step. In this phase, the only requirement is to set the entities’ attributes to the required values.

note *For more information about Pricing Engine entities and complex attributes, see the* Pricing Engine APIs *document.*

The following example demonstrates how to populate entities:

```
…
try
{
  //Set a value to a string attribute
  pEntity->init(“Event ID”,”1000”);
  //Set a value to a date
  pEntity->init(“Start time”,”2003-02-17 19:50:00”);
  //Set a value to a numeric attribute
  pEntity->init(“Cycle code”,”11”);
  //Set values to a complex attribute
  IComplexAttribute complexAttr =
      pEntity->get(“Code group”);
  complexAttr.set(CodedComplexElementID(0,0),
      NumericType(12));
  complexAttr.set(CodedComplexElementID(0,1),
      NumericType(13));
}
catch(generalException &ge)
{
  // Report on failure during event initialization
  GEAR_ERR_Report_0(ENTITY_INITIALIZATION_FAILED)
  // Flush the error stack
  ERR_FlushErrors
  //abort
  exit(1);
}
…
```

Creating and Initializing Extension Functions

There are two major types of extension functions: regular extensions (entity type, elementary type, and entity subroutine) and pseudo-attribute extensions. This section describes the creation and initialization of extension functions.

Regular Extensions

A regular extension function is inherited from either the EntityMethodImplBase and EntitySubroutineImplBase class or the ElementaryTypeMethodImplBase class. This inheritance is required so that Pricing Engine can activate the extension function during its processing.

A special base class is provided for testing regular extension functions: ExtensionTestingBase. This class has the methods the extension functions need to access their input parameters and context entities (e.g., Customer Parameters, Event, etc.).

When the extension function code is ready for testing, its inheritance should be changed such that it inherits the ExtensionTestingBase class. At the end of the testing, the inheritance should be changed back.

note

For the ExtensionTestingBase class definition, see the "ExtensionTestingBase" section.

Definition Example

The following example demonstrates the definition of an extension function in the testing simulator. The function is an elementary type extension function that is activated on a date type value, receives a base year as an input. It then calculates the number of years since the given base year and returns it as a numeric value.

```
// Creating and initializing an extension function
// GetYearTest.h
// class GetYearTest: public
// ElementaryTypeMethodImplBase<DateType,NumericType>
class GetYearTest: public ExtensionTestingBase
{
public:
  // DTOR.
  virtual ~GetYearTest (){};
  // Init the extension (Optional).
  virtual void init();
  // Release the extension (Optional).
  virtual void release();
  // Create a new GetYearTest instance and return it.
  virtual IClonableNode* clone()const;
  ///Calculate
  virtual NumericType doEval(DateType
      i_attributeValue) const;
};
```

Usage Example

This example illustrates how to create an extension function, initialize it, and set the extension function input parameters.

```
//Create the extension
GetYearTest myExtension;
//Init the extension (optional operation)
myExtension.init();
//Set base year input parameters
NumericType baseYear(1930);
myExtension.setParameter (“Base year”, baseYear);
```

Context Entities

In extension functions, it is possible to use entities that exist in the Pricing Engine computation context. For example, while qualifying an event, the Event, Customer Offers, and Customer Parameters entities are available in the Pricing Engine computation context. In a PIT handler, the Event, Performance Indicator, Customer Offers, Customer Parameters, variables, and Item Parameters entities are available. If such entities are used by the extension function, they should be set prior to running that extension function.

The context entities can be accessed from the extension function implementation using the RatingContext helper class methods.

For ExtensionTestingBase class definition, see the “ExtensionTestingBase” class methods.

Setting Context Entity – Example

This example sets an entity in the computation context:

```
myExtension.setContextEntity(pEntity);
```

Getting Context Entity – Example

This example gets a reference to the Item Parameters entity:

```
const Entity& itemParameters =
RatingContext::getItemParameters();
```

Service Provider

Extension functions can receive services, such as DBServices. To simulate work with services, the following steps should be performed:

1. Create a service.
2. Create a Testing Service Provider and insert the service into it.

   For TestingServiceProvider class definition, see the “TestServiceProvider” section.

3. Activate the extension function initService() method with the service to initialize the service.
4. Run the extension function.

5. Activate the extension function releaseService() method with the service to release the service.

   There is no need to use Perform Subscription for Service in this case.

The following example shows how to use services:

```
{
…
  try
  {
      //Create a service - DBContext is an example for a
      //service
      IDBContext* usageContext =

      PricingEngine::prepareConnection(acm::common::io::I
      Connection* i_pConnection, USAGE);

      //Create a service provider
      TestServiceProvider lServiceProvider;

      //Insert the service into the service provider
      lServiceProvider.insertService(usageContext);

      //Activate init service (this is optional)
      myExtension.initService(usageContext);
      …
      NumericType returnVal =
          myExtension.doEval(DateType::getCurrentDate(),
                    lServiceProvider);
      …
      //Release the extension info from the service
      myExtension.releaseService(usageContext);
  catch(generalException &ge)
  {
      ERR_Report_0 (EXTENSION_ACTIVATION_FAILED);
      throw ErrorException();
      catch(…)
      {
      ERR_Report_0 (EXTENSION_ACTIVATION_FAILED);
      throw ErrorException();
  }
…
}
```

Pseudo-attribute Extensions

There is no need for special actions to create pseudo-attributes. There is also no need to initialize them.

The example below shows how to create pseudo-attributes:

```
PseudoOfferID myExtension;
```

note *Complex pseudo-attributes such as Segments require event processing, and cannot be tested with the testing simulator.*

Running Extensions

The method for running the extension function depends on the regular extension type (entity type, elementary type, and entity subroutine) or pseudo-attribute extension. The methods for both types are described in subsequent sections.

Regular Extensions

After an extension function is created, and all its parameters and context entities are set, the developer can run that extension function by activating the doEval() method. The developer can add printed logging to facilitate debugging.

At the end of the run, the resources acquired by the extension function must be released.

The example below demonstrates running and releasing an extension function. The extension function in the example returns a numeric value and receives a date value.

```
NumericType returnVal =
  myExtension.doEval (DateType::getCurrentDate());


…


//Release the extension
myExtension.release();
```

Pseudo-attribute Extensions

Pseudo-attributes can have two services: set and get. The developer needs to test both of these services.

After a pseudo-attribute extension function is created, and all its parameters and context entities are set, the developer can run the extension function by activating the doSet() and doGet() methods. The developer can add printed logging to facilitate debugging.

The example below demonstrates retrieving the value of a numeric pseudo-attribute. The extension receives an instance of the entity on which it was defined and a numeric type object for the returned value.

```
NumericType o_offerID;

myExtension.doGet(pEntity, o_offerID);

// Retrieving the value of the pseudo-attribute of
// numeric type
// Set a new value to the pseudo-attribute
NumericType offered(1234);

myExtension.doSet(pEntity, o_offerID);
```

Testing Simulation Class Definition

This section describes how to test simulation class definitions.

ExtensionTestingBase

This class is a stub implementation replacing all extension types' base classes. Its major role is to simulate the access method to the extension function parameters and Pricing Engine computation context entities. It also provides methods to set entities to the extension function computation context.

Subsequent sections describe the methods of this class.

getParameter

The ExtensionTestingBase class implements the exact interface of the getParameter methods implemented by the original extension base class. For more information, see the “Extension Function Input Parameters” chapter.

get[type]Param

The ExtensionTestingBase class implements the exact interface of the get[type]Param group of methods implemented by the original extension base class. For more information, see the “Extension Function Input Parameters” chapter.

getDynamicParameter

The ExtensionTestingBase class implements the exact interface of the qetDynamicParameter group of methods implemented by the original extension base class. For more information, see the “Extension Function Input Parameters” chapter.

get[type]DynamicParameter

The ExtensionTestingBase class implements the exact interface of the get[type]DynamicParameter group of methods implemented by the original extension base class. For more information, see the “Extension Function Input Parameters” chapter.

getDynamicParametersCount

The ExtensionTestingBase class implements the exact interface of the getDynamicParametersCount method implemented by the original extension base class. For more information, see the “Extension Function Input Parameters” chapter.

setParameter

This group of methods is used to set the values of the input parameters for an extension function. The parameter name is a unique value that is used later by the getParameter() method to retrieve a specific parameter value.

The syntax is:

```
void setParameter(const UtString& i_parameterName,
NumericType& i_value);
void setParameter(const UtString& i_parameterName,
BooleanType& i_value);
void setParameter(const UtString& i_parameterName,
DateType& i_value);
void setParameter(const UtString& i_parameterName,
UtString& i_value);
void setParameter(const UtString& i_parameterName,
IComplexAttribute& i_value);
void setParameter(const UtString& i_parameterName,
AuxiliaryObjectType& i_value);
void setParameter(const UtString& i_parameterName,
AuxiliaryValueType& i_value);
```

The parameters are:

Name	Direction	Type	Description
i_parameterName	In	UtString&	The parameter name
i_value	In	Depends on the parameter type	The parameter value

pushDynamicParameter

This group of methods is used to set the values of the dynamic parameters for an extension function. The method pushes the parameter as the next parameter after the previously pushed parameter. The index of the dynamic parameter starts from zero. The index value of a pushed parameter is the value of the previously pushed parameter incremented by one. The dynamic parameters can be retrieved by calling the getDynamicParameter() method.

The syntax is:

```
void pushDynamicParameter(NumericType& i_value);
void pushDynamicParameter(BooleanType& i_value);
void pushDynamicParameter(DateType& i_value);
void pushDynamicParameter(UtString& i_value);
void pushDynamicParameter(AuxiliaryObjectType& i_value);
void pushDynamicParameter(AuxiliaryValueType& i_value);
```

The parameters are:

Name	Direction	Type	Description
i_value	In	The dynamic parameter type	The parameter

setContextEntity

This method is used to set an entity in the computation context. The entities are stored in the Pricing Engine internal storage and can be retrieved later from an extension function.

The syntax is:

```
bool setContextEntity(Entity* i_pEntity);
```

The parameters are:

Name	Direction	Type	Description
i_pEntity	In	Entity*	The entity
	Return value	bool	Success or Fail.

init

The ExtensionTestingBase class implements the exact interface of the init method implemented by the original extension base class. For more information, see the “Implementing Extension Functions” chapter.

release

The ExtensionTestingBase class implements the exact interface of the release method implemented by the original extension base class. For more information, see the “Implementing Extension Functions” chapter.

initService

The ExtensionTestingBase class implements the exact interface of the initService method implemented by the original extension base class. For more information, see the “Implementing Extension Functions” chapter.

releaseService

The ExtensionTestingBase class implements the exact interface of the releaseService method implemented by the original extension base class. For more information, see the "Implementing Extension Functions" chapter.

TestServiceProvider

This class is a stub implementation of IServicesProvider. This class acts as a service provider sent to extension functions.

Subsequent sections describe the methods of this class.

insertService

This method assigns the given pointer to the member pointer.

The syntax is:

```
void insertService(ServiceBase *i_pServiceBase);
```

The parameters are:

Name	Direction	Type	Description
i_pServiceBase	In	ServiceBase *	The service

getService

This method checks if the requested name and type match the service name and type. If yes, it returns the service; if not, it returns Null.

The syntax is:

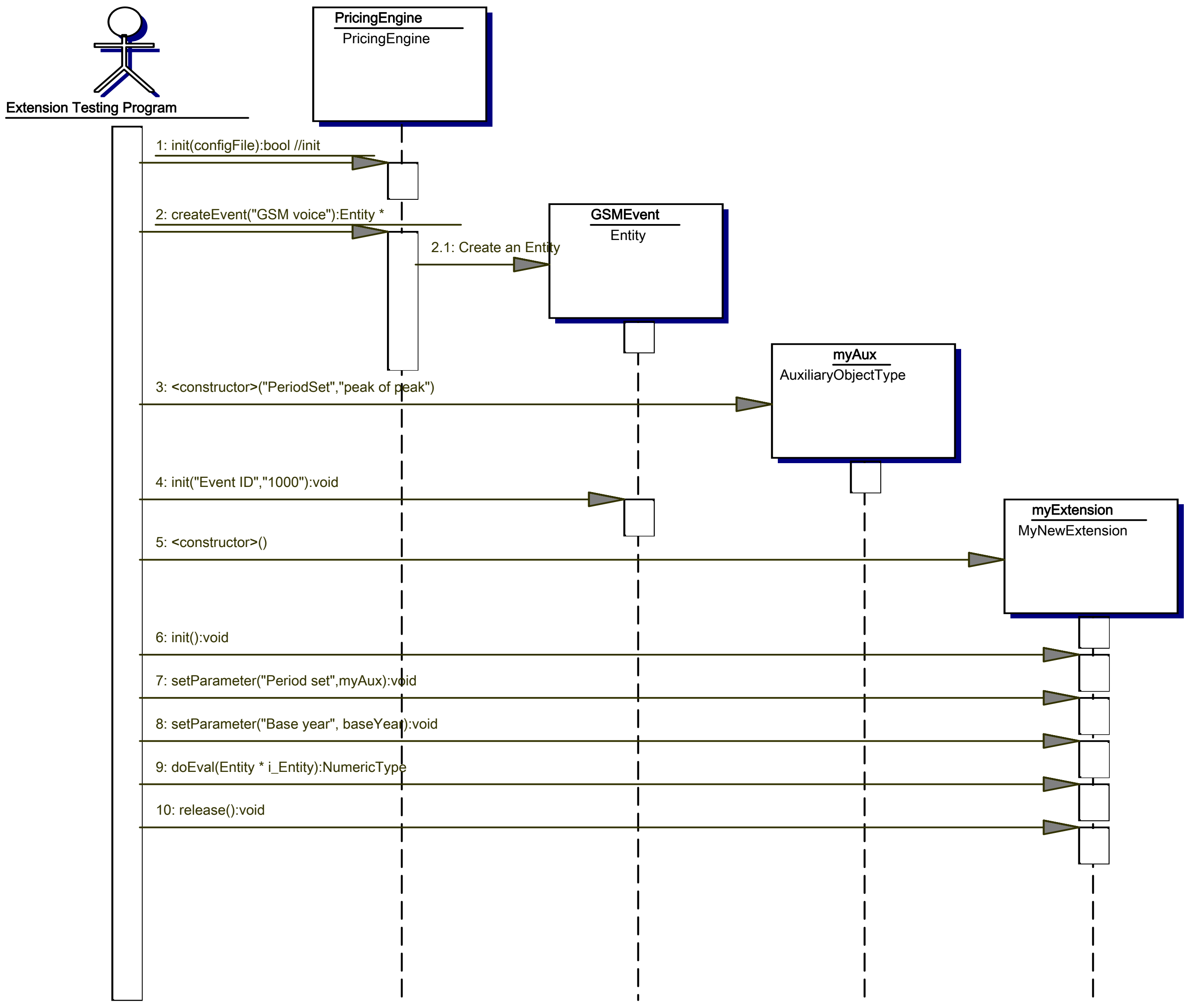
```
ServiceBase* getService(const UtString & i_serviceName,
ServiceType i_serviceType)
```

The parameters are:

Name	Direction	Type	Description
i_serviceName	In	UtString &	The service name
i_serviceType	In	ServiceType	The service type

Testing Flow

The diagram below illustrates the testing flow.


Extension Testing Program
PricingEngine
PricingEngine
GSMEvent
Entity
myAux
AuxiliaryObjectType
myExtension
MyNewExtension
1: init(configFile):bool //init
2: createEvent("GSM voice"):Entity *
2.1: Create an Entity
3: <constructor>("PeriodSet","peak of peak")
4: init("Event ID","1000"):void
5: <constructor>()
6: init():void
7: setParameter("Period set",myAux):void
8: setParameter("Base year", baseYear):void
9: doEval(Entity * i_Entity):NumericType
10: release():void

8. DEPRECATED APIS

This chapter describes the APIs that is deprecated in version 6.0 as well as their substitutes.

The MethodDelegators class provides methods that access context entities. All these methods are available with the RatingContext helper class. Therefore, these methods are deprecated in version 6.0 of Pricing Engine.

The following is an example of the code change that is required as a result of the interface deprecation. This example is for the Item Parameters entity, but it holds true for all other entities (such as Customer Offers, Event, etc.).

- Current usage:

  ```
  const Entity& itemParameters = getItemParameters();
  ```

- New usage:

  ```
  const Entity& itemParameters =
  RatingContext::getItemParameters();
  ```

Appendix A. Extension Functions – Quick Reference

Following are general instructions for creating an extension function:

1. Design the extension function. Define the extension type, and input and output parameters. According to the business logic, decide to which extension library the function belongs.
2. Write the extension class. Check if the function should use services, and implement the relevant method (Init(), doEval(), release(), etc.).

   note *Make sure to name the class in the extension name and set the parameters' name strings as they are written in Implementation Repository.*

3. Register the extension function class in the init.cpp of the relevant extension library.

   note *For description of pseudo-attribute extension functions, see the "Pseudo-attribute Extension Functions" section.*

Following is the skeleton of the Init.cpp file:

`Init.cpp`

```
#include <ExtensionManager.h>
#include <ExtensionFunctionHeader.h>
DEFINE_EXTENSION_LIBRARY();
       all the registrations of extension functions
END_EXTENSION_LIBRARY();
```

Appendix B. Extension Library Definition

Extension functions belong to extension libraries. Each extension library contains the extension functions grouped according to specific application criteria.

The extension libraries are loaded during Pricing Engine initialization, and are specified in the Pricing Engine configuration file under the <Extension> tag. This tag maps a logical extension name (Name attribute) to the physical location of the extension library (Path attribute).

Following is an example of declaring extension libraries in the Pricing Engine configuration file:

```
<Extensions>
  <Extension
  path="/iphome/ip/ccip/ccip/proj/grt200/bin/libutilsext.
  so" name="Utility core extensions"/>
  <Extension
  path="/iphome/ip/ccip/ccip/proj/grt200/bin/libpriceext.
  so" name="Pricing core extensions"/>
  <Extension
  path="/iphome/ip/ccip/ccip/proj/grt200/bin/libpricectl.
  so" name="Pricing control extensions"/>
</Extensions>
```

The mandatory attributes for the Extension tag are:

Attribute Name	Attribute Description
Path	Shared library's full path and name
Name	Shared library's logical name

The library naming convention is "<Meaningful Name> extensions". The project containing the library's function is named as follows:

- Windows – <Meaningful Name>Extensions
- UNIX – <Name>ext

For example, a shared library containing all core extension functions that are not event-specific could be named as follows:

- Windows – Utility Core Extensions
- UNIX – utilsext

To populate the <Extension> tag, the 'PM_EXTENSION_LIBRARY_FILE' environment variable should be initialized during the Rater run to be the name of a file adding the extension library to the Pricing Engine configuration file. This initialization should be made in a script called 'op_pm#_env_sh', and the file responsible for adding the extension library itself should be called pm#customized, where # is the number of the customization layer.

For example:

op_pm9_env_sh:

export PM_EXTENSION_LIBRARY_FILE=customized

pm9customized:

<Extension path="`#GL libcpm9priceext.$SHARED_LIB_EXT`" name="Implementation reference tables extension" />