ORACLE
INSURANCE

Oracle® Insurance Policy Administration

Extensibility

Version 9
E16287_01
December 2009

Contents

1. Introduction

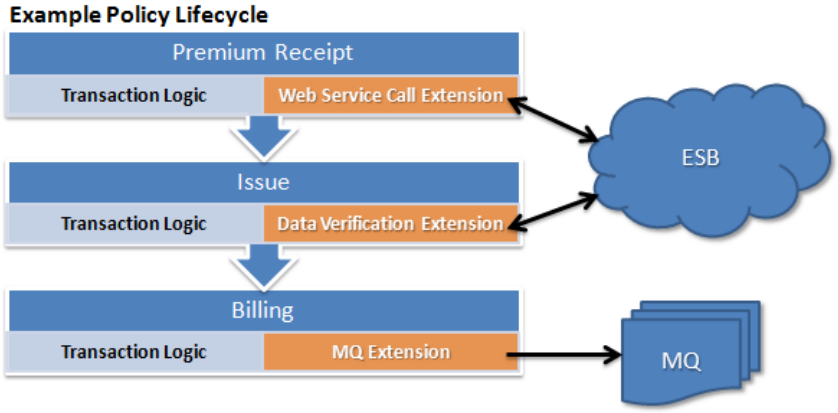
Extensibility is a critical factor when selecting an architecture that can be enhanced with the speed of your organizations' needs. The ability to extend the system and hook in new system capability without making major infrastructure changes is necessary for system maintainability and in avoiding early obsolescence. The Oracle Insurance Policy Administration system (OIPA) provides several mechanisms for extensibility. Currently, all extensions are implemented as Java classes that are injected into specific points or levels in the OIPA infrastructure. Extension developers need only implement the requisite Java interfaces in order to access this powerful OIPA feature.

Primarily, there are two levels of extensions which are called the transaction level and the system level.

1. The **transaction level** extensions allow for custom logic within the context of a policy's lifecycle.
2. The **system level** extensions allow for custom logic through the lifecycle of specific system events. In general, system level extensions allow for finer-grained customization.

Transaction Level Extensions

The key benefit of transaction level extensions is to allow for greater control over a policy's lifecycle. Since transaction level extensions are provided with data from running transactions, they are powerful tools for integration. Transaction level processing is the logic that executes when an activity or event is ran against a policy. In the example below, a policy lifecycle includes the OIPA transactions of Premium Receipt, Issue and Billing respectively. The first two transactions illustrate how the system can perform messaging over an enterprise service bus (ESB). The last transaction, Billing, illustrates MQ series integration.



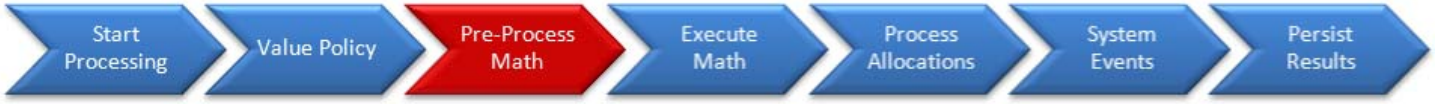
System Level Extensions

When finer-grained control over the application's lifecycle is required, system level extensions can be employed. System level extensions are provided through the Extensibility Framework which is discussed later in this document.

Below is a simplified rules engine processing example that illustrates how system level extensions can provide pre- or post-processing or replace a processing step altogether.

1) Default implementation

2) Pre-processing added via an extension



3) Post-processing added via an extension

4) Math processing lifecycle step is replaced with custom math via an extension

2. Transaction Processing Extensions

There are several facilities in place that enable extensibility within transaction processing. They are in the Math processing of a transaction rule, via the ExternalProcess business rule and through the FileReceived web service. Each mechanism for extending OIPA transaction processing is explained in detail below.

2.1 Math Extensions

Some tasks may require interaction with custom Java code during the math processing of rules. OIPA has a dedicated math variable type which can be added to any XML math section that contains the `<MathVariable>` elements. The math variable type is PROCESS, see the OIPA XML Syntax guide for further detail. This has the advantage of allowing for rule logic to dictate which extension should execute and which parameters should be passed.

Example XML Configuration
This example illustrates the PROCESS type <MathVariable> and the available attributes and parameter passing.

```
<MathVariable NAME="VariableName"
              TYPE="PROCESS"
              NAMESPACE="com.example.package"
              OBJECT="ClassName"
              ISARRAY="YES|NO"
              DATATYPE*">
   <Parameters>
      <Parameter NAME="InputVar" TYPE="INPUT">Literal|VariableName</Parameter>
      <Parameter NAME="OutputVar" TYPE="OUTPUT">VariableName</Parameter>
   </Parameters>
</MathVariable>
```

***Data type can be one of: TEXT, INTEGER, DECIMAL, BOOLEAN, DATE*

Java Implementation Details

The above example requires that a class, com.example.package.ClassName, implements the interface IProcessableObject. This class must be present on the class path of the application's class loader. The IProcessableObject interface is defined as follows:

```
public interface IProcessableObject {

   public Object execute( Map<String, Object> inputParameterMap,
                          Map<String, Object> outputParameterMap )
                          throws Exception;
}
```

The implementation will receive two `java.util.Map` instances. The first, `inputParameterMap`, contains all of the values defined as input parameters in the Math XML, keyed on the parameter name. Similarly, the `outputParamaterMap` contains all of the values defined as output parameters in the Math XML, keyed on the parameter name. Output parameters will be copied back to the original variable name specified as the value of the parameter. In other words, the *value* of the output variables will be passed into the extension. Any changes to that value will be applied to the variable that is provided by the parameter definition. In essence, the variable is "passed by reference" to the extension.

By supporting output parameters, the extension can return any number values. By default the extension returns a single value, which is stored in the variable given by the VARIABLENAME attribute from the example above.

2.1.1 Example Use of Math Extension

Assume there is some service that is available in the enterprise that will validate that a postal code, provided by the user, matches the city provided by the user. The extension will return a Boolean indicating whether or not the postal code/city pair are a match. It will also return the valid city name for the given postal code in the event that the city and postal code do not match.

Example XML Configuration

```
<MathVariable NAME="PostalCodeMV"
              TYPE="FIELD" DATATYPE="TEXT">Policy:IssuePostalCode</MathVariable>
<MathVariable NAME="CityMV"
              TYPE="FIELD" DATATYPE="TEXT">Policy:IssueCity</MathVariable>
<MathVariable NAME="CorrectCityMV"
              TYPE="VALUE" DATATYPE="TEXT"></MathVariable>
<MathVariable NAME="PostalCodeAndCityMatch"
              TYPE="PROCESS"
              NAMESPACE="com.example.package"
              OBJECT="ValidatePostalCode"
              ISARRAY="NO"
              DATATYPE="BOOLEAN">
   <Parameters>
      <Parameter NAME="PostalCode" TYPE="INPUT">PostalCodeMV</Parameter>
      <Parameter NAME="City" TYPE="INPUT">CityMV</Parameter>
      <Parameter NAME="CorrectedCity" TYPE="OUTPUT">CorrectCityMV</Parameter>
   </Parameters>
</MathVariable>
```

Extension Pseudo-Code

```
class VerifyPostalCode implements IProcessableObject {

   public Object execute( Map <String, Object> inputVariableMap,
                          Map <String, Object> outputVariableMap ) throws Exception {

      String postalCode = ( String )inputVariableMap.get( "PostalCode" );
      String city = ( String )inputVariableMap.get( "City" );
      VerificationResult result = ExternalService.verifyPostalCode( postalCode, city );

      if( result.isValid() ) {
         return true;
      }
      else {
         outputVariableMap.put( "CorrectedCity", result.getCorrectedCity() );
         return false;
      }
   }

}
```

2.2 ExternalProcess Business Rule

Every transaction in OIPA can have business rules associated with it. These business rules are responsible for applying logic at the end of the transaction lifecycle. Typical business rules will perform actions at a higher level than the transaction itself, such as generating a suspense account or writing data to fields (i.e. policy, role, client, etc.) Business rules are the only mechanism that allows for data persistence within the transaction's unit of work.

In order to facilitate greater flexibility, OPA includes a special business rule that is capable of calling custom Java code called `ExternalProcess`.

Implementing `ExternalProcess` requires the following:

1. A business rule named `ExternalProcess` attached to the transaction.
    a. The business rule will describe the Java class to be executed.
2. The business rule name must exist in the Transaction-Business-Rule-Packet business rule in the order it should be executed compared to other attached business rules.
3. An extension class that implements the `IApeExtension` interface

Example XML Configuration

```
<ExternalProcess>
   <Process>
      <Assembly>com.example.extension</Assembly>
      <Object>ExternalProcessImpl</Object>
   </Process>
   <Parameters>
      <Parameter NAME="ParameterName">ParameterValue</Parameter>
   </Parameters>
</ExternalProcess>
```

Java Implementation Details

The above example requires that a class, `com.example.extension.ExternalProcessImpl`, implements the interface `IApeExtension`. This class must be present on the class path of the application's class loader. The `IApeExtension` interface is defined as follows:

```
public interface IApeExtension {

   public void process( IActivityBll activityBll, Map <String, String> parameterCollection );

   public void processUndo( IActivityBll activityBll, Map <String, String> parameterCollection );
}
```

The `process(..)` method is executed during forward processing. The `processUndo(..)` method is executed during reversal.

The `activityBll` parameter exposes the necessary surface area for extension. It's critical to understand that activities are executed as a single unit of work. That is, if any operation fails, then no changes will have been made to the system. To enable this, insert, change and delete operations are exposed by `IActivityBll`. All data changes should be made through these mechanisms as they ensure proper transactional integrity. *Note: direct modification of the database can lead to undesirable~~,~~ or inaccurate~~,~~ results*.

2.3 FileReceived Web Service

The FileReceived web service, sometimes referred to as “AsFile”, exposes extensions before and after the insert operation occurs. Illustrated below is the FileReceived lifecycle and the associated insert extension opportunities.

The basic lifecycle for the FileReceived web service is:

1. Web service request received
2. The appropriate AsFile entry is retrieved based on the given FileID
3. The AssignAttributes XML is processed
4. The XSLT associated with the AsFile entry maps the request XML to AsXml
5. If the AsXml contains Validation Errors, AsFile stops and a SOAP fault is returned to the caller
6. AsFile maps the AsXml to business objects
7. PreInsert extensions are executed against the objects
8. The objects are inserted into the database
9. PostInsert extensions are executed against the objects
10. The AsXml is returned to the caller

Example XML Configuration

```
<File>
	<AssignAttributes>
	...
	</AssignAttributes>
	<PreInsert>
		<Object CLASS="com.example.extension.PreInsertExtension">
			<Parameters>
				<Parameter NAME=”Name”>Value</Parameter>
				<Parameter NAME=”Name”>Value</Parameter>
				...
			</Parameters>
		</Object>
		<Object CLASS="com.example.extension.PreInsertExtension2" />
		...
	</PreInsert>
	<PostInsert>
		<Object CLASS="com.example.extension.PostInsertExtension" />
		<Object CLASS="com.example.extension.PostInsertExtension2" />
		...
	</PostInsert>
</File>
```

Java Implementation Details

The `PreInsert` and `PostInsert` extensions must implement the `IFilePreInsertProcessorBll` and `IFilePostInsertProcessorBll` respectively. If the `Parameters` element is present, the specified parameters will be passed to the extension. The text of the `Parameter` elements should contain the name of an attribute from the `AssignAttributes` section. This allows for the passage of data to the defined extensions.

The `PreInsert` and `PostInsert` interfaces are defined as:

```
public interface IFilePostInsertProcessorBll {

   public <T extends AdminServerPersistentDcl> String process( ArrayList<T> dclList, String requestXml, Map
<String, String> parameterMap );
}
```

```
public interface IFilePostInsertProcessorBll {

   public <T extends AdminServerPersistentDcl> String process( ArrayList <T> dclList, String requestXml, Map
<String, String> parameterMap );
}
```

3. System Level Extensions (Extensibility Framework)

The Extensibility Framework provides a mechanism by which system event lifecycles can be extended. Custom Java code can be added before or after a lifecycle step and it can also replace the lifecycle step altogether. The Extensibility Framework is present at the transaction processing, web service and user interface levels.

The Extensibility Framework maps custom Java classes to named points in the system. These names vary based on the context and are described in detail later. A single Java class can be mapped to multiple extension points and, conversely, a single extension point can have multiple Java classes mapped to it.

The Extensibility Framework allows for wildcards in the specification of the extension point name. For instance, `Activity.*` will call a custom Java class for every extension point starting with `Activity.*`. If "`*`" is used by itself, then all extension points will be processed by the Java class.
Wildcard options are as follows:

`<Qualifier>.<Name>`	`Activity.StartProcessing`	Only the StartProcessing event will be intercepted
`<Qualifier>.*`	`Activity.*`	All extension points starting with Activity will be intercepted
*	*	All extension points will be intercepted

Lifecycle flow control is managed through the two methods exposed by every extension point interface, `processPre(..)` and `processPost(..)`. The `processPre(..)` method is executed prior to the lifecycle event. This method always returns a Boolean where true indicates that the lifecycle step should execute and false indicates that it should be skipped. This allows for lifecycle steps to be overridden by custom code. The `processPost(..)` method is called after the lifecycle step has completed. If that lifecycle step generated any data, it will be present in the context map as "`Result`".

It's important to note that extensions are not thread-safe. That is, for performance, a single instance is created and continually executed. The use of member variables in an extension is discouraged unless proper locking is in place.

All extension points contain context data in the form of a `java.util.Map`. The contents of this map will vary depending on the extension point. This map is shared between the `processPre(..)` and `processPost(..)` methods, but it is not shared with other extension points. Inter-extension communication can be achieved through the use of some context data mechanism, depending on the implementation. This could be in the form of a `ThreadLocal` variable, or an HTTP/Servlet request context, for example.

Each extension point can have one or more extension registered to it. These extensions are executed in the order in which they appear in the XML configuration.

Example XML Configuration

```
<extensions>
	<extensionPoint type="com.example.extension.ExtensionPointClassName"
			extensionPointName="ExtensionPoint.Name">
		<register extension="com.example.extension.ExtensionClassName1" />
		<register extension="com.example.extension.ExtensionClassName2" />
	</extensionPoint>
</extensions>
```

3.1 Shared Rules Engine

The Shared Rules Engine (SRE) can be extended through the use of the extensibility framework. The SRE powers activity processing and can be extended at every critical lifecycle point.

Example XML Configuration

```
<extensionPoint type="com.adminserver.sre.extensibility.SreExtensionPoint"
       extensionPointName="Activity.*">
       <register extension="com.example.SreExtension" />
</extensionPoint>
```

All extensions written for the `SreExtensionPoint` must implement `ISreExtension`.

```
public interface ISreExtension {

   public boolean processPre( SreExtensionContext extensionContext );

   public void processPost( SreExtensionContext extensionContext );
}
```

3.1.1 SRE Extension Points

Forward Processing

Lifecycle Step	Extension Point	Available Variables
ActivityBll.Process	Activity.InitializeProcessing	ClientNumber, ActivityProcessType
ActivityProcessorBll.Process	Activity.StartProcessing	InputVariableMap, ApplicationCallback, ActivityProcessType
DoSuspense	Activity.StartSuspense	Activity, Transaction
ProcessSuspense	Activity.ProcessSuspense	Activity
ProcessMultiSuspense	Activity.ProcessMultiSuspense	Activity
DoValuation	Activity.StartValuation	Activity, Transaction
PolicyValuation.Value	Activity.ValuePolicy	ValuationInformation
DoBeginPointInTimeValuation	Activity.StartPITValuation	Activity, PointInTimeValuationProcess
DoMath	Activity.ProcessMath	Activity
DoBusinessLogic	Activity.StartBusinessLogic	Activity
Rule.ProcessRule	Activity.ProcessBusinessRule	BusinessRule, RuleOption, Activity
DoAssignment	Activity.StartAssignment	Activity, Transaction
ProcessAssignments	Activity.ProcessAssignments	AssignmentList, Activity, ExpressionValidator
ProcessAssignment	Activity.ProcessAssignment	Assignment, Activity
DoDisbursement	Activity.StartDisbursement	Activity, Transaction
ProcessDisbursements	Activity.ProcessDisbursements	DisbursementDetails, Activity
ProcessBalanced	Activity.ProcessBalancedDisbursements	DisbursementDetails, Activity, DisbursementData
ProcessUnbalanced	Activity.ProcessUnBalancedDisbursements	DisbursementDetails, Activity, DisbursementData
DoAccounting	Activity.StartAccounting	Activity
ProcessAccounting	Activity.ProcessAccounting	Activity
DoSpawn	Activity.ProcessSpawn	Activity, Transaction
DoEndPointInTimeValuation	Activity.CompletePITValuation	Activity, PointInTimeValuationProcess, ActivityStatus
DoWrite	Activity.Persist	ActivityProcessResult
DoWriteOnSystemError	Activity.SystemError	Exception

Reverse Processing

Lifecycle Step	Extension Point	Available Variables
ActivityBll.Process	Activity.InitializeProcessing	ClientNumber, ActivityProcessType
UndoProcessor.Process	*Activity.StartUndoProcessing*	*InputVariableMap, ApplicationCallback*
DoBusinessLogicForNuvPending	*Activity.StartNuvPendingBusinessLogic*	*Activity, Transactin*
Rule.ProcessNuvPending	*Activity.ProcessBusinessRuleNuvPending*	*Activity, BusinessRule*
ProcessValuationForUndo	*Activity.StartValuation*	*Activity, Transaction*
DoAccounting	Activity.StartAccounting	Activity, Transaction
ProcessAccounting	Activity.ProcessAccounting	Activity
DoBusinessLogicForUndo	*Activity.StartBusinessLogic*	*Activity, Transaction*
Rule.ProcessUndo	*Activity.ProcessBusinessRuleUndo*	*ActivityBll, BusinessRuleBll*
DoWrite	Activity.Persist	ActivityProcessResult
DoWriteOnSystemError	Activity.SystemError	Exception

3.2 Web Services

Web services can be extended through the use of the extensibility framework.

Example XML Configuration

```
<extensionPoint type="com.adminserver.webservice.extensibility.WebserviceExtensionPoint"
       extensionPointName="SecuredWebService.*">
       <register extension="com.example.WebServiceExtension" />
</extensionPoint>
```

All extensions written for the `WebserviceExtensionPoint` must implement `IWebserviceExtension`

```
public interface IWebserviceExtension {

   public boolean processPre( WebserviceExtensionContext extensionContext );

   public void processPost( WebserviceExtensionContext extensionContext );
}
```

The general `WebserviceExtensionPoint` has a single extension point name, `SecureWebService.PerformAuthorization`. This extension point allows for the overriding of the web service security mechanism.

This extension will receive the following parameters in the context map:

User	The username passed to the web service for authentication
Password	The password passed to the web service for authentication
WebMethod	The web method being called
WebServiceName	The web service containing the web method being called
Parameters	The parameters passed to the web service

In the `processPost` method, the context map will contain `AuthorizationResult`, which contains the result of the authorization request. This object can be altered to adjust the behavior of the authentication.

3.2.1 File Received

The FileReceived web service can be extended through the use of the extensibility framework.

Example XML Configuration

```
<extensionPoint type="com.adminserver.webservice.extensibility.FileReceivedExtensionPoint"
       extensionPointName="FileReceived.*">
       <register extension="com.example.FileReceivedExtension" />
</extensionPoint>
```

All extensions written for the `FileReceivedExtensionPoint` must implement `IFileReceivedExtension`

```
public interface IFileReceivedExtension {

   public boolean processPre( FileReceivedExtensionContext extensionContext );

   public void processPost(FileReceivedExtensionContext extensionContext );
}
```

3.2.2 File Received Extension Points

Lifecycle Step	Extension Point	Available Variables
FileReceived.ProcessFileReceived	FileReceived.StartProcessingFileReceived	FileId, IncomingXml
GetFileProcessDcl	FileReceived.StartRetrievingFileRecord	FileId
FindByFileId	FileReceived.FindRecord	FileProcessData
CreateFileProcessDcl	FileReceived.CreateDataCarrier	XmlHelperUtility, FileProcessData
ProcessAssignAttributes	FileReceived.ProcessAssignAttributes	FileProcessData
ProcessRequest	FileReceived.StartProcessingRequest	FileProcessData
TransformToAsXml	FileReceived.TransformToXml	FileProcessData
ValidateAsXml	FileReceived.ValidateXml	XmlDocument
MapXmlToObject	FileReceived.Deserialize	FileProcessData
ProcessImportedObject	FileReceived.StartProcessingDataCarriers	FileProcessData,PendingImportedObject, XmlDocument
PerformPreInsert	FileReceived.StartPreInsert	FileProcessData, PendingInsertObjects
RetrieveDclList	FileReceived.StartRetrievingDataCarriers	PendingImportedObject
BuildDclListFromAsXml	FileReceived.BuildDataCarriersList	PendingImportedObject
DoPreInsert	FileReceived.PreInsert	FileProcessData, PendingInsertObjects
PerformInsert	FileReceived.InsertData	PendingInsertObjects
PerformPostInsert	FileReceived.StartPostInsert	FileProcessData, PendingInsertObjects, XmlDocument
DoPostInsertProcessing	FileReceived.StartPostInsertProcessing	FileProcessData, PendingInsertObjects, XmlDocument
DoPostInsert	FileReceived.PostInsert	FileProcessData, PendingInsertObjects
BuildResultString	FileReceived.StartBuildingResult	FileProcessData
LoadOutputXslt	FileReceived.LoadOutputXslt	FileProcessData

3.3 User Interface

The extension point names for the user interface are convention based. They take the form of `<PageName>.<StartProcessing|Process[Action]>`. An example would be `Policy.ProcessSave`. All of the extension point names can be discovered by writing a “catch-all” extension point and logging the extension point names.

All extension points will receive the current form as an input parameter which can be retrieved with the `getCurrentForm()` method in the `UipExtensionContext`.

Example XML Configuration

```
<extensionPoint type="com.adminserver.pas.uip.extensibility.UipExtensionPoint"
       extensionPointName="*">
       <register extension="com.example.UipExtension" />
</extensionPoint>
```

All extensions written for the `UipExtensionPoint` must implement `IUipExtension`

```
public interface IUipExtension {

   public boolean processPre( UipExtensionContext extensionContext );

   public void processPost( UipExtensionContext extensionContext );
}
```