

Oracle® Insurance Policy Administration

Extensibility

Version 9.4.0.0

Documentation Part Number: E18894-01

June 2011

Copyright © 2009, 2011, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are “commercial computer software” or “commercial technical data” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Where an Oracle offering includes third party content or software, we may be required to include related notices. For information on third party notices and the software and related documentation in connection with which they need to be included, please contact the attorney from the Development and Strategic Initiatives Legal Group that supports the development team for the Oracle offering. Contact information can be found on the Attorney Contact Chart.

The information contained in this document is for informational sharing purposes only and should be considered in your capacity as a customer advisory board member or pursuant to your beta trial agreement only. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle Software License and Service Agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

Table of Contents

Overview	4
Transaction Level Extensions.....	4
System Level Extensions	5
Transaction Processing Extensions	6
Math Extensions	6
ExternalProcess Business Rule	9
File Received Web Service	11
System Level Extensions (Excessibility Framework)	13
Shared Rules Engine.....	14
SRE Extension Points.....	15
Web Services.....	17
File Received	18
User Interface	19

Overview

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 (OIPA) system 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: transaction level and system level.

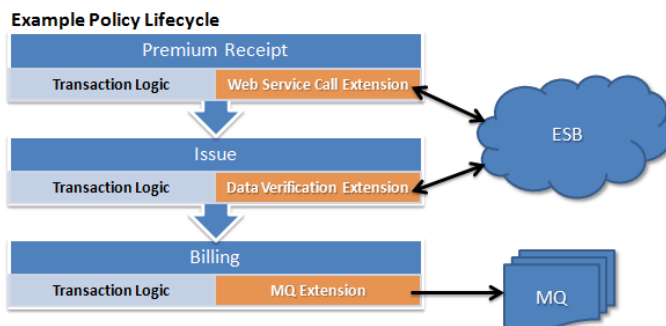
1. The **transaction level** extensions allow for custom logic within the context of a policy's lifecycle.
2. The **system level** extensions allow for custom logic through the lifecycle of specific system events. In general, system level extensions allow for fine-tuned customization.



Transaction Level and System Level Extension Diagram

Transaction Level Extensions

The key benefit of transaction level extensions is that they 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 run 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.



Transaction Level Extension Example

System Level Extensions

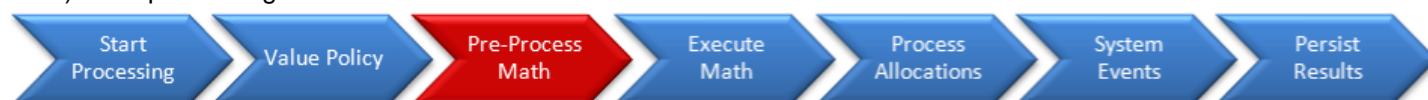
When fine-tuned control over the application's lifecycle is required, system level extensions can be employed. System level extensions are provided through the Extensibility Framework, which is discussed later in this document.

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.

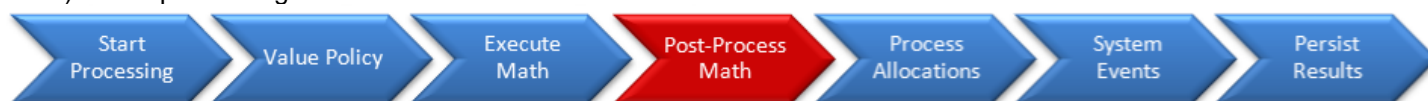
1) Default implementation



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



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.

Math Extensions

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

Example

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="DataType*">
  <Parameters>
    <Parameter NAME="InputVar"
              TYPE="INPUT">Literal|VariableName</Parameter>
    <Parameter NAME="OutputVar" TYPE="OUTPUT">VariableName</Parameter>
  </Parameters>
</MathVariable>
```

Note: 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`, implement 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 `outputParameterMap` 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.

Example Use of Math Extension

Assume there are some services that are 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

```
<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;  
        }  
    }  
}
```


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 allow for data persistence within the transaction's unit of work.

In order to facilitate greater flexibility, OIPA 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.
2. The business rule will describe the Java class to be executed.
3. The business rule name must exist in the `TransactionBusinessRulePacket` business rule in the order it should be executed compared to other attached business rules.
4. An extension class that implements the `IApeExtension` interface.

Example

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

Java Implementation Details

The previous example requires that a class, `com.example.extension.ExternalProcessImpl`, implement 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 transaction integrity.

Note: Direct modification of the database can lead to undesirable or inaccurate results.

File Received Web Service

The FileReceived Web Service, sometimes referred to as AsFile, exposes extensions before and after the insert operation occurs. The FileReceived lifecycle is illustrated below along with the associated insert extension opportunities.

The basic lifecycle for the FileReceived Web Service is:

1. FileReceived Web Service receives a request via a SOAP message.
2. AsFile entry is looked up using the FileID specified in the request.
3. Math in the AsFile entry's XMLData is processed.
4. AssignAttributes in the AsFile entry's XMLData are processed.
5. The XSLT maps the request XML to AsXml.
6. The transformed AsXml is mapped to data objects.
7. PreInsert operations are performed on the objects.
8. Objects are inserted into the database.
9. PostInsert operations are performed on the objects.
10. Load output XSLT from AsFileOutput based on assign attributes.
11. Build response Xml.
12. If the ValidationErrors section is configured in the XSLT then a SOAP fault is created (with embedded response XML) and sent to the caller. Otherwise, response AsXml is returned.

Example

```
<File>
  <RequestType> ... </RequestType>
  <Math ID="MathVariablePrefix">
    <MathVariables>
      <MathVariable> ... </MathVariable>
    </MathVariables>
  </Math>
  <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 either a constant or the name of an attribute from the `AssignAttributes` section. This allows for the passage of data to the defined extensions.

The `PreInsert` interface is defined as:

```
public interface IFilePreInsertProcessorBll {  
  
    public <T extends AdminServerPersistentDcl> ArrayList <T> process( ArrayList <T>  
dclList, String requestXml, Map <String, String> parameterMap ) throws AsExceptionUtl;  
}
```

The `PostInsert` interface is defined as:

```
public interface IFilePostInsertProcessorBll {  
  
    public <T extends AdminServerPersistentDcl> String process( ArrayList <T> dclList,  
String requestXml, Map <String, String> parameterMap ) throws AsExceptionUtl;  
}
```

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

<code><Qualifier>.<Name></code>	<code>Activity.StartProcessing</code>	Only the <code>StartProcessing</code> event will be intercepted
<code><Qualifier>.*</code>	<code>Activity.*</code>	All extension points starting with <code>Activity</code> will be intercepted
<code>*</code>	<code>*</code>	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 means, 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.

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

Example

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

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

```
<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 );

}
```

SRE Extension Points

Forward Processing

Lifecycle Step	Extension Point	Available Variables
ActivityBII.Process	Activity.InitializeProcessing	ClientNumber, ActivityProcessType
ActivityProcessorBII.Process		InputVariableMap, ApplicationCallback,
	Activity.StartProcessing	ActivityProcessType
DoSuspend	Activity.StartSuspend	Activity, Transaction
ProcessSuspend	Activity.ProcessSuspend	Activity
ProcessMultiSuspend	Activity.ProcessMultiSuspend	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		AssignmentList, Activity,
	Activity.ProcessAssignments	ExpressionValidator
ProcessAssignment	Activity.ProcessAssignment	Assignment, Activity
DoDisbursement	Activity.StartDisbursement	Activity, Transaction
ProcessDisbursements	Activity.ProcessDisbursements	DisbursementDetails, Activity
ProcessBalanced		DisbursementDetails, Activity,
	Activity.ProcessBalancedDisbursements	DisbursementData
ProcessUnbalanced		DisbursementDetails, Activity,
	Activity.ProcessUnBalancedDisbursements	DisbursementData
DoAccounting	Activity.StartAccounting	Activity
ProcessAccounting	Activity.ProcessAccounting	Activity
DoSpawn	Activity.ProcessSpawn	Activity, Transaction
DoEndPointInTimeValuation		Activity, PointInTimeValuationProcess,
	Activity.CompletePITValuation	ActivityStatus
PrepareActivityChanges	Activity.PrepareActivityChanges	Activity
DoWrite	Activity.Persist	ActivityProcessResult
DoWriteOnSystemError	Activity.SystemError	Exception

Reverse Processing

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

Web Services

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

Example

```
<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.

File Received

The FileReceived Web Service can be extended through the use of the extensibility framework.

Example

```
<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 );

}
```

File Received Extension Points

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

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

```
<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 );
}
```