Unified Correlation Analyzer
Value Pack Development Guide

Version 3.4
Edition: 1.0

Hewlett Packard
Enterprise

Notices

Contents

List of tables

List of figures

Preface

About this guide

This guide provides an overview of the Unified Correlated Analyzer for Event Based Correlation product and describes how to create Value Packs to target customer specific use cases.

Product Name: Unified Correlation Analyzer for Event Based Correlation (also referred in this document as UCA for EBC)

Product Version: 3.4

Kit Version: 3.4

Audience

Here are some recommendations based on possible reader profiles:

- Solution Developers and integrators
- Software Development Engineers

Software Versions

The term UNIX is used as a generic reference to the operating system, unless otherwise specified.

The software versions referred to in this document are as follows:

Table 1 - Software versions

Product Version	Supported Operating systems
UCA for Event Based Correlation Server Version 3.4	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server release 6.x & 7.x
UCA for Event Based Channel Adapter 3.4	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server release 6.x & 7.x
UCA for Event Based Correlation Software Development Kit Version 3.4	• Windows XP / Vista 64 bits • Windows Server 2012 • Windows 7 64 bits

Typographical Conventions

`Courier` Font:

- Source code and examples of file contents.
- Commands that you enter on the screen.
- Pathnames
- Keyboard key names

Italic Text:

- Filenames, programs and parameters.
- The names of other documents referenced in this manual.

Bold Text:

- To introduce new terms and to emphasize important words.

Associated Documents

The following documents contain useful reference information:

[R1] Unified Correlation Analyzer for Event Based Correlation - Installation Guide

[R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide

[R3] Unified Correlation Analyzer for Event Based Correlation – Administration, Configuration and Troubleshooting Guide

[R4] Unified Correlation Analyzer for Event Based Correlation – Value Pack Examples

[R5] OSS Open Mediation V7.2 - Reference Guide

[R6] OSS Open Mediation V7.2 – Installation and Configuration Guide

[R7] Unified Correlation Analyzer for Event Based Correlation – User Interface Guide

[R8] Unified Correlation Analyzer for Event Based Correlation – Topology Extension

[R9] Unified Correlation Analyzer for Event Based Correlation IM – User Guide

Support

Please visit our HPE Software Support Online Web site at https://softwaresupport.hpe.com/ for contact information, and details about HP Enterprise Software products, services, and support.

The Software support area of the Software Web site includes the following:

- Downloadable documentation.

- Troubleshooting information.
- Patches and updates.
- Problem reporting.
- Training information.
- Support program information.

Chapter 1
Introduction

This guide explains how to create a new correlation project, how to package it and deploy it on a Unified Correlated Analyzer for Event Based Correlation (UCA for EBC) Server in just a few minutes.

After validating some pre-requisites and installing both UCA for EBC (runtime) and UCA for EBC Development Kit products, the following chapters will dive into the development of UCA for EBC Value Packs and explain how to create new scenarios, how to develop alarm/event correlation rules based on samples and how to customize UCA for EBC.

Throughout this document, we use the ${UCA_EBC_HOME} environment variable to reference the root directory ("static" part) of UCA for EBC. The default value for the ${UCA_EBC_HOME} environment variable is /opt/UCA-EBC. The ${UCA_EBC_HOME} environment variable thus references the /opt/UCA-EBC directory unless UCA for EBC "static" part has been installed in an alternate directory.

We also use ${UCA_EBC_DATA} environment variable to reference the data directory ("variable" part) of UCA for EBC. The default value for the ${UCA_EBC_DATA} environment variable is /var/opt/UCA-EBC. The ${UCA_EBC_DATA} environment variable thus references the /var/opt/UCA-EBC directory unless UCA for EBC "variable" part has been installed in an alternate directory.

Since UCA-EBC V2.0, the ${UCA_EBC_DATA} directory may contain multiple instances of UCA-EBC. In this document, we will use the value ${UCA_EBC_INSTANCE} for referring to ${UCA_EBC_DATA}/instances/<instance-name> directory. At installation, a single <instance-name> is configured: default.

Chapter 2
Getting started with UCA for EBC

2.1 Software Prerequisites

2.1.1 Operating system

To know on which OS you can use the Development Toolkit, refer to [R1] Unified Correlation Analyzer for Event Based Correlation - Installation Guide

2.1.2 Java JRE/JDK

The following table lists the Java JRE[1]/JDK pre-requisites for UCA for EBC Development Kit:

Table 2 - Java JRE/JDK Prerequisites for UCA for EBC Development Kit

Software	Version	Supported
Java JDK	1.6.0 or later	Yes
Java JDK	1.7.0 or later	Yes and Recommended
Java JDK	1.8.0 or later	No

You can check whether Java is already installed on your system and which version of the Java JRE/JDK is installed by issuing the following commands:

On Windows XP, Windows Vista, Windows 7, and Windows Server 2012:

To check if you already have Java installed, open a command-line (Run... -> cmd.exe) and type:

```
C:\> java -version
```

You should get an output similar to the following:

```
java version "1.6.0_17"
Java(TM) SE Runtime Environment (build 1.6.0_17-b04)
Java HotSpot(TM) Client VM (build 14.3-b01, mixed mode, sharing)
```

Alternatively to using the command-line, you can check if you already have Java installed by going to the Control Panel and selecting the Java icon. In the Java tab, you will find information on the Java version installed on your system.

The latest JDK package for Windows XP, Windows Vista, Windows 7, and Windows Server 2012 can be downloaded (for free) from www.hp.com/go/java

On Linux:

---

[1] Java 1.6 JRE is enough for using the UCA for EBC Development Kit. However the JDK comes with some useful debugging tools (jconsole, jvisualvm, etc...) that may prove helpful for troubleshooting. It is therefore recommended to install the JDK.

To check if Java is already installed:

```
$ rpm -qa | grep jdk
```

Red Hat Enterprise Linux Server comes with OpenJDK Java VM. You should get an output similar to the following (here 1.6.0 and 1.7.0 are installed):

```
java-1.6.0-openjdk-1.6.0.0-1.41.1.10.4.el6.x86_64
java-1.6.0-openjdk-devel-1.6.0.0-1.41.1.10.4.el6.x86_64
java-1.7.0-openjdk-1.7.0.9-2.3.4.1.el6_3.x86_64
java-1.7.0-openjdk-devel-1.7.0.9-2.3.4.1.el6_3.x86_64
```

You can also download (for free) the latest Java packages (HotSpot Java VM) from Oracle from http://java.com/en/download/manual.jsp. If this is installed (usually under /usr/java), you should get an output similar to the following:

```
jdk-1.7.0_75-fcs.x86_64
```

2.1.3 Eclipse IDE

The UCA for EBC Development Kit has been designed for an easy integration with the Eclipse Integrated Development Environment (IDE) tool.

Before starting the development of any UCA for EBC value pack, it is necessary to download and install the Eclipse ™ application development environment.

The following table lists the Eclipse IDE pre-requisites for UCA for EBC Development Kit:

Table 3 - Eclipse IDE Prerequisites for UCA for EBC Development Kit

Software	Version
Eclipse IDE	3.7 (Indigo) or higher

The minimum version of Eclipse IDE required by the UCA for EBC Development Kit is version 3.4 but we recommended Eclipse IDE version 3.7 (Indigo) or higher.

If you already have Eclipse IDE installed on your system, you can either use this version with the UCA for EBC Development Kit (provided this version complies with the version requirement: version 3.4 or higher) or you can install a new version of Eclipse IDE.

If you want to install Eclipse IDE, please go to the following URL for downloading Eclipse IDE: http://www.eclipse.org/downloads/

At the time of writing, the Eclipse IDE version is Neon 4.6.

We recommend you to download either (other choices may also be valid):

- Eclipse IDE for Java Developers, or
- Eclipse IDE for Java EE Developers

Then you need to choose to install either the 32-bit or 64-bit version of Eclipse IDE depending on whether you have a 32-bit or 64-bit operating system.

2.1.4 Installing UCA for EBC and UCA for EBC Development Kit

Detailed information on how to install UCA for EBC and UCA for EBC Development Kit is provided in the [R1] `Unified Correlation Analyzer for Event Based Correlation - Installation Guide`

2.1.5 Post-install environment setup

2.1.5.1 The UCA_EBC_DEV_HOME variable

The UCA for EBC Development Kit installation procedure adds the `${UCA_EBC_DEV_HOME}` environment variable to your user environment.

This variable is necessary for various development phases of a UCA for EBC value pack development, especially the build and packaging phases.

To verify that this variable is correctly set after the UCA for EBC Development Kit has been installed, please follow the instructions below.

On Windows:

Open a command-line (Run... -> cmd.exe) and type:

```
C:\> echo %UCA_EBC_DEV_HOME%
```

You should get an output similar to the following:

```
C:\UCA-EBC-DEV\3.4\
```

📄 NOTE: On Windows 7, you should log out and log back in again for the new environment variable to be taken into account after installation of the UCA for EBC Development Kit.

On Linux:

Open a terminal window and type:

```
$ echo ${UCA_EBC_DEV_HOME}
```

You should get an output similar to the following:

```
/opt/UCA-EBC-DEV
```

📄 NOTE: On Linux this variable must be manually set in the user's environment, as specified in the UCA for EBC Installation Guide.

2.1.5.2 Ant configuration

The UCA for EBC value pack packaging is based on the use of the Apache Ant tool. This tool requires a specific version and specific settings. Be sure to use the Apache Ant tool provided with UCA for EBC in the `%UCA_EBC_DEV_HOME%\3pp\ant` directory (`${UCA_EBC_DEV_HOME}/3pp/ant` on Linux).

Be sure that you don't have the ANT_HOME environment variable set to the path of another version of Apache Ant, which would create conflicts with the version of Apache Ant in the `3pp\ant\bin` folder. If you do, you should either clear the `ANT_HOME` environment variable:

```
C:\> set ANT_HOME=
```

Or set it to the directory of the Apache Ant version that comes with the UCA for EBC development kit:

```
C:\> set ANT_HOME=%UCA_EBC_DEV_HOME%\3pp\ant

$ANT_HOME/bin/ant -version
Apache Ant(TM) version 1.8.2 compiled on December 20 2010
```

The delivered Apache Ant version that comes with the UCA for EBC development kit is:

```
# $ANT_HOME/bin/ant -version
Apache Ant(TM) version 1.8.2 compiled on December 20 2010
```

2.1.6 UCA for EBC Eclipse plug-in installation instructions

The UCA for EBC Development Kit delivers an Eclipse plug-in that eases UCA for EBC value pack project creation under Eclipse.

This plugin is delivered in the `%UCA_EBC_DEV_HOME%\eclipseplugin\ucaEbcEclipsePluginSite-3.3.1-assembly.zip` file.

The installation of this plug-in is made as follows:

From the Eclipse 'Help' menu, choose 'Install new software' and then click on the Add... button.

Select the UCA for EBC eclipse plug-in ZIP file using the Archive... button and give it the name "UCA for EBC plug-in" as shown in the picture below:
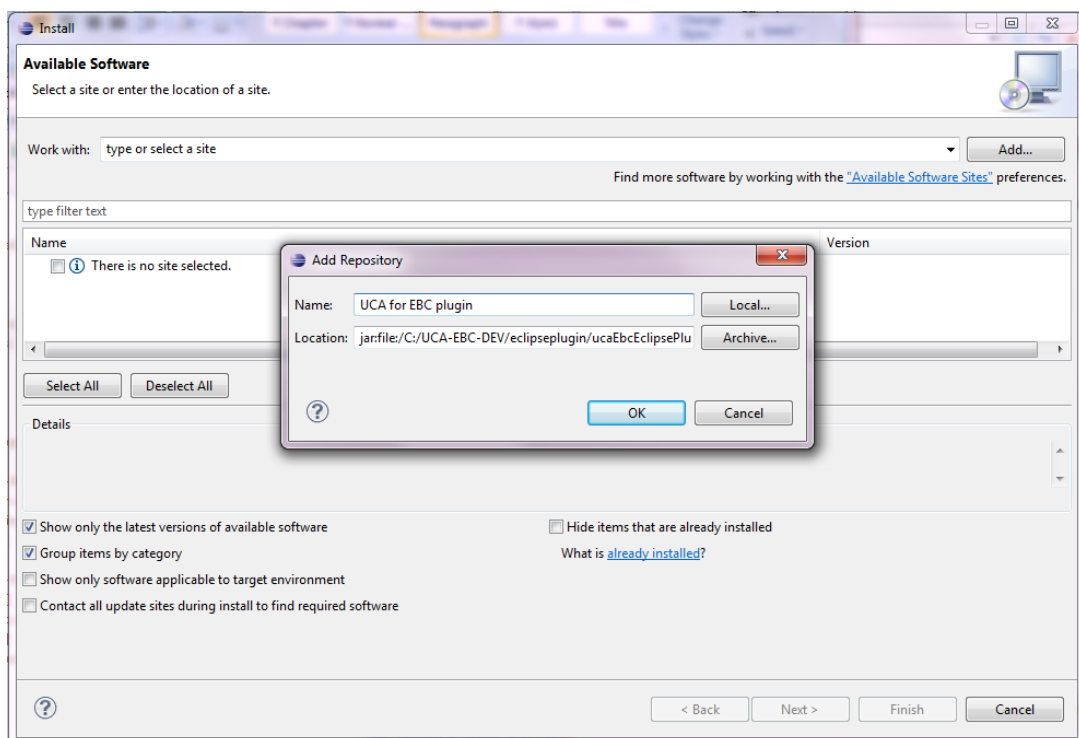
Figure 1 - UCA for EBC Eclipse plug-in: Installation step 1

Then click on the OK button.

The screen should then display the archive content as follow:
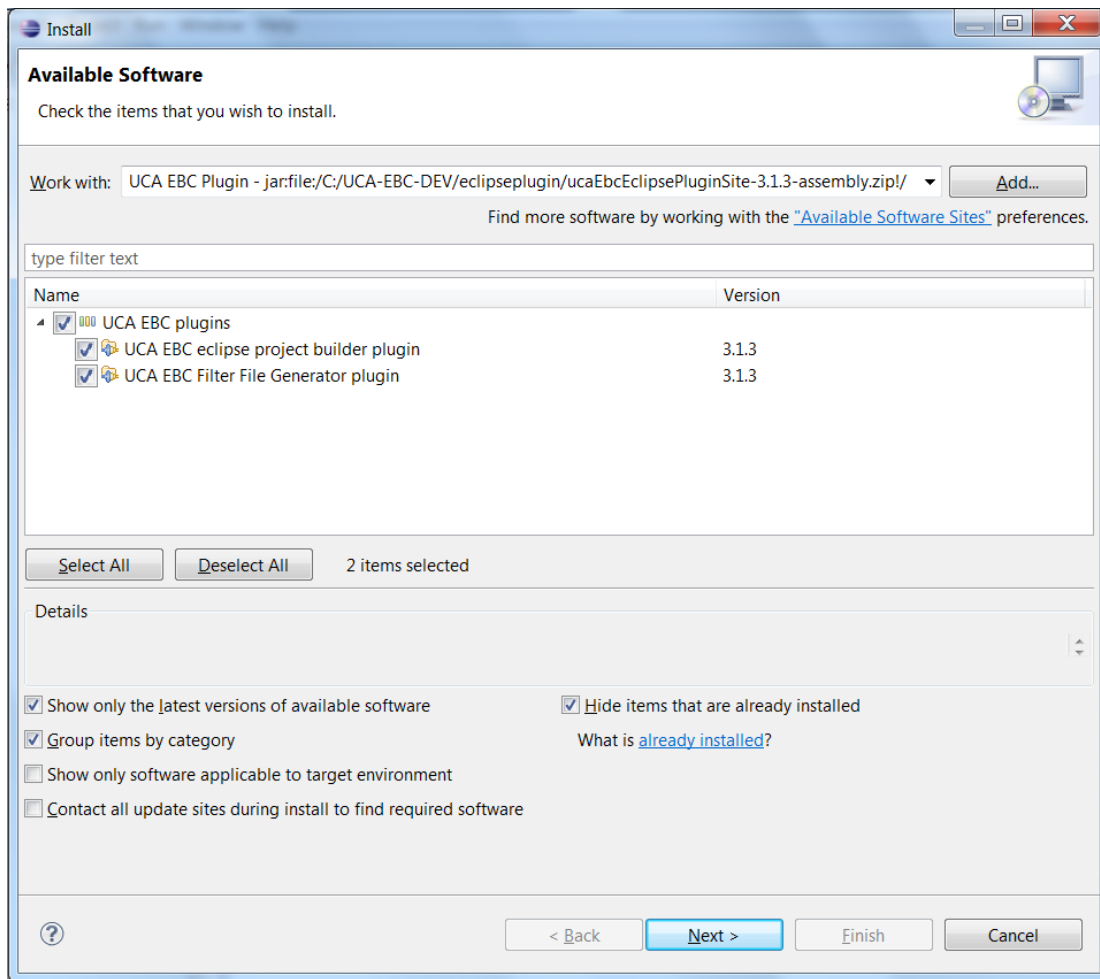
Figure 2 - UCA for EBC Eclipse plug-in: Installation step 2

Check the "UCA EBC plugins" checkbox, uncheck the "Contact all update sites…", and then click on the Next > button.

The following screen is displayed:
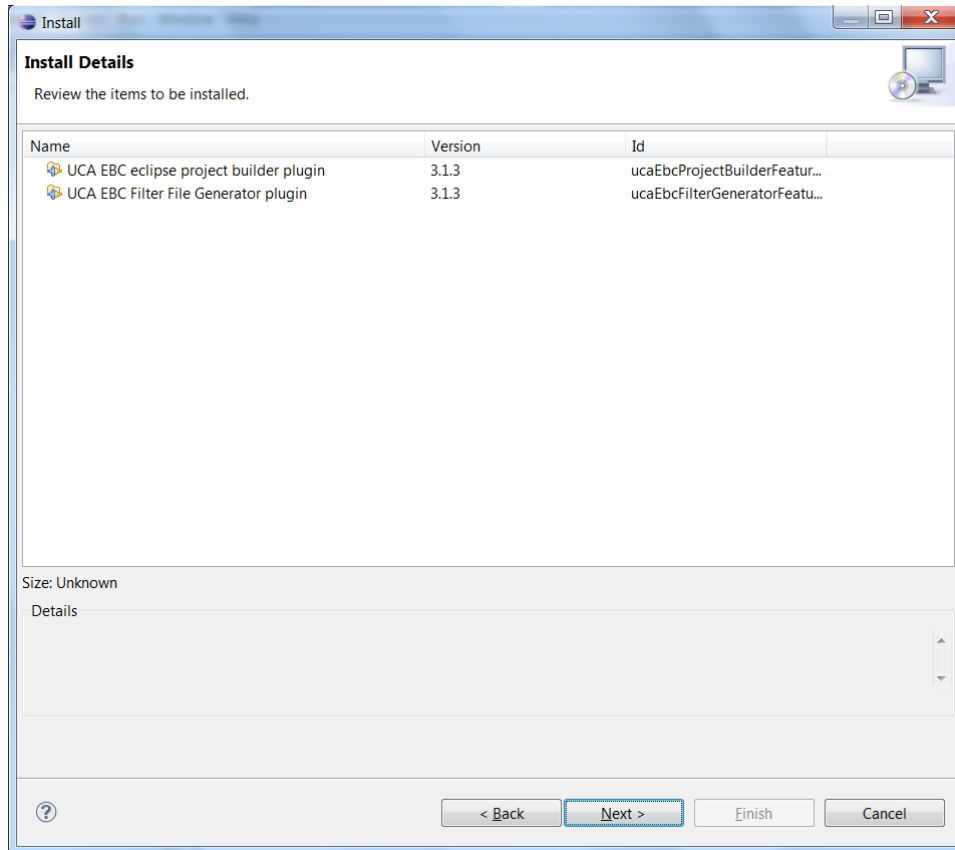
Figure 3 - UCA for EBC Eclipse plug-in: Installation step 3

Click on the Next > button for installing the plug-ins after accepting the license terms.

The following message appears during the installation. This is a normal message as the provided jar files are signed.
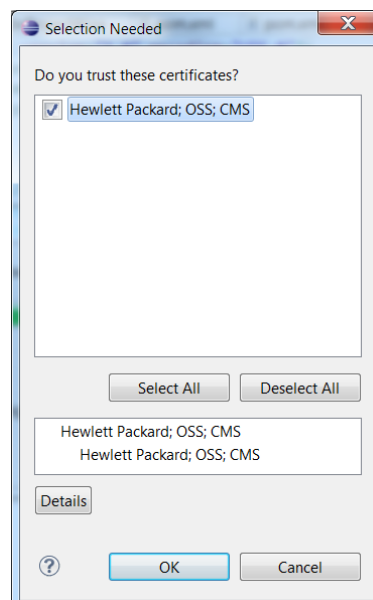
Select the listed Certificated and Click OK to continue the installation.

The plug-in installation requires a restart of your Eclipse IDE environment. Please restart eclipse before any attempt to create a UCA for EBC project.

Chapter 3 Value pack development lifecycle

3.1 Value pack and Scenario definitions

3.1.1 Value Pack Definition

Creating a Value Pack can be seen as implementing a "Correlation" bundle for managing a special correlation use case. The following are example of such correlation use cases:

- a Low Level Filtering use case
- a domain-specific correlation use case like IP MPLS or L2 Metro Ethernet
- a simple 'operator' use case that groups/correlates alarms based on specific rules

A Value Pack is a "functional container" that contains one or more scenarios, each scenario implementing a part of the whole correlation use case targeted by the Value Pack.

Scenarios can be cascaded so that the output of one scenario can be the input of another scenario.

> **NOTE:**
> For additional information about Value Pack and Scenario configuration parameters, please refer to: [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide

3.1.2 Scenario Definition

A scenario is fully defined by implementing the following steps:

- Defining the properties of the scenario
- Defining the filter of the scenario (this will determine what type of alarms will enter the scenario)
- Implementing Alarm enrichment processing (optional)
- Implementing scenario rules

> **NOTE:**
> The first two steps "Scenario definition file" and "Filter definition file" are described in the following document: [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide
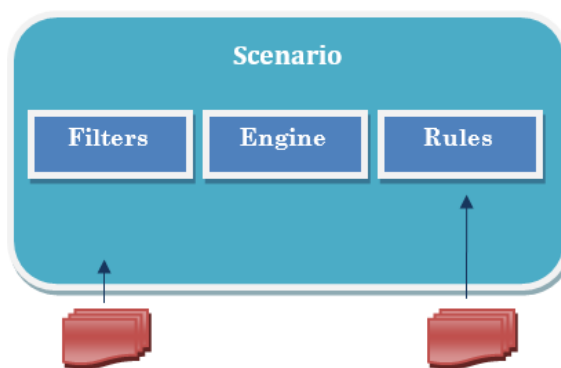
Figure 4 – The UCA for EBC Scenario Components

3.2 Life Cycle

The process of creating a UCA for EBC Value Pack is described by the following figure:
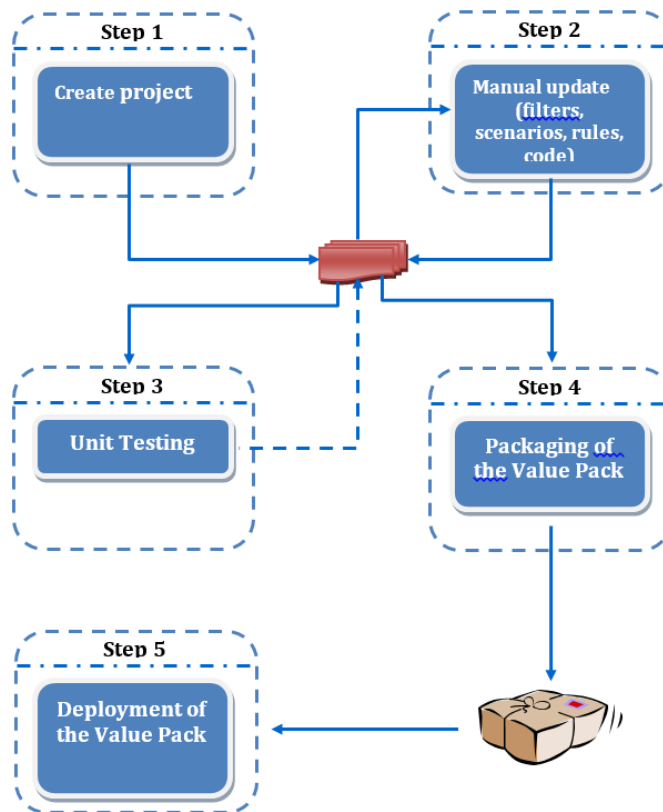
Figure 5 - The 5 steps to create a UCA for EBC Value Pack

For step 1 "Create a new UCA for EBC Value Pack project", use the UCA for EBC project builder eclipse plug-in.

Step 2 "Update the UCA for EBC Value Pack project" is the main step when creating new UCA for EBC Value Packs. This part is explained in details in the next paragraphs and sections.

Step 3 "Develop correlation rules" is also a main step when creating new UCA for EBC Value Packs.

Step 4 is performed automatically using Apache Ant. The `build.xml` file has all necessary targets to compile, test, and generate a ZIP file for your Value Pack.

Step 5 involves copying your Value Pack zip file to the `${UCA_EBC_INSTANCE}/valuepacks` folder on a UCA for EBC Server, as mentioned in Chapter 2 "Getting started with UCA for EBC" of this document.

Developing correlation features involves creating one or more correlation scenarios for your Value Pack, each scenario using its own filter and implementing its own rules.

3.3 Creating a new UCA for EBC Value Pack

UCA for EBC can be seen as an application container in which so called UCA for EBC "Value Packs" are deployed.

A Value Pack represents a set of features (scenarios) that are grouped together to implement one or more correlation use cases.

A UCA for EBC value pack thus includes for example: event filtering, event based rules, customized java code and possibly configuration files for each of these scenarios.

3.3.1 Creating a value pack project within Eclipse

The UCA for EBC eclipse plug-in provides a project creation wizard allowing the creation of a new value pack project in just a few clicks and dialog boxes.

This wizard can be launched from the eclipse main toolbar by clicking on the UCA/EBC icon:
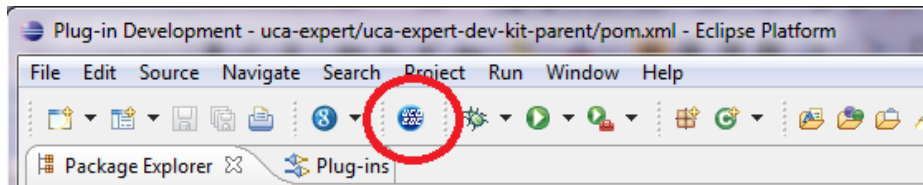
Or from the Eclipse "New Project" Menu as follow:
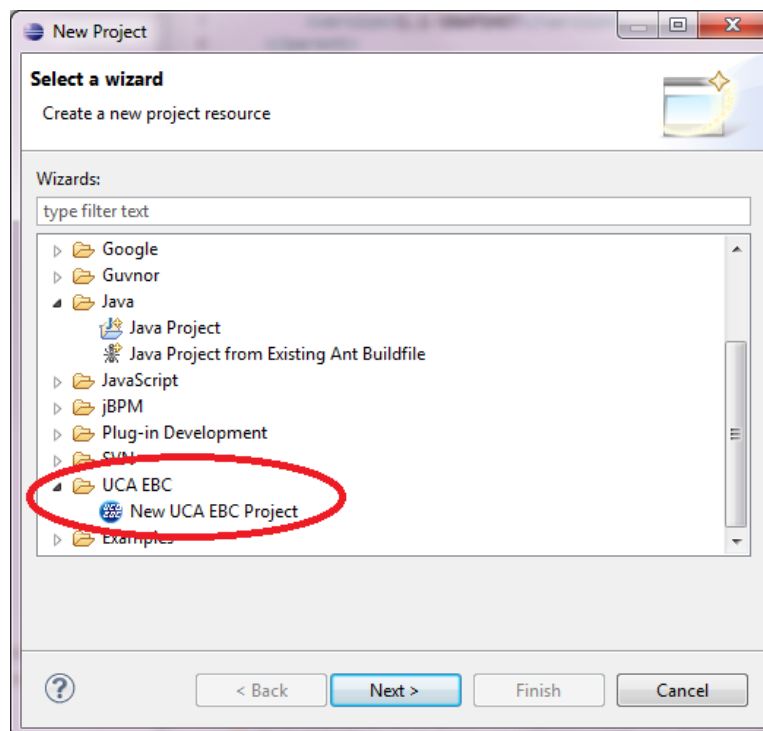
This launches the UCA EBC value pack wizard:
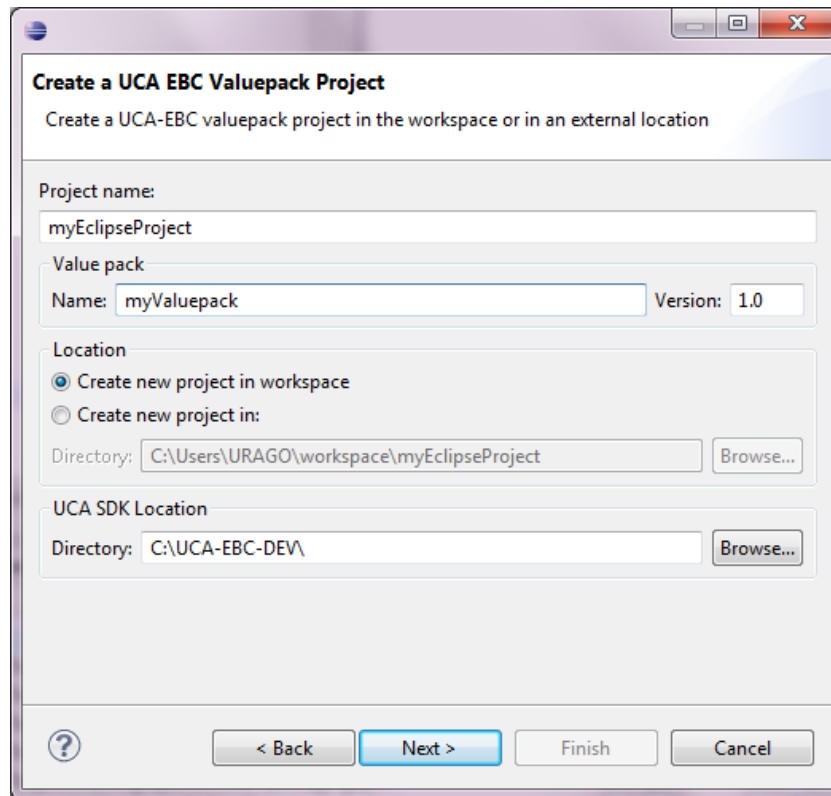
Figure 6 - Value pack project creation wizard Step1

From this panel you can set the project and value pack configuration:

- On the first line you must enter the name of the eclipse project to be created.
- On the second line you need to give the value pack name and its version
- Then the 'location' panel allows specifying the location of the created project. It can be in the current workspace or in an external directory of your choice.
- Finally the UCA SDK Location allows specifying the home directory of the UCA for EBC Development kit. The default value is obtained from the `%UCA_EBC_DEV_HOME%` environment variable.

Then Click on the Next > button for getting the next wizard step.

This is the scenario panel configuration. Note that the project creation wizard allows creating a single initial scenario per value pack. The creation of additional scenarios for a given value pack must be done manually by editing the various value pack configuration files.

Figure 7 - Value pack project creation wizard Step2

At this step you can set the scenario parameters:

- On the first line you must enter the scenario name.
- On the second line you need to give the scenario package name. This package name will be used for all the scenario's java source code files.
- In the filter panel you have to enter the name of the filter file for this scenario. As this is an XML file, the '.xml' suffix is mandatory.
- Then the rule panel allows you specifying the rule file name (and a description) and also specify if this scenario will use template rules file or not (this is done by checking the 'Use template rule' box.

Then Click on the Finish button for creating the Project.

This project creation wizard execution leads to the creation of an Eclipse project skeleton. It exhibits a basic correlation scenario that can compile and unit test successfully. From this example, developers can extend it to build their own Value Packs.
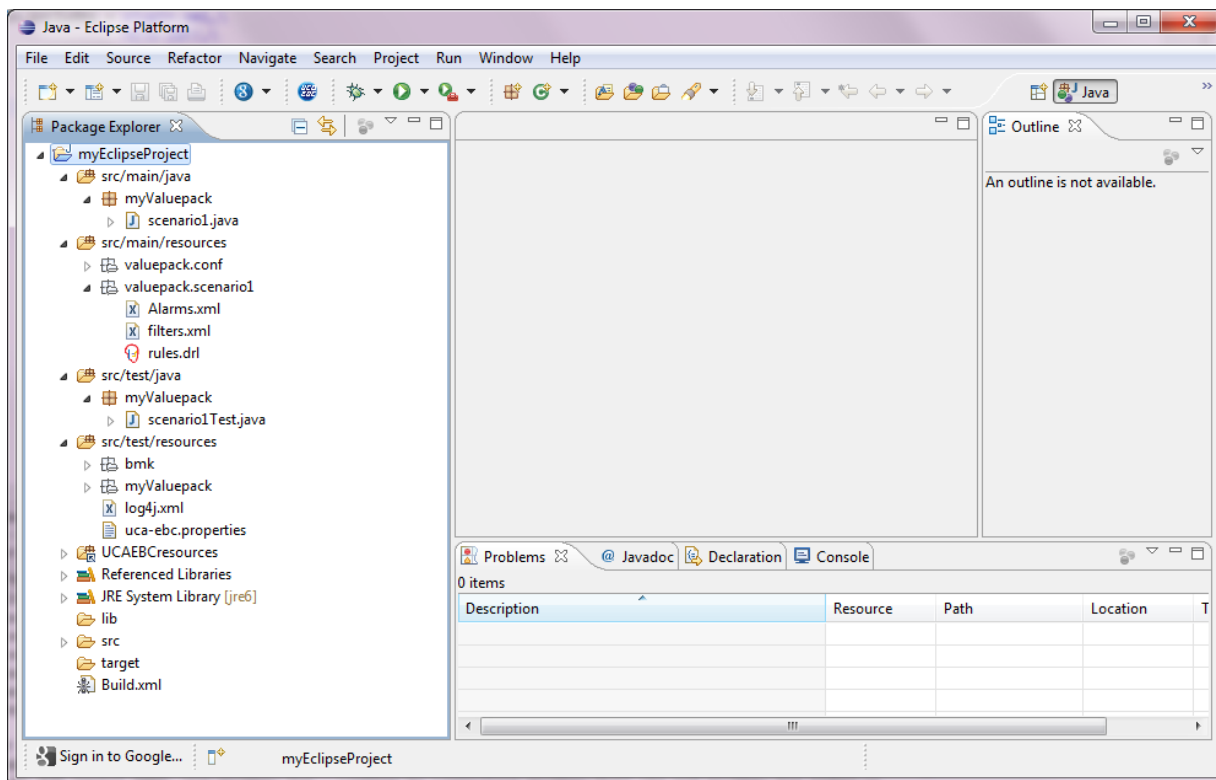
Figure 8 - Created Value pack

NOTES:

For creating "topology based" Value Pack project, please refer to [R8] Unified Correlation Analyzer for Event Based Correlation – Topology Extension

For creating "Inference Machine", "Problem Detection", "Topology State Propagator" Value Pack projects, please refer to [R9] Unified Correlation Analyzer for Event Based Correlation IM – User Guide

3.3.2 Anatomy of the created project

Using Eclipse IDE, you can browse through the different directories that compose the created "Skeleton" project.

Please see below for a glimpse at the folder structure of the created project:
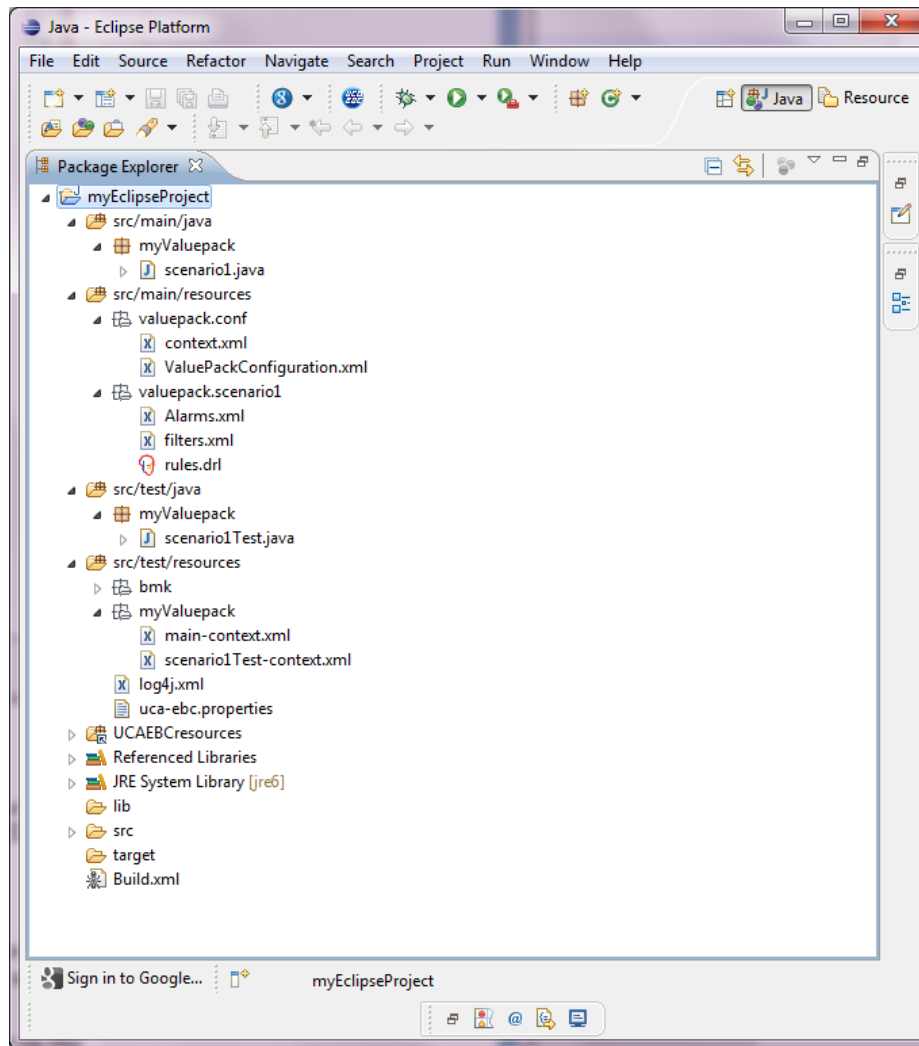
Figure 9 - Folder structure of the created project

The created "Skeleton" project also comes with an Apache Ant build.xml file that is used for building and packaging the value pack outside of the Eclipse IDE.

3.3.3 Validation of the created project

The created project contains predefined test classes that automatically load/compiles the value pack resources (scenario definitions, filters and rules files) and validate them (at least syntactically).

JUnit tests can be run either directly from eclipse, by right-clicking on the test package and choosing "Run As > JUnit Test" as shown in the following screen shot:
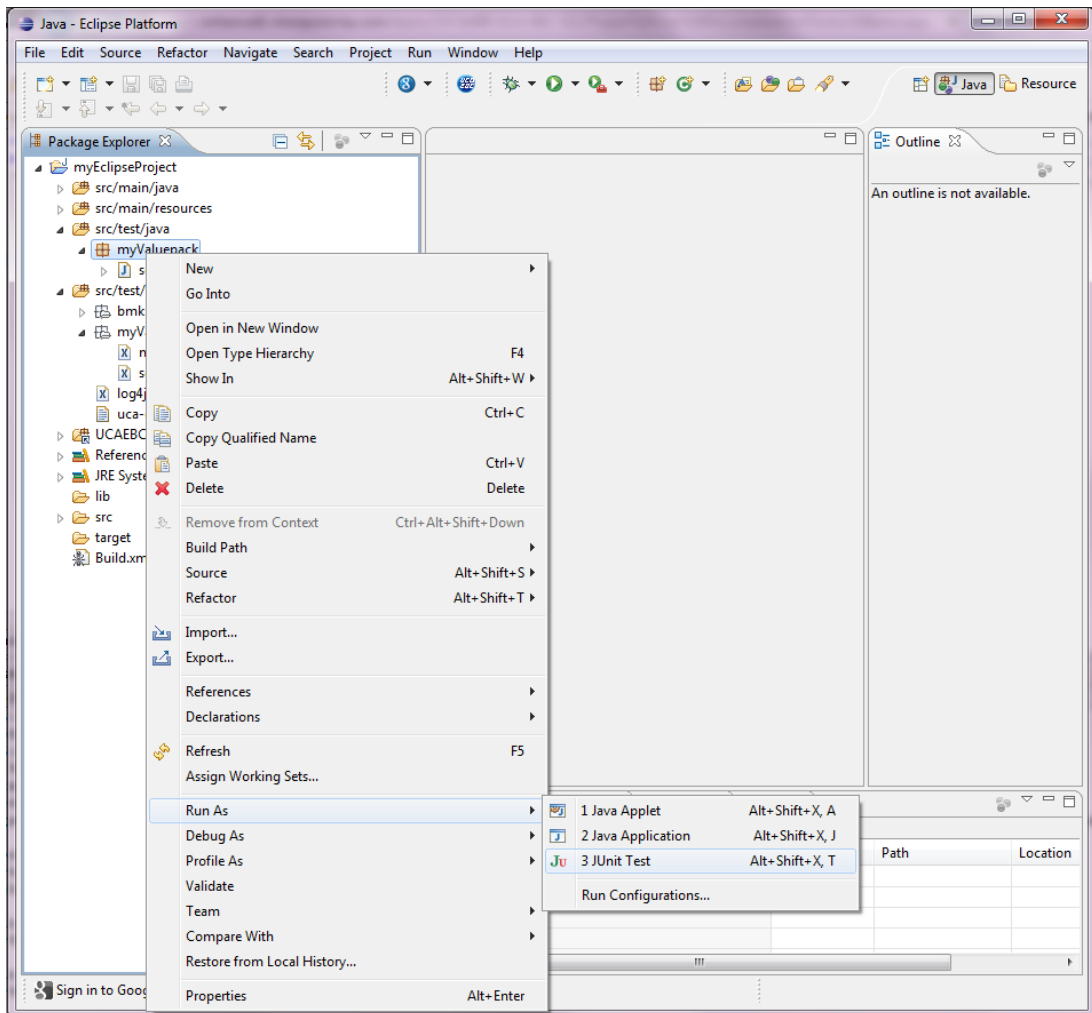
Figure 10 - Running JUnit tests on the created project in Eclipse IDE

In which case the test results can be seen directly in Eclipse IDE:
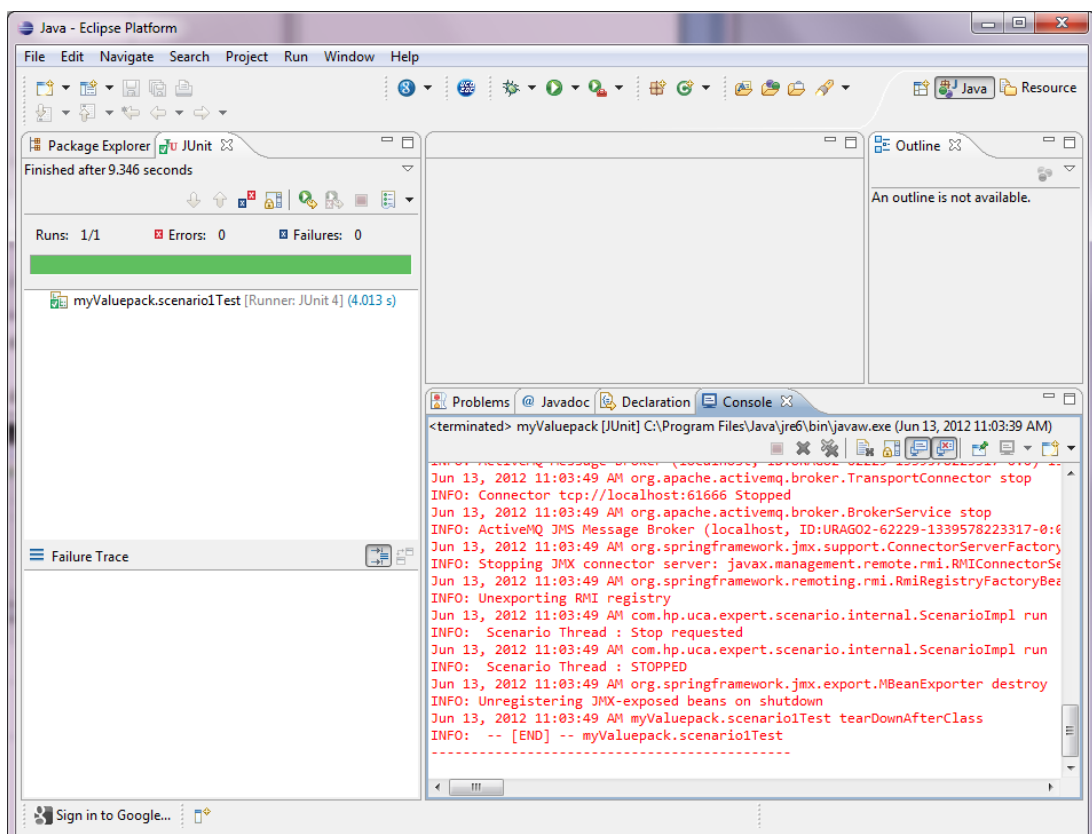
Figure 11 - JUnit tests results on the created project in Eclipse IDE

Or from the command line by executing the Apache Ant tool and selecting the "test" Ant target (You need to run the "**ant test**" command from the root directory of your project workspace) as shown in the following screen shot:

Figure 12 - Running JUnit tests on the created project at the command-line using Ant

In which case the results can be shown in your preferred Web browser by opening the index.html file in the `target\vp-build-dir\reports\junitreport` directory of your project workspace:

Figure 13 - JUnit tests results on the created project viewed using a Web browser

3.4 **Customizing the created 'skeleton' Value Pack project**

The project generated by the UCA for EBC project builder eclipse plug-in provides a simple scenario implementing some basic alarm statistics that is just here for validating the project structure.

Of course you have to turn the created 'skeleton' project into your new Correlation-project value pack. For this you have to customize:

- The Value Pack configuration files
- The scenario filter file
- The scenario rule files
- The associated Java code files

📄 **NOTE:**
For additional information about Value Pack and Scenario configuration parameters, please refer to: [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide

3.4.1 Updating the scenario filters

There is a filter file named `filters.xml` that is associated with the scenario of the created value pack.

The goal of this file is to define the passing filter for Alarms that will be consumed by the current scenario. Then, all alarms entering UCA for EBC will be evaluated against the filter file of each scenario, to decide if they should be forwarded to the scenario or not.

If the properties of an alarm match the passing filter(s) defined in the filters file then the alarm is forwarded to the scenario. On the other hand, if the properties of an alarm don't match the passing filter(s) of the filters file then the alarm is not forwarded to the scenario.

The default generated filter allows any alarm to be forwarded to the scenario.


```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <filters xmlns="http://hp.com/uca/expert/filter">
3     <topFilter name="test">
4        <allCondition>
5           <stringFilterStatement>
6              <fieldName>originatingManagedEntity</fieldName>
7              <operator>matches</operator>
8              <fieldValue>.*</fieldValue>
9              <!--  Or another example of filter (for filtering on the BOX class)
10                <fieldValue>BOX .*</fieldValue>
11             -->
12          </stringFilterStatement>
13       </allCondition>
14    </topFilter>
15 </filters>
16
```

Figure 14 - **The default "catch all" project's filters.xml file**

📄 **NOTES:**
Please refer to: [R2] `Unified Correlation Analyzer for Event Based Correlation - Reference Guide` for a full description of the Filter file syntax.
Refer to section 5.9 of this document for a description on how to use the UCA-EBC eclipse filter editor.

3.4.2 Updating the correlation rules file

By default, the generated rules file defines a single rule implementing a basic statistic use case. This rule is just for demoing and testing. It is just an example, which must be changed to something relevant.

3.5 Generating the Value Pack kit

Once your project has been updated, it is necessary to generate the kit associated with it so that it can be deployed on UCA for EBC (this is the packaging phase). To do this, you just need to execute the following commands:

```
C:\> cd <Project Base>
```

```
C:\> ant all
```


Figure 15 - Building the kit of your customized Value Pack

The kit of the project is then generated in the `target/vp-build-dir/vp` directory of the <Project Base> directory as a zip file called `<my valuepack name>-vp-<my valuepack version>.zip`:
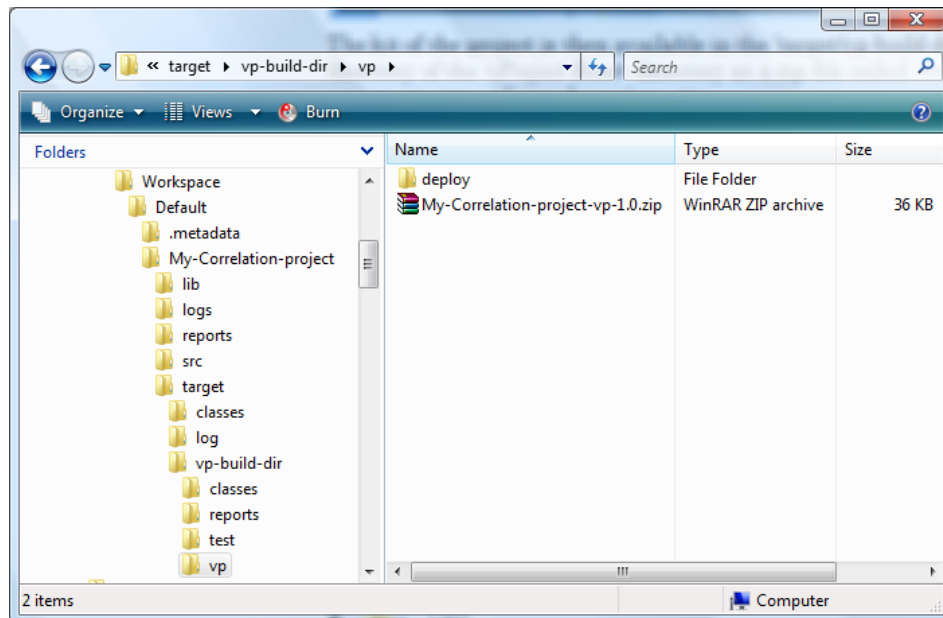
Figure 16 - The kit of your customized Value Pack

The ZIP file of your customized Value Pack contains the following information:

- The Configuration (`conf/`) directory that contains:
    - The Value Pack Spring beans file: `context.xml`
    - The Value Pack configuration file: `ValuePackConfiguration.xml`
- The Library (`lib/`) directory that contains:
    - The JAR file of the Value Pack containing the compiled Java code that you developed for your Value Pack in addition to the rules files
    - Any custom JAR files that you need to run this Value Pack
- The Scenario (`<your-scenario-name>/`) directory that contains:
    - The filters file(s)
    - The external parameters file(s), if your Value Pack contains rules files that are template-based
    - The rule file(s)

```
$ unzip -l target/vp-build-dir/vp/myVP1-vp-1.0.zip
Archive:  target/vp-build-dir/vp/myVP1-vp-1.0.zip
  Length      Date    Time    Name
---------  ---------- -----    ----
        0  05-30-2013 17:46   myVP1-1.0/
        0  05-30-2013 17:46   myVP1-1.0/conf/
        0  05-30-2013 17:46   myVP1-1.0/lib/
        0  05-30-2013 17:46   myVP1-1.0/myScenario1/
     2726  05-30-2013 17:46   myVP1-1.0/conf/ValuePackConfiguration.xml
     1100  05-30-2013 17:46   myVP1-1.0/conf/context.xml
     6423  05-30-2013 17:46   myVP1-1.0/lib/myVP1-lib-1.0.jar
     2596  05-30-2013 17:46   myVP1-1.0/myScenario1/Alarms.xml
      626  05-30-2013 17:46   myVP1-1.0/myScenario1/filters.xml
      420  05-30-2013 17:46   myVP1-1.0/myScenario1/filtersTags.xml
     3299  05-30-2013 17:46   myVP1-1.0/myScenario1/rules.drl
---------                     -------
    17190                     11 files
```

Figure 17 - Contents of the ZIP file of your customized Value Pack

3.6 Deploying the Value Pack kit on UCA for EBC

To deploy your value pack in the UCA server, the following three steps are necessary:

- Install the Value Pack ZIP file on UCA for EBC Server
- Deploy the Value Pack on UCA for EBC Server
- Start the Value Pack on UCA for EBC Server

3.6.1 Install the Value Pack package (ZIP file) on an HP Itanium or Linux system running UCA for EBC Server

Copy your Value Pack package (the ZIP file located at: `target/vp/<my value pack name>vp-<my value pack version>.zip`) to the `${UCA_EBC_INSTANCE}/valuepacks` directory on the UCA for EBC system

For example:

```
$ cp  target/vp-build-dir/vp/myVP1-vp-1.0.zip
${UCA_EBC_DATA}/instances/default/valuepacks/
```

📄 NOTE:
Alternatively, you use UCA-EBC GUI to upload your Value Pack directly on UCA for EBC system without the need of logging into it (just need to log in as admin in GUI application).
Please refer to [R7] *Unified Correlation Analyzer for Event Based Correlation – User Interface Guide*

3.6.2 Deploy the Value Pack

To deploy the Value Pack in the `${UCA_EBC_INSTANCE}/deploy` directory, use the "--deploy" option of the **uca-ebc-admin** administration tool (executed as **uca** user):

```
> cd ${UCA_EBC_HOME}/bin
> uca-ebc-admin --deploy -vpn <my value pack name> -vpv <my value pack
version>
```

You should get an output similar to the following:

```
UCA for EBC Home directory set to: /opt/UCA-EBC
UCA for EBC Data directory set to: /var/opt/UCA-EBC
INFO  - Value Pack name: <my value pack name> version: <my value pack version> has
been successfully deployed
INFO  - Exiting...
```

📄 NOTE:
Alternatively, you can also deploy the value pack from the UCA for EBC GUI.
Refer to [R7] *Unified Correlation Analyzer for Event Based Correlation – User Interface Guide*

3.6.3 Start the Value Pack on UCA for EBC Server

Two different ways are available to you to start value packs deployed on UCA for EBC depending on whether UCA for EBC is started or not.

You can check whether UCA for EBC is running or not by issuing the following command:

```
> ${UCA_EBC_HOME}/bin/uca-ebc show
```

If UCA for EBC is stopped, restarting UCA for EBC will load all value packs deployed in the `${UCA_EBC_INSTANCE}/deploy` folder including your value pack.

If UCA for EBC is running, use the "--start" option of the **uca-ebc-admin** administration tool (executed as <u>uca</u> user) to start your value pack:

```
> cd ${UCA_EBC_HOME}/bin
> uca-ebc-admin --start -vpn <my value pack name> -vpv <my value pack
version>
```

You should get an output similar to the following:

```
UCA for EBC Home directory set to: /opt/UCA-EBC
UCA for EBC Data directory set to: /var/opt/UCA-EBC
INFO  - Exiting...
```

> **NOTE:**
> Alternatively, you can also start the value pack from the UCA for EBC GUI.
> Refer to [R7] Unified Correlation Analyzer for Event Based Correlation – User Interface Guide

You can get the list of running value packs on UCA for EBC using the "--list" option of the **uca-ebc-admin** command-line administration tool:

```
> cd ${UCA_EBC_HOME}/bin
> uca-ebc-admin --list
```

> **NOTE:**
> For additional information about the **uca-ebc-admin** command-line administration tool, please refer to: [R3] Unified
> Correlation Analyzer for Event Based Correlation – Administration, Configuration and Troubleshooting Guide

3.7 Testing the Value Pack in real-time

Now that both UCA for EBC and your value pack are up and running, the UCA for EBC application implements the 'Statistic circuit' correlation package and is ready to listen to incoming alarms.

In order to provide an easy way to test the global solution, a simple tool is provided that lets you inject a set of alarms (defined in a XML file) into UCA for EBC.

As the action provided in the properties file is to "log" information to a log file (in "append" mode), it is easily possible to test the circuit in real-time.

A sample `Alarms.xml` input file containing sample alarms to use with your value pack is provided in the `${UCA_EBC_INSTANCE}/deploy/<your value pack name>-<your value pack version>/skeleton` folder. The output log file named `output.xml` is located in the `${UCA_EBC_HOME}` root folder.

Following is an example of the **uca-ebc-injector** command-line tool used to inject Alarms into UCA for EBC in order to test your Value Pack in real conditions:

```
>${UCA_EBC_HOME}/bin/uca-ebc-injector -file
${UCA_EBC_INSTANCE}/deploy/skeleton-project-1.0/mypackage/Alarms.xml
>tail -f ${UCA_EBC_HOME}/output.xml &
```

You should get an output similar to the following:

```
### STATISTICAL ALARM: 2 Alarms received ###
```

NOTE:
For additional information about the **uca-ebc-injector** command-line tool, please refer to:: [R3] Unified Correlation Analyzer for Event Based Correlation – Administration, Configuration and Troubleshooting Guide

Chapter 4 Focus on development key points

4.1 Implementing Alarm enrichment

Alarm enrichment processing is called by the UCA for EBC framework after the alarm passed the scenario filters and before it is inserted in the scenario Working Memory.

The enrichment is implemented by performing the following steps:

Step 1:   Extend the UCA Java class `com.hp.uca.expert.lifecycle.alarm.`AlarmLifeCycle
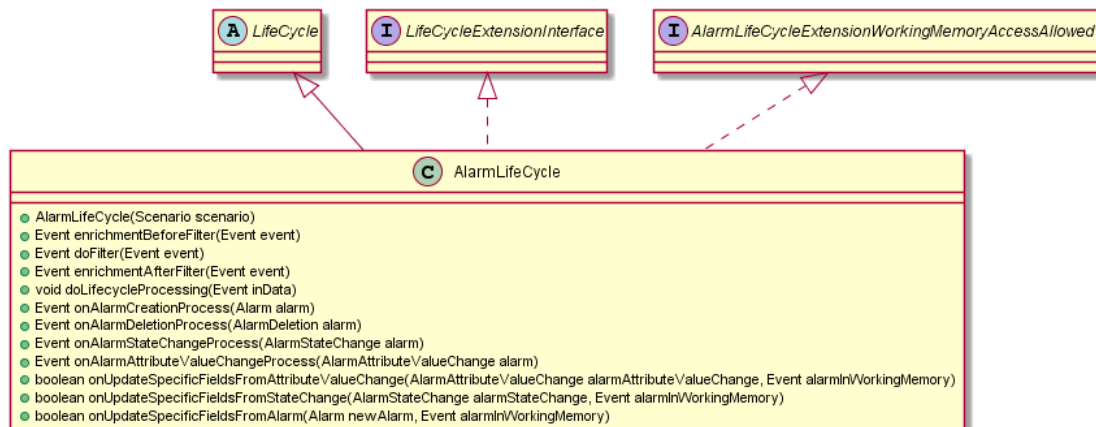
And override the following methods:

- `onAlarmCreationProcess(Alarm alarm)`: to extend alarm creation objects
- `onAlarmDeletionProcess(AlarmDeletion alarm)`: to extend alarm deletion objects
- `onAlarmStateChangeProcess(AlarmStateChange alarm)`: to extend alarm state change objects
- `onAlarmAttributeValueChangeProcess(AlarmAttributeValueChange alarm)`: to extend alarm attribute value change objects

Example of AlarmLifeCycle Extension:

```
package com.hp.uca.ebc.enrichmentexample;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.hp.uca.common.trace.LogHelper;
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmCommon;
import com.hp.uca.expert.lifecycle.alarm.AlarmLifeCycle;
import
com.hp.uca.expert.lifecycle.alarm.AlarmLifeCycleExtensionWorkingMemoryAccessA
llowed;
import com.hp.uca.expert.lifecycle.common.LifeCycleExtensionInterface;
import com.hp.uca.expert.scenario.Scenario;

public class ExtendedLifeCycle extends AlarmLifeCycle implements
            LifeCycleExtensionInterface,
            AlarmLifeCycleExtensionWorkingMemoryAccessAllowed {

    private static Logger log = LoggerFactory
                .getLogger(ExtendedLifeCycle.class);
```

```
    public ExtendedLifeCycle(Scenario scenario) {
        super(scenario);

        /*
         * If needed more configuration, use the context.xml to define
any beans
         * that will be available here using scenario.getGlobals()
         */
        // scenario.getGlobals()
    }

    @Override
    public AlarmCommon onAlarmCreationProcess(Alarm alarm) {
        LogHelper.enter(log, "onAlarmCreationProcess()");

        EnrichedAlarm customAlarm = new EnrichedAlarm(alarm);
        customAlarm.setCustomizedInformation("New Custom Information only
available from CustomAlarm");

        LogHelper.exit(log, "onAlarmCreationProcess()");
        return customAlarm;
    }

}
```

In this example, the enrichment is performed only in the case of an alarm creation event.

Step 2: Declare the `ExtendedLifeCycle` class at the scenario definition Level:

This is done by using the `<customLifeCycleClass>` in the Scenario Definition section of the
`ValuepackConfiguration.xml` file.

Example:

```
<scenarios>
 <scenario name="com.hp.uca.ebc.enrichmentexample.myscenario">
  <alarmEligibilityPolicy>
     NetworkState!=&quot;CLEARED&quot;
  </alarmEligibilityPolicy>
  <filterFile>
   src/main/resources/valuepack/myscenario/filters.xml
  </filterFile>
  <fireAllRulesPolicy>EACH_ACCESS</fireAllRulesPolicy>
  <globals></globals>
  <processingMode>CLOUD</processingMode>
  <rulesFiles>
      <rulesFile>
          <filename>
          file:./src/main/resources/valuepack/myscenario/rules.drl
        </filename>
          <name>my scenario rules</name>
          <ruleFileType>DRL</ruleFileType>
      </rulesFile>
  </rulesFiles>
  <customLifeCycleClass>
     com.hp.uca.ebc.enrichmentexample.ExtendedLifeCycle
  </customLifeCycleClass>
 </scenario>
</scenarios>
```

Step3:   Extend the Alarm object if necessary

In order to ease the rule writing, it may be easier to store the enrichment information in some dedicated alarm object attributes.

In such case the Alarm objects (Alarm, AlarmDeletion, AlarmAttributeValueChange and AlarmStateChange) can be extended.

Example of Alarm extension:

```
package com.hp.uca.ebc.enrichmentexample;
import javax.xml.bind.annotation.XmlRootElement;
import org.neo4j.graphdb.Relationship;
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmHelper;

@XmlRootElement
public class EnrichedAlarm extends Alarm {

    /**
     * New Alarm field
     */
    private String location;

    public EnrichedAlarm() {
        super();
    }

    public EnrichedAlarm (Alarm alarm) {
        super(alarm);
    }


    @Override
    public EnrichedAlarm clone() throws CloneNotSupportedException
 {
        EnrichedAlarm newAlarm = (EnrichedAlarm) super.clone();
        newAlarm.location = this.location;
        return newAlarm;
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    @Override
    public String toFormattedString() {
        StringBuffer toStringBuffer=
AlarmHelper.toFormattedStringBuffer(this);

        AlarmHelper.addFormatedItem(toStringBuffer, "Location:",
getLocation());

        return toStringBuffer.toString();
    }

}
```

4.2 Developing the scenario rules

Rules files are files containing correlation rules interpreted by the Drools inference engine of the scenario.

The Drool Expert engine used in UCA for EBC has its own rule language. The rule file content must comply with this language.

Please refer to Drools Expert guide, Chapter 5 "The Rule Language" for a description of the language: http://www.jboss.org/drools/documentation

> **IMPORTANT:**
> Drools keywords for inserting, updating, and deleting objects in Working Memory (i.e. **insert**, **update**, **retract**) MUST NOT be used directly when developing UCA-EBC rules. This is for working memory integrity, and due to the locking mechanism implemented within the UCA-EBC framework.
> - Instead of using `insert(myObject)` directly, you should use `theScenario.getSession().insert(myObject)` from Drools files or `ScenarioThreadLocal.getScenario().getSession().insert(myObject)` from Java code
> - Instead of using `update(myObject)` directly, you should use `theScenario.getSession().update(myObject)` from Drools files or `ScenarioThreadLocal.getScenario().getSession().update(myObject)` from Java code
> - Instead of using `retract(myObject)` directly, you should use `theScenario.getSession().retract(myObject)` from Drools files or `ScenarioThreadLocal.getScenario().getSession().retract(myObject)` from Java code
>
> The `ScenarioThreadLocal` class is located in the `com.hp.uca.expert.scenario` package. Also, all timer based keywords should be avoided: **duration, timer, calendar**.

On top of the basic rule language syntax, additional operators are available to deal with time constraints:

Temporal operator: see *Drools Fusion guide, Chapter 2.4. Temporal Reasoning*

Sliding Time Window Feature: see *Drools Fusion guide, Chapter 2.6. Sliding Time Window*

See https://docs.jboss.org/drools/release/5.6.0.Final/drools-fusion-docs/html_single/ for more information on how to create rules that deal with time constraints.

> **NOTE:** To use the sliding time window feature, objects in working memory must be declared as Event (and not as Fact).
>
> Please see Drools Fusion guide, Chapter 2.1. **"Events semantics"** at URL https://docs.jboss.org/drools/release/5.6.0.Final/drools-fusion-docs/html_single/ for more information on what events are compared to facts and how to declare them.

4.2.1 Basics

Any rules file contains one or multiple rules, and has a '.drl' extension.

Here are the different parts composing a rule file:

```
package package-name

imports

globals

functions

queries
```

```
rules
```

Package

The package name is optional, but it is recommended to partition your rules in different packages for clarity.

Imports

The "imports" part, allows you to import Java classes that can be used in the `Action` or `Condition` parts of a rule.

> **IMPORTANT:**
> In UCA for EBC, importing the Alarm Java class (com.hp.uca.expert.alarm.Alarm) is necessary in order to be able to use alarm attributes in rule conditions.

Globals

The "globals" part is used to define variables that have a global scope (across rules). The global variables have to be initialized by the application.

Functions

Functions let you define functions that let you avoid repeating the same lines of code over the entire rules file.

Queries

UCA for EBC does not currently provide support for queries.

Rules

The rules define the behaviour of the expert system.

Please refer to Drools Expert guide, for a full description of rule files:
https://docs.jboss.org/drools/release/5.6.0.Final/drools-expert-docs/html_single/

4.2.2 Sample rules on Alarm facts in CLOUD mode

In CLOUD mode, the UCA for EBC system inserts Alarm facts in Working Memory and these facts remain infinitely in working memory unless they are specifically removed in the rules (using the retract statement). This retract statement is generally done in the right end side part of rules.

UCA for EBC contains an Alarm Java class (`com.hp.uca.expert.alarm.Alarm`) which represents a "generic" Alarm as a fact. Rules can rely on attributes and services of the Alarm object. For instance, testing a specific value of an attribute in the condition part or setting a specific attribute of the Alarm in the action part.

To use the CLOUD mode, the scenario processing mode must be set to "CLOUD" in the `ValuePackConfiguration.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      name="myValuepackName" version="myValuepackVersion">
     <scenarios>
    <scenario name="myScenario">
            <filterFile>${uca.home}/myValuePack/myScenario/myScenario-
filter.xml</filterFile>
            <fireAllRulesPolicy>WATCHDOG</fireAllRulesPolicy>
            <globals>
```

```
        </globals>
        <processingMode>CLOUD</processingMode>
        <rulesFiles>
            <rulesFile>
    <filename>file:${uca.home}/myValuePack/myScenario/myScenarioRules.drl</
filename>
                <name>myRules</name>
                <ruleFileType>DRL</ruleFileType>
            </rulesFile>
        </rulesFiles>
    </scenario>
 </scenarios>
</valuePackConfiguration>
```

Here is a simple example that identifies "Similar alarms" (i.e. Alarms that have the same alarm type, managed object and probable cause as another Alarm). This example illustrates a case where the UCA for EBC engine is in CLOUD processing mode.

The rule file called `myScenarioRules.drl` contains a rule, the "Similar Alarm" rule, which performs the following processing:

When an alarm 'a' is found in Working Memory (with a severity different from 'clear') and if there is another not cleared (severity different from 'clear') alarm (this !=a) with the same attribute values for the `originatingManageEntity`, `alarmType` and `probableCause` properties then display a text.

```
package scenario.sample;

import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;

rule "Similar Alarm"
when
        a: Alarm(perceivedSeverity != PerceivedSeverity.CLEAR)
        a1: Alarm(
    this != a &&
    perceivedSeverity != PerceivedSeverity.CLEAR &&
    originatingManagedEntity == a.originatingManagedEntity &&
    alarmType == a.alarmType &&
    probableCause == a.probableCause)

    then
  System.out.println("Executing: "+drools.getRule().getName());
        System.out.println(a1.getIdentifier() + "similar to "+
a.getIdentifier());
end
```

IMPORTANT:
Importing the Alarm Java class (`com.hp.uca.expert.alarm.Alarm`) is necessary. Declaring the Alarm class as a Fact in the "declare" section of the rules file is not mandatory however. By default, if they are not declared at all, objects are understood to be Facts in Working Memory.

Another rule, the "Clear Alarm" rule focuses on cleared alarms:

```
rule "Clear Alarm"
when
        a: Alarm(perceivedSeverity != PerceivedSeverity.CLEAR)
        a1: Alarm(
    perceivedSeverity == PerceivedSeverity.CLEAR &&
    originatingManagedEntity == a.originatingManagedEntity &&
    alarmType == a.alarmType &&
    probableCause == a.probableCause &&
    timeInMilliseconds > a.timeInMilliseconds)
```

```
        then
    System.out.println("Executing: "+drools.getRule().getName());
          System.out.println(a1.getIdentifier() + " clears "+
a.getIdentifier());
end
```

NOTE: The **drools** object in the sample rule code above is a predefined Drools java object that you can use in the Action part of a rule to get information on the rule itself among other things. In our example, the method `drools.getRule().getName()`, called from a rule's Action part, returns the name of the rule. See https://docs.jboss.org/drools/release/5.6.0.Final/drools-expert-docs/html_single/ for more information on the drools predefined object.

4.2.3 Sample rules on Alarm events in STREAM mode

In STREAM mode, UCA for EBC inserts Alarm events in Working Memory only for a period of time. After that, Alarm events are automatically removed from working memory.

To use the STREAM mode, the scenario processing mode must be set to "STREAM" in the `ValuePackConfiguration.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      name="myValuepackName" version="myValuepackVersion">
      <scenarios>
          <scenario name="myScenario">
              <filterFile>${uca.home}/myValuePack/myScenario/myScenario-
filter.xml</filterFile>
              <fireAllRulesPolicy>EACH_ACCESS</fireAllRulesPolicy>
              <globals>
              </globals>
              <processingMode>STREAM</processingMode>
              <rulesFiles>
                  <rulesFile>

      <filename>file:${uca.home}/myValuePack/myScenario/myScenarioRules.drl</
filename>
                  <name>myRules</name>
                  <ruleFileType>DRL</ruleFileType>
              </rulesFile>
          </rulesFiles>
          </scenario>
  </scenarios>
</valuePackConfiguration>
```

IMPORTANT:
Importing the Alarm Java class (`com.hp.uca.expert.alarm.Alarm`) is necessary. Declaring the Alarm class as an Event in the "declare" section of the rules file is also mandatory.
By default, if they are not declared at all, objects are understood to be Facts in Working Memory. So, declaring Alarms as Events is mandatory.
Please see Drools Fusion guide, Chapter 2.1. "Events semantics" at URL http://docs.jboss.org/drools/release/5.6.0.Final/drools-fusion-docs/html/ch02.html#d0e184, for more information on what events are compared to facts and how to declare them.

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;
```

```
declare Alarm
    @role( event )
    @timestamp( timeInMilliseconds )
    @expires( 30m )
end
```

The above "Alarm" declaration specifies that:

- Alarms should be treated as Events in Working Memory, not Facts
- The timeInMilliseconds attribute (i.e. the EventTime attribute of the Alarm) is used as the timestamp of the Alarm instead of the time when the Alarm Event is actually inserted into working memory, which is the default timestamp for Events in Working Memory. The timestamp of the Alarm Event plays a role when time constraints are used in rules.
- Alarm Events expiration time is 30 minutes: the Alarm Events will be removed from working memory automatically after 30 minutes.

Generally, rules in STREAM mode are used to identify patterns of Events (Events that occurs in a specific order) during a specific time window.

The "Store not cleared Alarm" rule is an example of such a rule in STREAM mode. It performs the following rules:

When an alarm 'a' is in Working Memory (an alarm on a "BOX" item with a severity different from 'clear') and if there are no other alarms (matching specific criterias) received within 2 seconds of alarm 'a' then the AdditionalInformation attribute of alarm 'a' is updated

```
rule "Store not cleared Alarm"
when
        a: Alarm(originatingManagedEntity matches "BOX .*" &&
                perceivedSeverity != PerceivedSeverity.CLEAR)

        not Alarm(originatingManagedEntity == a.originatingManagedEntity
&&
                perceivedSeverity == PerceivedSeverity.CLEAR &&
                this after[ 0s, 2s ] a)

then
        System.out.println("Executing rule:
"+drools.getRule().getName()+" on " + a.getAdditionalText());

        // Add the correlation time and rule name in the Additional
Information Field of the alarm
        Date now=new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("EEE MMM dd HH:mm:ss
zzz yyyy",
        Locale.FRENCH);
        a.setAdditionalInformation("correlated by rule:
"+drools.getRule().getName()
                +" at " +sdf.format(now));

        // Store the alarm
        acmeActionManager.doDummyAction(a);
end
```

NOTE: The JBoss Drools documentation contains a lot of other examples of rules in both STREAM (Drools Fusion) and CLOUD (Drools Expert) modes. As writing the correlations rules is the major undertaking of creating a correlation project, it is highly recommended to constantly refer to the Drools documentation when writing Rules. Please see http://www.jboss.org/drools/documentation for documentation on how to write rules for Drools Expert and Drools Fusion.

4.2.4 Defining and using rule templates

For information about rule templates, please refer to: [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide

4.2.5 Introducing Java code in the rules

Drools rules files natively support Java code in the consequence part of the rules (after the "then" keyword). All you have to do is import the packages/classes that you need in the import section of the rules files and then write Java code referencing these classes.

For example, you declare the `java.util.Date` class in the rules file:

```
template header
timeslot


package com.hp.uca.expert.vp.llef.grouping;

#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import com.hp.uca.expert.example.hibernate.AlarmDao;


import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;
import java.util.ArrayList;
import java.util.Iterator;

import com.hp.uca.expert.scenario.ScenarioPublic;
import com.hp.uca.common.trace.LogHelper;
import com.hp.uca.expert.flag.Flag;
import com.hp.uca.expert.testmaterial.AbstractJunitIntegrationTest;

#declare any global variables here
global AlarmDao alarmDAO;
global ScenarioPublic theScenario;
```

Then you can create and use `java.util.Date` objects in the consequence part (after the "then" keyword) of your rules:

```
// Description: find a root cause and the associated symptoms in a given time
window
// Constraints:
//    - the root cause is not cleared during the time window
template "Update Root Cause with Symptoms no clearance received"
rule "Update Root Cause with Symptoms no clearance received"

    when
            [...]

    then
        LogHelper.enter(theScenario.getLogger(),
drools.getRule().getName(),rootAlarm.getOriginatingManagedEntity()+" - "+
rootAlarm.getAdditionalText());

        // Add the correlation time and rule name in the Additional
Information Field of the alarm
        Date now=new Date();
```

```
        SimpleDateFormat sdf = new SimpleDateFormat("EEE MMM dd HH:mm:ss
zzz yyyy",
        Locale.FRENCH);
      String addInfo="correlated by rule: "+drools.getRule().getName()
              +" at " +sdf.format(now) + "\nAssociated sympthoms:\n";
```

The `java.util.Date` objects that you create are not stored in Working Memory unless you do so explicitly using the "insert" statement.

📄 NOTE: For more information, please see the Drools documentation: <u>http://www.jboss.org/drools/documentation</u>

4.3 Defining your own beans

Spring beans (corresponding to the external Java services that you want to use) are defined in the `context.xml` of your Value Pack.

Here below is an example of a bean named "dbForwarder" that is relevant for forwarding alarms into an SQL data store.

```
<bean id="dbForwarder" class="com.hp.uca.expert.alarm.JDBCAlarmForwarder">
        <property name="alarmDao" ref="alarmDao" />
</bean>
```

You can define any bean in this file.

In order to retrieve the Java instance of that bean object, you will need to use following API in your value pack:

```
Scenario.getValuePack().getApplicationContext()
```

In order to retrieve the Spring ApplicationContext that will allow you to retrieve your bean.

With above example, typical code would have been:

```
return (JDBCAlarmForwarder) theScenario.getValuePack()
                    .getApplicationContext().getBean("dbForwarder");
```

4.4 Executing external actions from the rules

External actions in rules are basically any action that either uses UMB framework services, or OSS Open Mediation framework services or external Java services.

There are three categories of external actions that will be described in the following sections:

- **Standard external actions**: these actions use the Action class, defined by the UCA for EBC framework, to execute actions on the Unified Mediation Bus framework (i.e. execute actions on any application connected to the UMB framework using a mediation adapter) or on the legacy OSS Open Mediation framework (i.e. execute actions on any application connected to the OSS Open Mediation framework using a Channel Adapter)
- **Calling services defined using Spring**: Spring beans are defined in the `context.xml` of your Value Pack and global variables that reference these Spring beans are defined in your scenario(s) and used in your rule file(s).
- **Forwarding alarms or events to external systems**: Alarm and Event forwarders are defined using Spring beans and used from the rules to forward alarms or events to files, JMS queues/topics, the UMB framework, the OSS Open Mediation framework, or any database that has a JDBC interface

4.4.1 Standard external actions

Standard external actions are defined as actions that are to be executed by the UMB framework of by the OSS Open Mediation framework.

The UCA for EBC framework defines a Java class named Action that you can use to perform standard external actions in rules, like for example executing a shell script or a TeMIP directive on a TeMIP director.

In order to be able to use the methods of the Action class, you have to import the class in the "import" part of the rule file:

```
package com.hp.uca.expert.action;


#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.CustomFields;
import com.hp.uca.expert.x733alarm.CustomField;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;
import com.hp.uca.expert.x733alarm.NetworkState;          // NOT_CLEARED,
CLEARED
import com.hp.uca.expert.x733alarm.OperatorState;         // NOT_ACKNOWLEDGED,
ACKNOWLEDGED, TERMINATED
import com.hp.uca.expert.x733alarm.ProblemState;          // NOT_HANDLED,
HANDLED, CLOSED
import com.hp.uca.mediation.action.client.Action;
import com.hp.uca.mediation.action.jaxws.ActionResponseItem;
import java.util.ArrayList;
```

Then you can create Action objects in the "then" part of a rule as described in the example below:

```
# Display properties of any new alarm

rule "Any Not Acknowledged Alarm (Action)"
 when
      a: Alarm(operatorState == OperatorState.NOT_ACKNOWLEDGED)
 then
      System.out.println("[RULE " + drools.getRule().getName() + "] Found not
acknowledged alarm: identifier = " + a.getIdentifier() + ":");
      System.out.println(a.toFormattedString());

  // Acknowledging the Alarm
  Action action = new Action("TeMIP_AO_Directives_localhost ");
  action.addCommand("directiveName", "ACKNOWLEDGE");
  action.addCommand("entityName", a.getIdentifier());
action.addCommand("UserId", "UCA Expert");
theScenario.addAction(action); // Associate the action with the scenario
      System.out.println("Executing synchronous ACKNOWLEDGE directive on
alarm: " + a.getIdentifier());
  action.executeSync();
      System.out.println("Done:");
      System.out.println("    - ActionId = " + action.getActionId());
      System.out.println("    - ActionStatus = " + action.getActionStatus());
      System.out.println("    - ActionStatusExplanation = " +
action.getActionStatusExplanation());
      if (!action.getListActionResponseItem().isEmpty()) {
              System.out.println("    - ActionResponseItems = ");
              // Loop through all action response items
          for (ActionResponseItem item :
action.getListActionResponseItem()) {
              if (!item.getOutput().getEntry().isEmpty()) {
                  // Loop through all output entries
                      for (ActionResponseItem.Output.Entry entry :
item.getOutput().getEntry()) {
```

```
                                System.out.println("                -> " +
entry.getKey() + " = " + entry.getValue());
                        }
                    }
            }
        }
        else {
            System.out.println("    - ActionResponseItems = none");
        }
        System.out.println("    - RawText = " + action.getRawTextAsString());
 end
```

If you use the new **UMB mediation layer** then you have two options to create the new Action object in the source code of your rule:

- • Either using the **ActionReference** parameter:

```
Action action = new Action("TeMIP_AO_Directives_localhost");
```

The value of this parameter must match an Action Reference defined in `${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml` file

- • Or using the **targetAdapter** and **targetActionName** parameters:

The value of these parameters must match a couple targetAdapter / targetActionName couple defined in `${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml` file

If you use the legacy **OSS Open Mediation (NOM) layer** then you also have two options to create the new Action object in the source code of your rule:

- • Either using the **ActionReference** parameter:

```
Action action = new Action("TeMIP_AO_Directives_localhostNOM");
```

The value of this parameter must match an Action Reference defined in `${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml` file

- • Or using the **NMS Name**, **Service Name**, **Mvp Name** and **Mvp Version** parameters:

The Mvp Name and Version must match a Mediation Value Pack MvpName and MvpVersion attributes in the `${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml` file

Here's the content of a sample `ActionRegistry.xml` file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ActionRegistryXML xmlns="http://registry.action.mediation.uca.hp.com/">

    <MediationValuePack MvpName="temip"
                        MvpVersion="1.0"

    url=http://localhost:26700/uca/mediation/action/ActionService?WSDL
brokerURL="failover://tcp://localhost:10000">

            <Action actionReference="TeMIP_AO_Directives_localhostNOM">
                <ServiceName>aoDirective</ServiceName>
                <NmsName>localTeMIP</NmsName>
            </Action>
            <Action actionReference="TeMIP_TT_Directives_localhostNOM">
                <ServiceName>ttDirective</ServiceName>
                <NmsName>localTeMIP</NmsName>
```

```
                    </Action>
                    <Action actionReference="TeMIP_FlowManagementNOM">
                            <ServiceName>subscriptionManagement</ServiceName>
                            <NmsName>localTeMIP</NmsName>
                    </Action>
            </MediationValuePack>

            <MediationValuePack MvpName="exec"
                                MvpVersion="1.0"
    url=http://localhost:26700/uca/mediation/action/ActionService?WSDL
    brokerURL="failover://tcp://localhost:10000">
                    <Action actionReference="Exec_localhost">
                            <ServiceName>commandsExecution</ServiceName>
                            <NmsName>localhost</NmsName>
                    </Action>
            </MediationValuePack>

            <!-- UMB Actions -->
            <UMBActions>
                    <UMBAction actionReference="TeMIP_AO_Directives_localhost"
    targetAdapterName="TeMIP" targetActionName="AOAction"/>

                    <UMBAction actionReference="TeMIP_TT_Directives_localhost"
    targetAdapterName="TeMIP" targetActionName="TTAction"/>

                    <UMBAction
    actionReference="TeMIP_Passthrough_Directives_localhost"
    targetAdapterName="TeMIP" targetActionName="PassthroughAction"/>

            </UMBActions>

    </ActionRegistryXML>
```

Please refer to [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide for more information on how to use the `Action` class or configure the `ActionRegistry.xml` file.

Please refer to [R6] OSS Open Mediation V7.2 – Installation and Configuration Guide for more information on how to configure OSS Open Mediation V7.2 to support the execution of Actions.

Once you have created an Action object, you can specify the parameters that will define what action to perform, in the following example a TeMIP directive:

```
action.addCommand("directiveName", "ACKNOWLEDGE");
action.addCommand("entityName", a.getIdentifier());
action.addCommand("UserId", "UCA Expert");
```

Using the `addCommand()` method you can specify the key/value pairs to use as parameters to the Action object. These parameters depend on the type of Action to perform.

For acknowledging a TeMIP Alarm, you need to specify the key/value pairs as shown above: specifying the `UserId` of the user acknowledging the alarm is optional, just like in TeMIP.

Then, you need to associate the Action to the current Scenario so that the Action can be properly processed:

```
theScenario.addAction(action);
```

Then, you need to execute the Action. Both synchronous and asynchronous actions are possible. Only one of the following lines of code is necessary, depending on whether you want to execute a synchronous or asynchronous action:

```
action.executeSync();
action.executeAsync(AODirectiveKey.ENTITY_NAME);
```

Synchronous actions are "blocking". The `action.executeSync()` call will block the execution of the rule until the action is completed. The whole rule engine for the scenario is blocked while the action is being executed.

Asynchronous actions are "non blocking". This is the reason why they are the recommended method for executing actions. The `action.executeAsync(…)` call doesn't block the execution of the rule. The rules continue to be executed.

There's a mandatory parameter to the `action.executeAsync(…)` method: the `synchronizationKey`. This key indicates the name of the action command key that will be used to synchronize asynchronous actions so that the order of asynchronous actions referring to the same action command key/value pair is preserved.

The `synchronizationKey` parameter enables you to preserve some kind of order among all the asynchronous actions triggered by your rules. By default (if you specify `Action.NO_SYNCHRONIZATION_KEY` as the synchronization key) there is no order. All asynchronous actions are executed in parallel by a pool of threads. There is no guarantee that the asynchronous actions will be executed in the order in which they were requested.

If you do not need asynchronous actions to be executed in any specific order, then you can use `Action.NO_SYNCHRONIZATION_KEY` as the synchronization key when calling the `action.executeAsync(…)` method.

On the other hand, if you need all asynchronous actions to be executed in the order they are requested, you need to use a command key (specified with the `action.addCommand(key, value)` method) that has the same value for all asynchronous actions as the synchronization key.

If you need only groups of asynchronous actions to be executed in the order they are requested, you need to use a command key (specified with the `action.addCommand(key, value)` method) that has the same value for all asynchronous actions of the same group as the synchronization key.

For example, for executing TeMIP AO Directives you can use the `AODirectiveKey.ENTITY_NAME` as synchronization key:

```
...
Action action = new Action("TeMIP_AO_Directives_localhost");
action.addCommand(AODirectiveKey.DIRECTIVE_NAME, AODirective.SET);
action.addCommand(AODirectiveKey.ENTITY_NAME, "OPERATION_CONTEXT OC1
ALARM_OBJECT 155");
action.addCommand(AODirectiveKey.ADDITIONAL_TEXT, "my text");
theScenario.addAction(action)
...
action.executeAsync(AODirectiveKey.ENTITY_NAME);
...
```

In the example above, as long as you execute TeMIP AO Directives using the `action.executeAsync(AODirectiveKey.ENTITY_NAME)` syntax, all TeMIP AO Directives actions on the same entity will be executed in the order that they are called.

If you do not want to use the synchronization key feature, you can pass `null` or `Action.NO_SYNCHRONIZATION_KEY` to the `executeAsync(…)` method:

```
...
action.executeAsync(Action.NO_SYNCHRONIZATION_KEY);
...
```

NOTE: For more information on synchronous and asynchronous actions (including how to use synchronization keys for asynchronous actions), please refer to: [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide.

Once the action has been performed on the Network Management System the result of the execution of the action can be retrieved using the following methods:

```
action.getActionStatus();
action.getActionStatusExplanation();
```

Other methods of the Action class provide even more detailed information on the result of the execution of the action. See the Java Documentation for the Action class for more information.

4.4.1.1 Writing Actions for the UMB TeMIP Mediation Adapter or for the OSS Open Mediation TeMIP Channel Adapter

The delivered value pack examples come with a `lib/` directory containing the TeMIP mapper jar file:

```
lib/uca-mediation-temip-mvp-mapper-keys-3.4.jar
```

This will allow you to benefit from java classes that have been designed to help you write rules that execute TeMIP Alarm Object (AO) directives or TeMIP Trouble Ticket (TT) directives (provided the TeMIP mediation adapter (UMB) or channel adapter (NOM) is deployed).

To do so, the first step is to add the following import statement in your rules file:

```
import com.hp.uca.temip.mvp.mapper.*;
```

Below is the list of classes that you can use to help you write rules (all AO classes are defined in the `com.hp.uca.temip.mvp.aodirective.mapper` package, while TT classes are defined in the `com.hp.uca.temip.mvp.ttdirective.mapper` package).

There are 2 sets of classes. The first set contains classes that define constants that should be used in the "key" part when using the `Action.addCommand(key, value)` method:

Table 4 - Java helper classes for TeMIP adapter

Class name	Class description
`AODirectiveKey` in `com.hp.uca.temip.mvp.aodirective.mapper` package	Contains string constants that list all the possible values for keys when using the `Action.addCommand(key, value)` method on AO Directives
`TTDirectiveKey` in `com.hp.uca.temip.mvp.ttdirective.mapper` package	Contains string constants that list all the possible values for keys when using the `Action.addCommand(key, value)` method on TT Directives

The most important constant in the `AODirectiveKey` class is the **`AODirectiveKey.DIRECTIVE_NAME`** (or the `TTDirectiveKey.DIRECTIVE_NAME` in the `TTDirectiveKey` class depending on whether you want to execute AO or TT directives).

Using this constant, you can define the name of the TeMIP Alarm Object (or Trouble Ticket) directive that you wish to execute:

```
...
Action action = new Action("TeMIP_AO_Directives_localhost");
action.addCommand(AODirectiveKey.DIRECTIVE_NAME, AODirective.SET);
...
```

```
theScenario.addAction(action);
...
action.executeAsync(AODirectiveKey.ENTITY_NAME);
...
```

The other constants define the names of AO (or TT) Directive parameters or attributes that you can use. For example:

```
...
Action action = new Action("TeMIP_AO_Directives_localhost");
action.addCommand(AODirectiveKey.DIRECTIVE_NAME, AODirective.SET);
action.addCommand(AODirectiveKey.ENTITY_NAME, "OPERATION_CONTEXT OC1
ALARM_OBJECT 155");
action.addCommand(AODirectiveKey.ADDITIONAL_TEXT, "my text");
theScenario.addAction(action);
...
action.executeSync();
...
```

The second set contains classes that define constants that should be used in the "value" part when using the Action.addCommand(key, value) method.

Below is the list of such classes for Alarm Object directives (besides the `AODirectiveKey` class that is explained above):

Table 5 - AO directives helper classes

Class name	Class description
AlarmClassType	Contains string constants that list all the possible values for the Alarm_Class attribute (of the SET directive for example). These constants should be used in the value part when using the Action.addCommand(key, value) method
AlarmObjectProblemStatus	Contains string constants that list all the possible values for the Problem_Status attribute (of the DUMP or SET directives for example)
AlarmObjectState	Contains string constants that list all the possible values for the State attribute (of the DUMP or SET directives for example) and the Previous_State attribute (of the SET directive for example)
AlarmOriginType	Contains string constants that list all the possible values for the Alarm_Origin attribute (of the SET directive for example)
AlarmType	Contains string constants that list all the possible values for the Alarm_Type attribute (of the CREATE, DUMP or SET directives for example)
AODirective	Contains string constants that list all the possible values for Alarm Object directive names (ACKNOWLEDGE, ADDPARENT, ARCHIVE, … for example)
AutomaticOperationsSeverity	Contains string constants that list all the possible values for the Automatic_Terminate_On_Close attribute (of the SET directive for example)

DeleteCondition	Contains string constants that list all the possible values for the State attribute (of the DELETE directive for example)
EntityScope	Contains string constants that list all the possible values for the entityScope attribute (of any directive)
EventID	Contains string constants that list all the possible values for the EventID attribute (of the GETEVENT directive for example)
Partition	Contains string constants that list all the possible values for the Partition attribute (of any directive)
ProbableCause	Contains string constants that list all the possible values for the Probable_Cause attribute (of the CREATE, DUMP or SET directives for example)
SecurityAlarmCause	Contains string constants that list all the possible values for the Security_Alarm_Cause attribute (of the CREATE, DUMP or SET directives for example)
Severity	Contains string constants that list all the possible values for the Severity (of the ARCHIVE directive for example), Perceived_Severity (of the CREATE, DELETE, DUMP, or SET directives for example), or Original_Severity (of the SET directive for example) attributes
SummarizeScope	Contains string constants that list all the possible values for the Scope attribute (of the DUMP directive for example)
TrendIndication	Contains string constants that list all the possible values for the Trend_Indication attribute (of the CREATE or SET directives for example)

Below is the list of such classes for Trouble Ticket (TT_SERVER) directives (besides the `TTDirectiveKey` class that is explained above):

Table 6 - TT directives helper classes

Class name	Class description
AttributeId	Contains string constants that list all the possible values for the AttributeId attribute (of the SHOW directive). These constants should be used in the value part when using the Action.addCommand(key, value) method
AutoResponseType	Contains string constants that list all the possible values for the Type attribute (of the ASSOCIATETT, CANCELTT, CLOSETT, CREATETT or DISSOCIATETT directives)
Partition	Contains string constants that list all the possible values for the Partition attribute (of any directive)

RegisterOperationType	Contains string constants that list all the possible values for the Operation attribute (of the REGISTER directive)
TTDirective	Contains string constants that list all the possible values for Trouble Ticket directive names (ASSOCIATETT, CANCELTT, CLEARALL, CLOSETT, CREATE ... for example)

The most important class in this set is the AODirective class (or the TTDirective class of Trouble Ticket directives) that lists all possible Alarm Object directive names (ACKNOWLEDGE, ADDPARENT, ARCHIVE, ... for example):

```
...
Action action = new Action("TeMIP_AO_Directives_localhost");
action.addCommand(AODirectiveKey.DIRECTIVE_NAME, AODirective.SET);
...
theScenario.addAction(action);
...
action.executeAsync(AODirectiveKey.ENTITY_NAME);
...
```

The other classes contain constants that define the list of possible value for AO Directive (or TT Directive) parameters or attributes.

```
...
Action action = new Action("TeMIP_AO_Directives_localhost");
action.addCommand(AODirectiveKey.DIRECTIVE_NAME, AODirective.SET);
action.addCommand(AODirectiveKey.ENTITY_NAME, "OPERATION_CONTEXT OC1
ALARM_OBJECT 155");
action.addCommand(AODirectiveKey.TREND_INDICATION,
TrendIndication.LESSSEVERE);
action.addCommand(AODirectiveKey.PROBABLE_CAUSE, ProbableCause.LOSSOFSIGNAL);
theScenario.addAction(action);
...
action.executeSync();
...
```

You can use Eclipse IDE's automatic completion feature (the keyboard shortcut for this feature is: CTRL+<Space>) to discover the constants defined in each of the classes mentioned above.

4.4.1.2 Writing Actions for the UMB Exec Mediation Adapter or for the OSS Open Mediation Exec Channel Adapter

The delivered value pack examples come with a lib directory containing the exec mapper jar file:

```
lib/uca-mediation-exec-mvp-mapper-keys-3.4.jar
```

To create an Exec Action for the UMB Exec adapter of for the OSS Open Mediation Exec adapter, you must first add the following import statement in your rule file:

```
import com.hp.uca.exec.mvp.mapper.*;
```

This will allow you to benefit from java classes that have been designed to help you write rules that execute command/executables/shell scripts (provided the Exec mediation adapter (UMB) channel adapter (OSS Open Mediation) is deployed).

Below is the list of classes that you can use to help you write rules (all classes are defined in the com.hp.uca.exec.mvp.mapper package):

Table 7 - Java helper classes for Exec adapter

Class name	Class description
ExecActionKey	Contains string constants that list all the possible values for keys when using the `Action.addCommand(key, value)` method

Here's an example of the `ExecActionKey` class use:

```
...
Action action = new Action("Exec_localhost");
action.addCommand(ExecActionKey.COMMAND, "ping");
action.addCommand(ExecActionKey.ARGUMENT, "127.0.0.1");
...
theScenario.addAction(action);
...
action.executeSync();
...
```

4.4.2 Calling services defined using Spring

Sometimes the actions performed in the THEN part of rules will be calls to nonstandard Java package services such as Hibernate, JMS... These services generally need to be initialized and the Spring configuration file of the Value Pack, *context.xml*, is one way to do it.

In order to be able to use these services from Drools rules files, Drools global variables need to be defined that reference the Spring beans defined in the *context.xml* file of the value pack.

Any service defined using Spring can be "retrieved" in any rule file using the "**global**" keyword.

Below is an excerpt from the Drools Expert documentation that explains the concept of global variables:

[…] With global you define global variables. They are used to make application objects available to the rules. Typically, they are used to provide data or services that the rules use, especially application services used in rule consequences, and to return data from the rules, like logs or values added in rule consequences, or for the rules to interact with the application, doing callbacks. Globals are not inserted into the Working Memory, and therefore a global should never be used to establish conditions in rules except when it has a constant immutable value. The engine cannot be notified about value changes of globals and does not track their changes. Incorrect use of globals in constraints may yield surprising results – surprising in a bad way.

If multiple packages declare globals with the same identifier they must be of the same type and all of them will reference the same global value. […]

Please refer to the [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide for more information about the Spring Framework integration with UCA for EBC.

First, in order to be able to use Spring beans in rules files, the Spring beans must be declared in the `context.xml` file of the Value Pack. Then global variable entries must be defined for each Spring bean in the `ValuePackConfiguration.xml` file as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     name="__PROJECT_NAME__" version="__PROJECT_VERSION__">
   <scenarios>
```

```
        <scenario name="Grouping-Scenario">

        <filterFile>src/main/resources/com/hp/uca/expert/vp/llef/grouping/group
ing-filter.xml</filterFile>
            <fireAllRulesPolicy>EACH_ACCESS</fireAllRulesPolicy>
            <globals>
                <global>
                    <key>alarmDAO</key>
                    <value>alarmDAO</value>
                </global>
            </globals>
            <processingMode>STREAM</processingMode>
            <rulesFiles>
                <rulesFile>
        <filename>file:./src/main/resources/com/hp/uca/expert/vp/llef/grouping/
grouping-template.drl</filename>
                    <name>grouping</name>

        <paramsFilename>file:./src/main/resources/com/hp/uca/expert/vp/llef/gro
uping/grouping-params.xml</paramsFilename>
                    <ruleFileType>XDRL</ruleFileType>
                </rulesFile>
            </rulesFiles>
        </scenario>
    </scenarios>
</valuePackConfiguration>
```

The `<globals>` XML tag in the *ValuePackConfiguration.xml* file defines a list (i.e. a Java map) of beans that will be available in your rules file(s) as global variables.

The following piece of code illustrates the use of external Java libraries from rule files:

```
package com.hp.uca.expert.example.hibernate;

#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;

…
import com.hp.uca.expert.example.hibernate.AlarmDao;
…
#declare any global variables here
global AlarmDao alarmDAO;
…

template "Root Cause without Symptom"
rule "Root Cause without Symptom"
    when
        …
then
…
    // Store the root cause alarm
alarmDAO.save(fatherAlarm);
…
```

Java class import → (import com.hp.uca.expert.example.hibernate.AlarmDao;)

Definition of global variables ← (global AlarmDao alarmDAO;)

External action using global variable ← (alarmDAO.save(fatherAlarm);)

4.4.3 Forwarding alarms to external systems

A common use case is when you want to forward alarms being processed by a scenario to external systems/applications.

You might want to create an XML file containing some alarms that you want to export from the scenario so that you can import these alarms on an external system/application.

Alternatively, if the external system/application that you want to export alarms to has a JMS queue/topic that can be used to import alarms, then you might want to export alarms directly to this JMS queue/topic.

Finally, if the external system/application is accessible from OSS Open Mediation V7.2 via a specific Channel Adapter, then you might want to export the alarms directly to the OSS Open Mediation V7.2 bus. If the external system/application has a UMB mediation adapter, then you can export the alarms or events through UMB as explained in section 4.4.4 Forwarding events through UMB.

The UCA for EBC framework defines standard classes that enable you forwarding Alarm objects (or collections thereof) located in Drools Working Memory or that have been defined in the rules of a scenario to either a file, a JMS queue/topic or OSS Open Mediation V7.2.

The following Java classes are part of the UCA for EBC framework:

- To forward alarms to a file:
  `com.hp.uca.expert.alarm.FileAlarmForwarder`
- To forward alarms to a JMS queue/topic: `com.hp.uca.expert.alarm.JMSAlarmForwarder`
- To forward alarms to OSS Open Mediation V7.2:
  `com.hp.uca.expert.alarm.OpenMediationAlarmForwarder`
- To persist alarms into a DB store:
  `com.hp.uca.expert.alarm.JDBCAlarmForwarder`

Please refer to UCA for EBC Javadoc for complete information on these classes. The Javadoc for UCA for EBC is located at `${UCA_EBC_DEV_HOME}/apidoc`

One way to forward alarms is to define an AlarmForwarder (either FileAlarmForwarder, JMSAlarmForwarder, OpenMediationAlarmForwarder or JDBCAlarmForwarder) bean in the Spring configuration file of the scenario (`context.xml`).

> 📄 NOTE: Please note that the recommended way for defining alarm forwarders is to define them in the Spring configuration file of the scenario: `context.xml`.
>
> A Thread is associated with each alarm forwarder (either FileAlarmForwarder, JMSAlarmForwarder, OpenMediationAlarmForwarder, or JDBCAlarmForwarder). This thread is automatically started when the associated AlarmForwarder object is created. If the AlarmForwarder has been created using the recommended method (in the Spring configuration file of the scenario: `context.xml`) then the associated thread will be automatically stopped when the bean associated with the alarm forwarder is destroyed. Otherwise you need to use the requestStop() method to explicitly stop the thread associated with the alarm forwarder when you don't need it anymore.
>
> The thread associated with an alarm forwarder provides compression to improve performance. Alarms may not be forwarded right away. They are accumulated in a queue for the duration of the compression period (by default 1 second) so that they can be forwarded as a batch of alarms at the end of the compression period (by default every second). You can change the value of the compression period using the setCompressionPeriod(long) method. If you set the compression period to 0 milliseconds, no compression will be performed.

Here's an example of defining such a bean in the `context.xml` file of a scenario:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jms="http://www.springframework.org/schema/jms"
       xmlns:p="http://www.springframework.org/schema/p"
xmlns:context="http://www.springframework.org/schema/context"
       xmlns:amq="http://activemq.apache.org/schema/core"
xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans.xsd
                          http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd
                          http://www.springframework.org/schema/jms
                          http://www.springframework.org/schema/jms/spring-
jms.xsd
                          http://activemq.apache.org/schema/core
                          http://activemq.apache.org/schema/core/activemq-
core.xsd">


        <context:annotation-config />

        <bean name="forwardedAlarmsFile" class="java.io.File">
                <constructor-arg index="0"><value>forwarded-
alarms.xml</value></constructor-arg><!-- String pathname -->
        </bean>

        <bean name="fileAlarmForwarder"
class="com.hp.uca.expert.alarm.FileAlarmForwarder" depends-
on="forwardedAlarmsFile">
                <constructor-arg index="0"><ref
bean="forwardedAlarmsFile"/></constructor-arg><!-- File file -->
                <constructor-arg index="1"><value>false</value></constructor-
arg><!-- boolean overwrite -->
        </bean>

        <bean name="jmsAlarmForwarder"
class="com.hp.uca.expert.alarm.JMSAlarmForwarder">
                <constructor-arg
index="0"><value>vm://localhost?broker.persistent=false</value></constructor-
arg><!-- String brokerURL -->
                <constructor-arg
index="1"><value>jms.alarm.forwarder.test.queue</value></constructor-arg><!--
String destinationName -->
                <constructor-arg index="2"><value>true</value></constructor-
arg><!-- boolean isQueue -->
        </bean>

        <bean name="openMediationAlarmForwarder"
class="com.hp.uca.expert.alarm.OpenMediationAlarmForwarder">
                <constructor-arg index="0"><value>UCA-
EBC_remotesystem</value></constructor-arg><!-- String actionReference -->
                <constructor-arg index="1"><value>Alarm Flow from UCA
EBC</value></constructor-arg><!-- String alarmFlowName -->
        </bean>
</beans>
```

Figure 18 - Defining AlarmForwarder beans in the context.xml file

The highlighted portion of the `context.xml` file shows the definition of a FileAlarmForwarder bean that will be used in the rule files of a scenario to forward alarms to an XML file.

Once the `context.xml` file has been properly set up, you need to define global variable entries in the `ValuePackConfiguration.xml` file for each Spring bean that you want to access from the rules as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      name="__PROJECT_NAME__" version="__PROJECT_VERSION__">

  <scenarios>
      <scenario name="alarmforwarder">
```

```
<filterFile>src/main/resources/valuepack/alarmforwarder/filters.xml</filterFi
le>
        <fireAllRulesPolicy>EACH_ACCESS</fireAllRulesPolicy>
        <globals>
                <global>
                        <key>fileAlarmForwarder</key>
                        <value>fileAlarmForwarder</value>
                </global>
                <global>
                        <key>jmsAlarmForwarder</key>
                        <value>jmsAlarmForwarder</value>
                </global>
                <global>
                        <key>openMediationAlarmForwarder</key>
                         <value>openMediationAlarmForwarder</value>
                </global>
        </globals>
        <processingMode>STREAM</processingMode>
        <rulesFiles>
            <rulesFile>
        <filename>file:./src/main/resources/valuepack/alarmforwarder/alarmforwa
rder.drl</filename>
                    <name>alarmforwarder rules</name>
                    <ruleFileType>DRL</ruleFileType>
            </rulesFile>
        </rulesFiles>
         </scenario>
     …
    </scenarios>
</valuePackConfiguration>
```

Figure 19 - Defining AlarmForwarder globals in the ValuePackConfiguration.xml file

The highlighted portion of the `ValuePackConfiguration.xml` file shows the definition of a fileAlarmForwarder global variable referencing the fileAlarmForwarder Spring bean defined in the `context.xml` file that will be used in the rule files of a scenario to forward alarms to an XML file.

Once the `ValuePackConfiguration.xml` file has been properly set up, you need to make some modifications to the rule files where you want to use the fileAlarmForwarder global variable:

Import the proper Java class:

- `com.hp.uca.expert.alarm.FileAlarmForwarder` for a FileAlarmForwarder
- `com.hp.uca.expert.alarm.JMSAlarmForwarder` for a JMSAlarmForwarder
- `com.hp.uca.expert.alarm.OpenMediationAlarmForwarder` for an OpenMediationAlarmForwarder

Declare the global variables (defined in the `ValuePackConfiguration.xml` file) that you want to use in the rule file

Below is an example of how to import the proper Java class, and declare the global variables that you want to use:

```
package com.hp.uca.expert.vp.alarmforwarder;

#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmDeletion;
import com.hp.uca.expert.alarm.AlarmStateChange;
import com.hp.uca.expert.alarm.AlarmAttributeValueChange;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;
import java.util.ArrayList;
```

```
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;
import com.hp.uca.expert.flag.Flag;
import com.hp.uca.expert.testmaterial.AbstractJunitIntegrationTest;
import com.hp.uca.expert.alarm.FileAlarmForwarder;
import com.hp.uca.expert.alarm.JMSAlarmForwarder;
import com.hp.uca.expert.alarm.OpenMediationAlarmForwarder;

#declare any global variables here
global Scenario theScenario;
global FileAlarmForwarder fileAlarmForwarder;
global JMSAlarmForwarder jmsAlarmForwarder;
global OpenMediationAlarmForwarder openMediationAlarmForwarder;

declare Alarm
    @role( event )
    @timestamp( timeInMilliseconds )
    @expires( 30m )
end
```

Figure 20 - Declaring the use of an AlarmForwarder global variable in a rule file

Once the proper Java classes have been imported and the global variables declared, you can just use global variable to write Alarms (or collections of Alarms) to an XML file (the one specified in the `context.xml` file):

```
…
import com.hp.uca.expert.alarm.FileAlarmForwarder;
import com.hp.uca.expert.alarm.JMSAlarmForwarder;
import com.hp.uca.expert.alarm.OpenMediationAlarmForwarder;

#declare any global variables here
global Scenario theScenario;
global FileAlarmForwarder fileAlarmForwarder;
global JMSAlarmForwarder jmsAlarmForwarder;
global OpenMediationAlarmForwarder openMediationAlarmForwarder;

declare Alarm
    @role( event )
    @timestamp( timeInMilliseconds )
    @expires( 30m )
end


# Forward any alarm received
rule "Forward any alarm received"
no-loop
        when
         $alarm : Alarm()
        then
        LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

        // Forward the alarm to a file, jms queue/topic or OSS Open Mediation
        fileAlarmForwarder.write($alarm);
        // Forward the alarm to a jms queue or topic
        jmsAlarmForwarder.write($alarm);
        // Forward the alarm to OSS Open Mediation
        openMediationAlarmForwarder.write($alarm);

        // Retract the alarm
        theScenario.getLogger().info("Retracting: \n"+
$alarm.toFormattedString());
        theScenario.getSession().retract($alarm);

        LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());
end
…
```

Figure 21 - Using an AlarmForwarder global variable to write Alarms to an XML file

The XML file generated by the FileAlarmForwarder is fully compatible with the XML schema for UCA for EBC Alarms defined at `${UCA_EBC_DEV_HOME}/lib/schemas/uca-expert-alarm.xsd`. For example, the generated XML file containing the alarms can be used as input to the `${UCA_EBC_HOME}/bin/uca-ebc-injector` command-line tool.

The JMSAlarmForwarder on the other hand can be used to forward alarms directly to a JMS queue/topic, for example the Alarm input queue of a UCA for EBC server (which is implemented as a JMS Topic). You can use the following values to forward alarms to a UCA for EBC alarm input queue:

- **brokerURL**: JMSAlarmForwarder.DEFAULT_UCA_EBC_BROKER_URL (the value of this constant is "tcp://localhost:61666")
- **destinationName**: JMSAlarmForwarder.DEFAULT_UCA_EBC_ALARMS_TOPIC_NAME (the value of this constant is "com.hp.uca.ebc.alarms")
- **isQueue**: false (because the UCA for EBC alarm input queue is in fact a JMS topic, not a JMS queue)

Finally the OpenMediationAlarmForwarder can be used to forward alarms to OSS Open Mediation V7.2. In order to use an OpenMediationAlarmForwarder, you must first create an action reference in the `${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml` file that will define how to connect to the UCA for EBC Channel Adapter on OSS Open Mediation V7.2, and how to reach the Channel Adapter of the system/application that you target.

Below is an example of an action reference defined in the `ActionRegistry.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ActionRegistryXML xmlns="http://registry.action.mediation.uca.hp.com/">

        <MediationValuePack MvpName="ApplicationX" MvpVersion="1.1"

        url="http://localhost:26700/uca/mediation/action/ActionService?WSDL"
                brokerURL="failover://tcp://localhost:10000">

                <Action actionReference="ApplicationX_remotesystem">
                        <ServiceName>applicationX-1.1</ServiceName>
                        <NmsName>remotesystem</NmsName>
                </Action>
        </MediationValuePack>
</ActionRegistryXML>
```

In the sample `ActionRegistry.xml` file above, an action reference has been defined for an "ApplicationX" application on a remote system connected to OSS Open Mediation V7.2 via an ApplicationX Channel Adapter (ApplicationX is a fictitious application).

The brokerURL attribute must match the URL of the ActiveMQ broker defined for the OSS Open Mediation V7.2 that you target. The hostname in the URL must match the hostname of the system where OSS Open Mediation V7.2 is installed. By default the port number used for the ActiveMQ broker on OSS Open Mediation V7.2 container instance 0 is 10000.

To verify what port number is used for your OSS Open Mediation V7.2 container instance, please check the value of the activemq.port property in the `/var/opt/openmediation-V71/containers/instance-<instance number>/conf/servicemix.properties` file.

The following JMS properties will be set for the alarms being forwarded to OSS Open Mediation V7.2. These properties can be used by consumer Channel Adapters to filter the alarms that they're interested in among all alarms pushed by various Channel Adapters to the OSS Open Mediation V7.2 alarms JMS topic:

Table 8 - JMS properties set for alarms being forwarded to OSS Open Mediation

JMS Property Name	Value
NOMOriginalProvider	set to the value of ${ca.name} in UCA EBC CA
NOMOriginalProviderEndpoint	"UCA EBC version on hostname"
NOMOriginalProviderPort	not set
NOMOriginalProviderHost	set to the value of ${nom_hostname} in UCA EBC CA
NOMOriginalProviderContainerInstanceNumber	set to the value of ${sys.nom_instance_number} in UCA EBC CA
NOMType	set to "http://hp.com/openmediation/alarms/2011/08" in UCA EBC CA
NOMActionMessageType	not set (this is not an action message, this is an alarm message)
NOMActionEntityHint	not set (this is not an action message, this is an alarm message)
NOMActionNameHint	not set (this is not an action message, this is an alarm message)
NOMFinalConsumer	the value of the "serviceName" attribute of the action reference (in the ActionRegistry.xml file) associated with the OpenMediationAlarmForwarder object
NOMFinalConsumerEndpoint	"mvpName mvpVersion on nmsName", where the names in italics are XML entities/attributes of the action reference (in the ActionRegistry.xml file) associated with the OpenMediationAlarmForwarder object
NOMFinalConsumerPort	"alarmFlowName" associated with the OpenMediationAlarmForwarder object or "UCA EBC Alarms" by default. You can set the FlowName attribute when you create the OpenMediationAlarmForwarder object
NOMFinalConsumerHost	the value of the "nmsName" XML entity of the action reference (in the ActionRegistry.xml file) associated with the OpenMediationAlarmForwarder object
NOMFinalConsumerConstainerInstanceNumber	not set

4.4.4 Forwarding events through UMB

One of the roles of the value packs is to forward correlation results (whatever their types: Events, Trouble tickets, alarms...) to some other applications.

From a scenario this is done by using an UMBForwarder object that makes the link between the scenario and the UCA-EBC flow service as defined in the `AdapterConfiguration.xml` file.

An UMBEventForwarder object can be easily created by requesting its creation from the value pack's Spring context (`context.xml` in the valuepack configuration directory).

Here is an example of UMBEventForwarder creation:

```xml
<bean name="mediationEventForwarder"
class="com.hp.uca.expert.event.UMBEventForwarder">

  <constructor-arg index="0">
<value>UcaStaticEventForwarderFlow</value>
  </constructor-arg>
</bean>
```

Figure 22 - Defining mediationEventForwarder bean in the context.xml file

The UMBEventForwarder object is created with an argument which is the name of the static flow as it is define in the UCA-EBC `AdapterConfiguration.xml` file.

Then from a rule file, this UMBEventForwarder object can be used as follow:

1. Define the object in the rule file 'global section'
2. Use the UMBEventForwarder push() method to forward an event to the bus.

Example of rule forwarding an event to the bus:

```
package com.hp.uca.expert.vp.alarmforwarder;

#list any import classes here.
import com.hp.uca.expert.event.EventForwarder;
import com.hp.uca.expert.event.Event;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;
import com.hp.uca.expert.util.MessageFileHandler;
import java.util.ArrayList;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;
import com.hp.uca.expert.flag.Flag;
import com.hp.uca.expert.testmaterial.AbstractJunitIntegrationTest;


#declare any global variables here
global Scenario theScenario;
global EventForwarder mediationEventForwarder;

# Forward any event received
rule "Forward any event received"
no-loop
      when
          $event : Event()
      then
      LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

      // Forward the event to ne new Mediation
      mediationEventForwarder.push($event);

      // Retract the event
      theScenario.getLogger().info("Retracting: \n"+
$event.toFormattedString());
      theScenario.getSession().retract($event);

      LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());
end
```

Then the UCA adapter must declare itself as provider of a static flow with a name corresponding to the name passed as parameter of the declaration of the mediationEventsForwarder bean:

Defining static flows

For static Flows the collectorClass must be set to:
`com.hp.uca.expert.mediation.adapter.UcaStaticCollector`

No flow parameters need to be defined.

Here is an example of Static Flow Service definitions for UCA-EBC:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="UCA-EBC" version="1.0" xmlns="http://hp.com/umb/config">
    <flowServices>
        <flow name="UcaStaticEventForwarderFlow" type="Static"
collectorClass="com.hp.uca.expert.mediation.adapter.UcaStaticCollector">
        </flow>
    </flowServices>
</adapter>
```

And then any remote application can consume the events forwarded by the UCA VP by consuming the flow. For that the AdapterConfiguration.xml of the UMB mediation adapter of the remote application should look like this:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="myRemoteApp" version="1.0" xmlns="http://hp.com/umb/config">
    <autoConsumers>
        <autoConsumer consumerIdentifier="myRemoteApp"
targetAdapterName="UCA-EBC"
                      targetFlowName="UcaStaticEventForwarderFlow"

messageConsumerClass="com.acme.foo.umb.adapter.UcaEventConsumer"/>

    </autoConsumers>
</adapter>
```

4.5 Making useful logs

The UCA for EBC product provides an advanced logging mechanism that is able to trace specific rule processing for each Scenario.

The UCA for EBC Administration GUI fully supports this logging mechanism.

> NOTE: For more information on how to troubleshoot scenarios using the UCA for EBC Administration GUI, please see: [R7] Unified Correlation Analyzer for Event Based Correlation – User Interface Guide, chapter Troubleshooting UCA for event based Correlation

To take benefits from this mechanism, the rule developer must use the logger provided by the UCA for EBC framework for each scenario by calling the following method:

- *theScenario.getLogger()* from Drools files
- *ScenarioThreadLocal.getScenario().getLogger()* from Java code

The *ScenarioThreadLocal* class is located in the *com.hp.uca.expert.scenario* package.

The *getLogger()* method provides access to a standard *org.apache.log4j.Logger* object. Consequently, all standard log4j *Logger* methods are available to better qualify the level of information needed (for example *info()*, *debug()*, *warn()*, etc…).

The following piece of code demonstrates how to use the UCA for EBC scenario logger to log messages from a Drools rule file:

```
package com.hp.uca.expert.vp.sample;

#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.CustomFields;
import com.hp.uca.expert.x733alarm.CustomField;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;
import com.hp.uca.expert.x733alarm.NetworkState;          // NOT_CLEARED,
CLEARED
import com.hp.uca.expert.x733alarm.OperatorState;         // NOT_ACKNOWLEDGED,
ACKNOWLEDGED, TERMINATED
import com.hp.uca.expert.x733alarm.ProblemState;          // NOT_HANDLED,
HANDLED, CLOSED
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;


#declare any global variables here
global Scenario theScenario;


rule "Any new Acknowledged Alarm"
  when
    a: Alarm(operatorState == OperatorState.ACKNOWLEDGED)
  then
        LogHelper.enter(theScenario.getLogger(),
drools.getRule().getName());

        theScenario.getLogger().info("[RULE " + drools.getRule().getName() + "]
Found new acknowledged alarm: identifier = " + a.getIdentifier()+ ":");
        theScenario.getLogger().debug(a.toFormattedString());

        LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());
end

rule "Any new Terminated Alarm"
  when
      a: Alarm(operatorState == OperatorState.TERMINATED)
  then
        LogHelper.enter(theScenario.getLogger(),
drools.getRule().getName());

        theScenario.getLogger().info("[RULE " + drools.getRule().getName() + "]
Found new terminated alarm: identifier = " + a.getIdentifier() + ":");
        theScenario.getLogger().debug(a.toFormattedString());

        LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());
end
```

Figure 23 - Scenario logger example

NOTE: Please refer to *Chapter "Scenario Loggers"* in the [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide for more information on how to use Scenario Loggers.

4.6 Creating JUnit Tests

Developing Value Packs involves creating correlation rules and writing code. In any case, it is highly recommended to unit test your rules and code.

To help you in that regard, the 'skeleton' project (the project created with the UCA Eclipse plug-in) provides you with a template of a JUnit test (based on JUnit 4.11) along with the complete infrastructure to compile, run and generate reports for unit tests.

The following JUnit test is a good starting point to create new unit tests:

It is a JUnit 4.11 test that also supports Java and Spring framework annotations: using @RunWith and @Configuration annotations automatically loads the associated Spring configuration file (called `<test file name>-context.xml`)

The template JUnit test class that we provide extends the **AbstractJunitIntegrationTest** class. This class is part of the UCA for EBC framework. It implements the Spring framework **ApplicationContextAware** interface, and thus provides access to the Spring beans (Java objects) defined in the Spring configuration file(called `<test file name>-context.xml`). You can easily retrieve any Spring bean defined in the Spring configuration file by using the **getApplicationContext().getBean(String name)** method from any JUnit test class that extends the AbstractJunitIntegrationTest class.

In JUnit 4.11, any method that represents a unit test needs to have the @Test annotation before the definition of the method.

It is mandatory to define a **`junit.framework.Test suite)`** method so that tests can be found in the Apache Ant project of your Value Pack. Defining the following method allows for automatic retrieval of all tests defined in the unit test class:

```
// Way to run tests via ANT Junit
public static junit.framework.Test suite() {
  return new JUnit4TestAdapter(SkeletonTest.class);
}
```

4.6.1 Testing with alarms

When designing Junits, it is a good practice to test alarms expected lifecycle, using different **AlarmListener** assigned with the different alarms identifier to be tested, for:

- alarm insertion (**`waitingForAlarmInsertion`**)
- alarm update (**`waitingForAlarmUpdate`**)
- alarm retraction (**`waitingForAlarmRetract`**)

Other good practice is to test different objects values with **`assertEquals, assertNull, assertNotNull`** and others methods furnished with the Junit library. Also, the number of Groups in memory can be tested by comparing it with the result of calling the method **`getGroupsFromWorkingMemory()`** and the number of Alarms in memory by calling the **`getAlarmsFromWorkingMemory()`**.

By comparing the historical engine events with a benchmark, you can easily check the whole test result with the expected one.

In the following code you can find a template JUnit test class using some of the methods described above:

```
package com.hp.uca.expert.vp.skeleton;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import junit.framework.JUnit4TestAdapter;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```java
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.annotation.DirtiesContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.hp.uca.common.misc.Constants;
import com.hp.uca.common.trace.LogHelper;
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.testmaterial.AbstractJunitIntegrationTest;
import com.hp.uca.expert.x733alarm.OperatorState;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class SkeletonTest extends AbstractJunitIntegrationTest {

      private static Logger log =
LoggerFactory.getLogger(SkeletonTest.class);
      private static final String SCENARIO_BEAN_NAME = "skeleton";
      private static final String ALARM_FILE =
"src/main/resources/valuepack/skeleton/Alarms.xml";

      /**
       * @throws java.lang.Exception
       */
      @BeforeClass
      public static void setUpBeforeClass() throws Exception {
            log.info(Constants.TEST_START.val() +
SkeletonTest.class.getName());
      }

      /**
       * @throws java.lang.Exception
       */
      @AfterClass
      public static void tearDownAfterClass() throws Exception {
            log.info(Constants.TEST_END.val() + SkeletonTest.class.getName()
                        + Constants.GROUP_ALT1_SEPARATOR.val());
      }

      // Way to run tests via ANT Junit
      public static junit.framework.Test suite() {
            return new JUnit4TestAdapter(SkeletonTest.class);
      }

      @Test
      @DirtiesContext()
      public void test() throws Exception {
            LogHelper.enter(log, "test()");

            /*
             * Initialize variables and Enable engine internal logs
             */
            initTest(SCENARIO_BEAN_NAME, BMK_PATH);

            /*
             * Create,Assign and store an Alarm Listener to the current
scenario
             */
            setAlarmListener(createAndAssignAlarmListener("1",
getScenario()));

            /*
             * Send alarms defined in Alarms.xml asynchronously with a tempo
of 2
```

```java
             * seconds between each alarm
             */
            getProducer().sendAlarmsAsync(ALARM_FILE, 2 * SECOND);

            /*
             * Wait for an alarm insertion in scenario working memory
             */
            waitingForAlarmInsertion(getAlarmListener(), 100 * MS, 10 *
SECOND);
            /*
             * Retrieve from Working memory the Alarm with identifier '1'
             */
            Alarm alarm = getAlarm("1");

            /*
             * Check that alarm with identifier '1' exists
             */
            assertNotNull("The alarm 1 should be present in WM", alarm);

            /*
             * Wait for an alarm update in scenario working memory
             */
            waitingForAlarmUpdate(getAlarmListener(), 100 * MS, 10 * SECOND);
            /*
             * Check the new values of attributes 'problemInformation' &
             * 'notificationIdentifier' of alarm '1'
             */
            assertEquals(
                        "The problemInformation should be New Problem
information",
                        "New Problem information",
alarm.getProblemInformation());
            assertEquals("The notificationIdentifier should be equals to
100",
                        "100", alarm.getNotificationIdentifier());

            /*
             * Wait for an alarm acknowledgement
             */
            waitingForAlarmAcknowledgement(getAlarmListener(), 100 * MS,
                        10 * SECOND);
            /*
             * Check if the OperatorState of alarm has been correctly changed
to
             * ACKNOWLEDGED
             */
            assertEquals("Alarm 1 has been acknowledged",
                        OperatorState.ACKNOWLEDGED,
alarm.getOperatorState());

            /*
             * Wait for an alarm retraction from scenario working memory
             */
            waitingForAlarmRetract(getAlarmListener(), 100 * MS, 10 *
SECOND);

            /*
             * Disable Rule Logger to close the file used to compare engine
             * historical events
             */
            closeRuleLogFiles(getScenario());
            /*
             * Check test result by comparing the historical engine events
with a
             * benchmark
             */
```

```
            checkTestResult(getLogRuleFileName(), getLogRuleFileNameBmk());

            LogHelper.exit(log, "test()");
        }
}
```

4.6.2 Testing with events

If the Value Pack to be tested is expected to handle events, other than alarms, the JUnit tests will also be extending the **AbstractJunitIntegrationTest** class.

The difference between sending alarms and sending events (non-alarms) is illustrated below.

```
// Send alarms
getProducer().sendAlarms(ALARM_FILE);
```

```
// Send events listed in file EVENTS_FILE every second
getEventsProducer().sendEvents(EVENTS_FILE,1000);
```

The format of events listed in the EVENTS_FILE is described below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Events xmlns="http://hp.com/uca/expert/event">

  <EventBoxBase eventClassName="event1 Class name">
    <eventString><![CDATA[ event1 xml description  ]]>
    </eventString>
      </EventBoxBase>

    <EventBoxBase eventClassName=" event2 Class name ">
      <eventString> <![CDATA[event2 xml description  ]]>
       <eventString>
      </EventBoxBase>

 <EventBoxBase> …
 </EventBoxBase>

</Events>
```

Here is an example with a events belonging to a class of events named Temperature:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Events xmlns="http://hp.com/uca/expert/event">

    <EventBoxBase eventClassName="com.hp.uca.expert.demo.Temperature">
      <eventString><![CDATA[<temperature
xmlns:ns2="http://hp.com/uca/expert/event"
xmlns:ns3="http://hp.com/uca/expert/demo"
xmlns:ns4="http://hp.com/uca/expert/x733Alarm">
<ns2:identifier>100</ns2:identifier>
<ns2:eventTime>0</ns2:eventTime>
<ns2:targetValuePack>MyVP</ns2:targetValuePack>
<ns3:value>37.2</ns3:value>
</temperature>]]></eventString>
      </EventBoxBase>

    <EventBoxBase eventClassName="com.hp.uca.expert.demo.Temperature">
      <eventString><![CDATA[<temperature
xmlns:ns2="http://hp.com/uca/expert/event"
xmlns:ns3="http://hp.com/uca/expert/demo"
xmlns:ns4="http://hp.com/uca/expert/x733Alarm"><ns2:identifier>101</ns2:ident
```

```
ifier><ns2:eventTime>01</ns2:eventTime><ns2:targetValuePack>MyVP</ns2:targetV
aluePack><ns3:value>100</ns3:value></temperature>]]></eventString>

      </EventBoxBase>

</Events>
```

Note that alarm events could also be sent encapsulated in EventBoxBase containers.

The Spring configuration file associated with the events test file (called `<test file name>-context.xml`) must contain the following elements

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:amq="http://activemq.apache.org/schema/core"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd
                                http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd
                                http://www.springframework.org/schema/jms
                                http://www.springframework.org/schema/jms/spring-
jms.xsd
                                http://activemq.apache.org/schema/core
                                http://activemq.apache.org/schema/core/activemq-
core.xsd
                                http://www.springframework.org/schema/util
                                http://www.springframework.org/schema/util/spring-
util-2.5.xsd">

  <context:annotation-config />

  <!-- Import main application context -->
  <import resource="classpath:/main-context.xml"/>


  <bean id="vpBootstrap"
class="com.hp.uca.expert.testmaterial.ValuePackTestBootsrap">
             <property name="configurationFile"
value="src/test/resources/com/hp/uca/expert/ft/collector/event/ValuePackConfi
guration.xml" />
  </bean>

  <!-- JMS Producer Configuration -->
  <bean id="jmsProducerConnectionFactory"
         class="org.springframework.jms.connection.SingleConnectionFactory"
         depends-on="broker"
         p:targetConnectionFactory-ref="jmsFactory" />

  <bean id="jmsEventProducerTemplate"
class="org.springframework.jms.core.JmsTemplate"
         p:connectionFactory-ref="jmsProducerConnectionFactory"
         p:defaultDestination-ref="destinationEvent" />

<bean id="jmsEventProducer"
      class="com.hp.uca.expert.testmaterial.EventBoxMessageProducerSpring" >
             <property name="template" ref="jmsEventProducerTemplate"/>
  </bean>

  <!-- ActiveMQ Destination -->
```

```xml
  <amq:topic id="destinationEvent" physicalName="com.hp.uca.ebc.events" />

  <jms:listener-container container-type="default"
          destination-type="topic" connection-
factory="jmsConsumerConnectionFactory"
          acknowledge="auto">
          <jms:listener id="T-EventCollector"
destination="com.hp.uca.ebc.events"      ref="eventBoxMessagingListener" />
  </jms:listener-container>

  <bean id="eventBoxMessagingListener"
      class="com.hp.uca.expert.collector.event.EventCollector">
      <property name="helper" ref="eventHelper" />
  </bean>

  <bean id="eventHelper"
class="com.hp.uca.expert.event.marshal.EventMarshallingHelper">
  </bean>
</beans>
```

4.6.3 State Listener

Since V3.2, when using the topology extension for developing for value packs as well as for developing Inference Machine or Topology State Propagator Value Packs, another listener was introduced for testing States lifecycle: the **StateListener.** In the same way as for testing alarms, the StateListener can be used in Junits for checking:

- state insertion (**waitingForStateInsertion**)
- state update (**waitingForStateUpdate**)
- state retraction (**waitingForStateRetract**)

Also, as seen for Groups, the number of PropagationGroups in memory is given by the method:

```
getPropagationGroupsFromWorkingMemory();
```

In the following code you can find a template JUnit test class using the states checking:

```java
package ft.tsp;

import static org.junit.Assert.assertEquals;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

import junit.framework.JUnit4TestAdapter;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.neo4j.loader.csv.Loader;
import org.neo4j.loader.csv.Report;
import org.neo4j.loader.csv.utils.TmpDir;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.test.annotation.DirtiesContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.hp.uca.common.misc.Constants;
import com.hp.uca.expert.alarm.Alarm;
```

```java
import com.hp.uca.expert.group.PropagationGroup;
import com.hp.uca.expert.scenario.exception.NoSuchScenarioException;
import com.hp.uca.expert.testmaterial.AbstractJunitIntegrationTest;
import com.hp.uca.expert.testmaterial.ActionListener;
import com.hp.uca.expert.testmaterial.AlarmListener;
import com.hp.uca.expert.testmaterial.StateListener;
import com.hp.uca.expert.topology.TopoAccess;

/**
 * The Class PropagationTest.
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class PropagationTest extends AbstractJunitIntegrationTest {

    /**
     * The log.
     */
    private static Logger log =
LoggerFactory.getLogger(PropagationTest.class);

    /**
     * The Constant ALARM_FILE.
     */
    private static final String ALARM_FILE =
"src/test/resources/alarms/Alarm_SwitchDown_G_SWITCH_3_TeMIP.xml";

    /**
     * The Constant SCENARIO_BEAN_NAME.
     */
    private static final String SCENARIO_BEAN_NAME =
"com.hp.uca.expert.vp.im.TopologyStatePropagator";

    /**
     * The Constant TOPOLOGY_DATALOAD_DIR.
     */
    private static final String TOPOLOGY_DATALOAD_DIR =
"valuepack/topology-dataload";

    /**
     * The tmp dir.
     */
    private TmpDir tmpDir = null;

    /**
     * @throws java.lang.Exception
     */
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        log.info(Constants.TEST_START.val());
    }

    /**
     * @throws java.lang.Exception
     */
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        log.info(Constants.TEST_END.val());
    }

    /**
     * @throws java.lang.Exception
     */
    @Override
    @Before
    public void setUp() throws Exception {
        log.info(Constants.TEST_START.val() + this.getClass().getName());
```

```
            tmpDir = new TmpDir(TOPOLOGY_DATALOAD_DIR);
            Loader loader = new Loader(TopoAccess.getGraphDB(),
tmpDir.tmpCsvPath());
            Report report = loader.loadAll();

            log.info(report.toString());
        }

        /**
         * @throws java.lang.Exception
         */
        @Override
        @After
        public void tearDown() throws Exception {
            log.info(Constants.TEST_END.val() + this.getClass().getName()
                        + Constants.GROUP_ALT1_SEPARATOR.val());
            tmpDir.cleanup();
        }

        /**
         * Initialize variables and Enable engine internal logs
         */
        protected void initTest() throws NoSuchScenarioException,
                    InterruptedException {
            initTest(SCENARIO_BEAN_NAME, BMK_PATH);
            getScenario().setTestOnly(true);
        }

        /**
         * Way to run tests via ANT Junit
         *
         * @return the JUnit4TestAdapter
         */
        public static junit.framework.Test suite() {
            return new JUnit4TestAdapter(PropagationTest.class);
        }

        /**
         * @throws Exception
         */
        @Test
        @DirtiesContext
        public final void testGeneratedSvcAlarm() throws Exception {

            initTest();

            Map<String, String> keyValues = new HashMap<String, String>();
            keyValues.put("directiveName", "SET");
            keyValues
                        .put("entityName", "operation_context oc1
alarm_object 123456");

            ActionListener actionListener = new ActionListener(keyValues);

            getScenario().getSession().addEventListener(actionListener);

            AlarmListener alarmSwitchDownListener =
createAndAssignAlarmListener(
                        "UCA-1416582585978-61", getScenario());
            AlarmListener alarmSAListener = createAndAssignAlarmListener(
                        "operation_context oc1 alarm_object 123456",
getScenario());
            StateListener stateSwitch1 = createAndAssignStateListener(
                        "StateBase#G_SWITCH1", getScenario());
            StateListener stateWM1 = createAndAssignStateListener(
                        "StateBase#G_VM1", getScenario());
```

```
            StateListener statePoolA3 = createAndAssignStateListener(
                    "StateBase#G_poolA3", getScenario());

            /*
             * Send alarms
             */
            getProducer().sendAlarms(ALARM_FILE);

            waitingForAlarmInsertion(alarmSwitchDownListener, 1 * SECOND,
                    10 * SECOND);
            waitingForAlarmInsertion(alarmSAListener, 1 * SECOND, 10 *
SECOND);
            /*
             * Waiting for the last Alarm that should be updated by the rule
itself
             */
            waitingForActionInsertion(actionListener, 1 * SECOND, 15 *
SECOND);

            /*
             * Checking alarm updated
             */
            waitingForAlarmUpdate(alarmSAListener, 1 * SECOND, 10 * SECOND);
            waitingForAlarmUpdate(alarmSwitchDownListener, 1 * SECOND, 10 *
SECOND);
            /*
             * Checking states insertion
             */
            waitingForStateInsertion(stateSwitch1, 1 * SECOND, 20 * SECOND);
            waitingForStateInsertion(stateWM1, 1 * SECOND, 20 * SECOND);

            /*
             * check state update
             */
            waitingForStateUpdate(stateWM1, 1 * SECOND, 20 * SECOND);

            /*
             * checking last state insertion
             */
            waitingForStateInsertion(statePoolA3, 1 * SECOND, 20 * SECOND);

            /*
             * Disable Rule Logger to close the file used to compare engine
             * historical events
             */
            closeRuleLogFiles(getScenario());

            if (log.isDebugEnabled()) {
                    getScenario().getSession().dump();
            }

            /*
             * Checking Alarm Number: 1 RCA, 2 for service g_payroll + g_hr,
1 for
             * phone service, 1 for customer gardens
             */
            Collection<Alarm> alarms = getAlarmsFromWorkingMemory();
            assertEquals(5, alarms.size());

            /*
             * Checking Group Number
             */
            Collection<PropagationGroup> groups =
getPropagationGroupsFromWorkingMemory();
            assertEquals(15, groups.size());

    }
```

```
}
```

📄 NOTE: The **AbstractJunitIntegrationTest** test utility class has been developed and is provided as part of the UCA for EBC Development Kit. A JavaDoc documentation is provided for this class. Please refer to the Java Documentation of the com.hp.uca.expert.testmaterial package for full explanations.

Using the Apache Ant `build.xml` file provided in the example project (Skeleton) project (or projects created with the UCA eclipse plugin) allows you to automatically compile, run the tests and generate the test reports (using the "test" Ant target).

📄 NOTE: The build.xml Ant file on runs Test Classes that have a name ended by 'Test'. All other classes will not be executed when launching the 'test' target. It is therefore highly recommended to name all test classes with a name ending with 'Test.java'.

JUnit test reports in HTML format are available in the `target/reports/junitreport` folder of your Value Pack:

Figure 24 - JUnit tests results for your Value Pack

4.7 Injecting alarms to UCA for EBC: Alarm Collector

The Alarm Collector is the UCA for EBC internal component responsible for collecting alarms from outside UCA for EBC in order to feed them to the scenarios of the Value Packs deployed on UCA for EBC.

The Alarm Collector is implemented as a JMS Topic that is registered using JNDI so that other applications can get access to it to post alarms that will feed UCA for EBC Value Packs, as shown below:
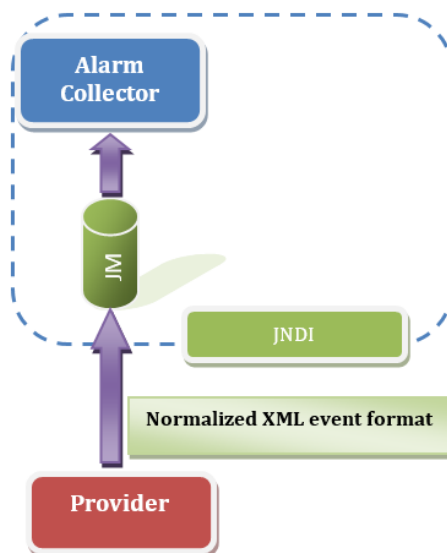
Figure 25 - UCA for EBC alarm collection

4.7.1 Normalized input

The UCA for EBC Alarm Collector defines a normalized alarm XML format based on the **X.733 standard alarm format**. Only alarms that comply with this format will be processed.

4.7.1.1 Sample alarms file

Below is a sample XML file that contains alarms in the X.733 alarm format:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<Alarms xmlns="http://hp.com/uca/expert/x733Alarm">
      <AlarmCreationInterface>
             <sourceIdentifier>src</sourceIdentifier>
             <identifier>1</identifier>
             <originatingManagedEntity>BOX B1</originatingManagedEntity>
             <alarmType>COMMUNICATIONS_ALARM</alarmType>
             <probableCause>Fire</probableCause>
             <perceivedSeverity>MINOR</perceivedSeverity>
             <alarmRaisedTime>2009-09-16T12:00:00.000+02:00</alarmRaisedTime>
      </AlarmCreationInterface>
      <AlarmCreationInterface>
             <sourceIdentifier>src</sourceIdentifier>
             <identifier>2</identifier>
             <originatingManagedEntity>BOX B1</originatingManagedEntity>
             <alarmType>COMMUNICATIONS_ALARM</alarmType>
             <probableCause>Fire</probableCause>
             <perceivedSeverity>CLEAR</perceivedSeverity>
             <alarmRaisedTime>2009-09-16T12:00:00.000+02:00</alarmRaisedTime>
      </AlarmCreationInterface>
</Alarms>
```

4.7.2 Command-line injector tool

UCA for EBC provides a tool to send alarms described in a simple XML File containing X.733 alarms to the UCA for EBC Alarm Collector.

This tool is located in the `${UCA_EBC_HOME}/bin` folder. It is called **uca-ebc-injector**.

This tool will inject alarms contained in an XML `file` into the input alarm queue (implemented as a JMS Topic) of a local or remote UCA for EBC Server instance.

Some samples of such an XML file containing alarms to be fed to UCA for EBC are located in the `${UCA_EBC_DEV_HOME}/vp-examples` folder.

> 📄 NOTE: For more information on the uca-ebc-injector command-line tool, please refer to the [R3] Unified Correlation Analyzer for Event Based Correlation – Administration, Configuration and Troubleshooting Guide

4.7.3 A sample Java Alarm injector

The following chapters describe how you can create your own sample Java Alarm injector application that can connect to UCA for EBC Alarm Collector JMS Topic to post Alarms to UCA for EBC.

4.7.3.1 Initializing the JNDI initial context

In order to create a sample Java Alarm injector, you must first initialize the JNDI context that will be used to retrieve the JMS Topic of the UCA for EBC Alarm Collector:

```
Context jndiContext = null;
/*
* Create a JNDI API InitialContext object
*/
try {
        jndiContext = new InitialContext();
    } catch (NamingException e) {
        System.out.println("Could not create JNDI API context: " +
e.toString());
        System.exit(1);
}
```

Please note that the `jndi.properties` file must be provided in the classpath of your sample Java Alarm injector.

4.7.3.2 Configuring the jndi.properties file

Here is the content of a sample `jndi.properties` file to be used by your sample Java Alarm injector:

```
java.naming.factory.initial =
org.apache.activemq.jndi.ActiveMQInitialContextFactory
topic.uca-ebc-alarms = com.hp.uca.ebc.alarms


# use the following property to configure the default connector
java.naming.provider.url =tcp\://localhost\:61666
```

The **topic.uca-ebc-alarms** property is used to record the name the UCA for EBC Alarm Collector JMS topic: **com.hp.uca.ebc.alarms**.

The **java.naming.provider.url** property can be configured to match the hostname and port number of UCA for EBC JNDI service.

4.7.3.3 Looking up the UCA for EBC Alarm Collector JMS topic

Once the JNDI context is initialized, the codes in your sample Java Alarm injector shall first lookup for the JNDI connection factory, and then retrieve the UCA for EBC Alarm Collector JMS topic by looking up its name:

```
ConnectionFactory connectionFactory = null;
     Destination destination = null;
     /*
     * Look up connection factory and destination.
     */
     try {
       connectionFactory = (ConnectionFactory) jndiContext
                .lookup("ConnectionFactory");
       destination = (Destination) jndiContext.lookup("uca-ebc-alarms");
     } catch (NamingException e) {
       System.out.println("JNDI API lookup failed: " + e);
   System.exit(1) ;
}
```

4.7.3.4 Connect and send the message

With the connectionFactory retrieved, you then need to create the connection, then the session, and finally the producer:

```
Connection connection = null;
     MessageProducer producer = null;

     try {
   connection = connectionFactory.createConnection();
   session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
   producer = session.createProducer(destination);
   TextMessage message = session.createTextMessage();

   StringBuffer buf = new StringBuffer();
   buf.append("<?xml version=\"1.0\" encoding=\"UTF-8\"
standalone=\"yes\"?>");
   buf.append("<Alarms>");
   buf.append("<AlarmCreationInterface>");
   buf.append("<sourceIdentifier>src</sourceIdentifier>");
   buf.append("<identifier>12301</identifier>");
   buf.
append("<originatingManagedEntityClass>BOX</originatingManagedEntityClass>");
   buf.append("<originatingManagedEntity>BOX B1</originatingManagedEntity>");
   buf.append("<alarmType>COMMUNICATIONS_ALARM</alarmType>");
   buf.append("<probableCause>Fire</probableCause>");
   buf.append("<perceivedSeverity>MAJOR</perceivedSeverity>");
   buf.append("<alarmRaisedTime>2009-09-
16T12:00:00.000+02:00</alarmRaisedTime>");
   buf.append("</AlarmCreationInterface>");
   buf.append("</Alarms>");
   message.setText(buf.toString());
   System.out.println("Sending message: " + message.getText());
   producer.send(message);
} catch (JMSException e) {
   System.out.println("Exception occurred: " + e);
} finally {
if (connection != null) {
   try {
     connection.close();
   } catch (JMSException e) {
   }
}
}
```

By now you should have a functioning sample Java Alarm injector.

4.8 Injecting events to UCA for EBC: Event Collector

The Event Collector is the UCA for EBC internal component responsible for collecting events from outside UCA for EBC in order to feed them to the scenarios of the Value Packs deployed on UCA for EBC.

The Event Collector is implemented as a JMS Topic that is registered using JNDI so that other applications can get access to it to post events that will feed UCA for EBC Value Packs, as shown in Figure 25.

For more information about the CLI used to send events to UCA for EBC,
please refer to [R3] `HP UCA for Event Based Correlation – Administration, Configuration and Troubleshooting Guide`
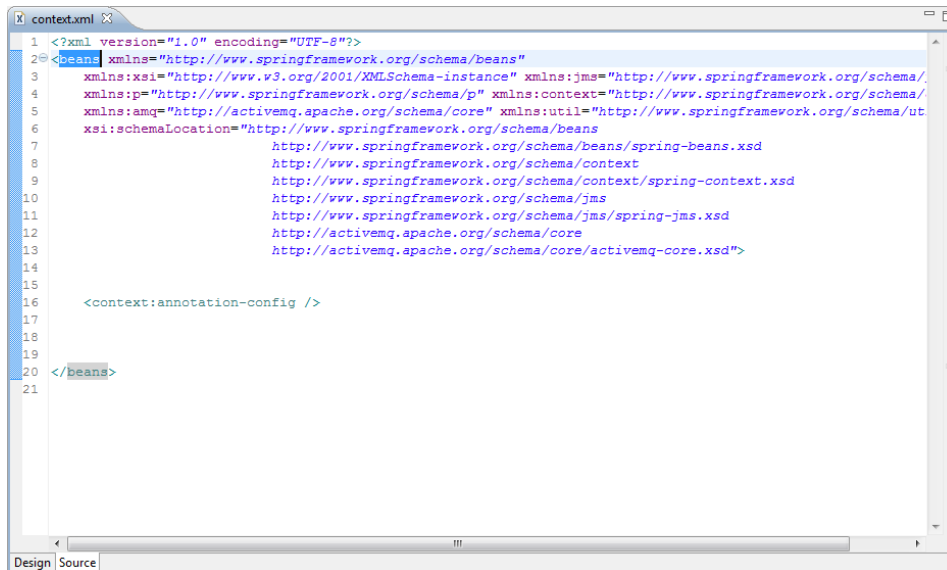
Chapter 5 Advanced Development features

5.1 Spring Framework integration

A Spring Framework `context.xml` file is provided in the `src/main/resources/valuepack/conf` folder. This file is defined for the whole "skeleton" value pack, i.e. it is common for all scenarios of the value pack.

All the Spring beans defined in this file will be available to each rule file of each scenario of the value pack.

By default the `context.xml` file is empty:

Figure 26 - **The default project's empty context.xml file**

You can define any number of Spring beans in the `context.xml` file. These beans will be accessible from within the rules files through global variables defined in your rules files provided you follow the instructions explained in the following sections.

5.1.1 Defining and using Spring Beans inside rule files using global variables

The Spring "dependency injection" framework is useful for defining global variables (already initialized) in rules files. In a normal Drools environment, this is done through some Java code. As UCA hides the Drools session object, global variables are "injected" with Spring, from a XML definition (`context.xml`).
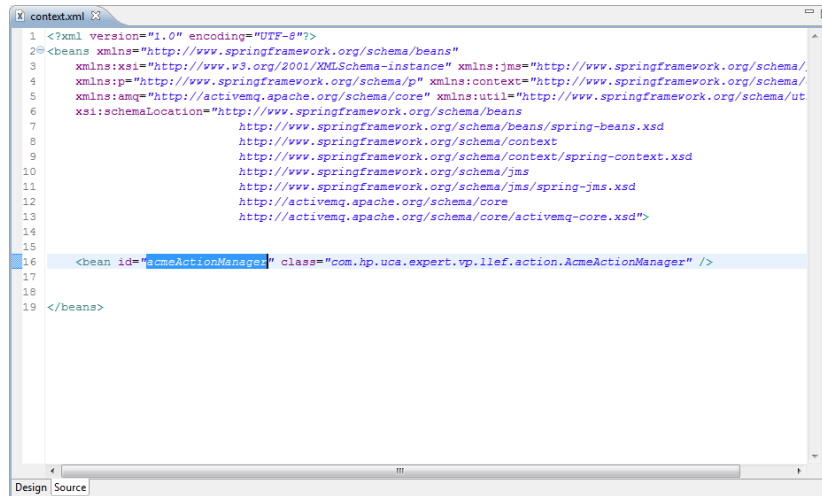
📄 NOTE: It is worth noting that there are 2 `context.xml` files in each value pack:

- In the `src/main/resources/valuepack/conf` folder is the `context.xml` that is used when the value pack runs on a UCA EBC Server instance
- In the `src/test/resources/<scenario folder name>` folder is the `<scenario name>-context.xml` that is used when the value pack runs in JUnit test mode.

Please make sure to define all your Spring beans in both files, otherwise the JUnit tests might fail.

First you need to define your Spring beans in the `context.xml` file (the following sample file comes from the Low Level Event Filtering value pack and is described in the *"UCA for EBC Value Packs Examples"* guide)

The Spring beans that you define in the *context.xml* file are defined at the Value Pack level, and thus are global to all scenarios of the Value Pack:

Figure 27 - **The "Low Level Event Filtering" Value Pack's context.xml file**

In the above screenshot, we define a Spring bean called `acmeActionManager`. This is just an example; with any other Spring bean, the process explained in the following paragraphs would have been the same.

Next we need to associate the Spring beans with global variables defined in your scenario. This is done in the `ValuePackConfiguration.xml` file that defines the configuration for all the scenarios of your value pack.

> NOTE: Although Spring beans are defined at the Value Pack level, global variables are defined at the scenario level. If you need a Spring bean to be global to all scenarios of your Value Pack, you need to configure the Spring bean as a global variable for each scenario of the Value Pack in the `ValuePackConfiguration.xml` file.

Figure 28 - Defining global variables in the ValuePackConfiguration.xml file

When you define global variables in the `ValuePackConfiguration.xml` file, the "key" has to match the name of the global variable you are defining (the name you choose must match the name of the global variable that you declare in your rules file(s)), and the "value" has to match the name of the bean defined in the `context.xml` file.

The last step is to define a global variable for the Spring bean in your rules file:

Figure 29 - Defining global variables in rules files
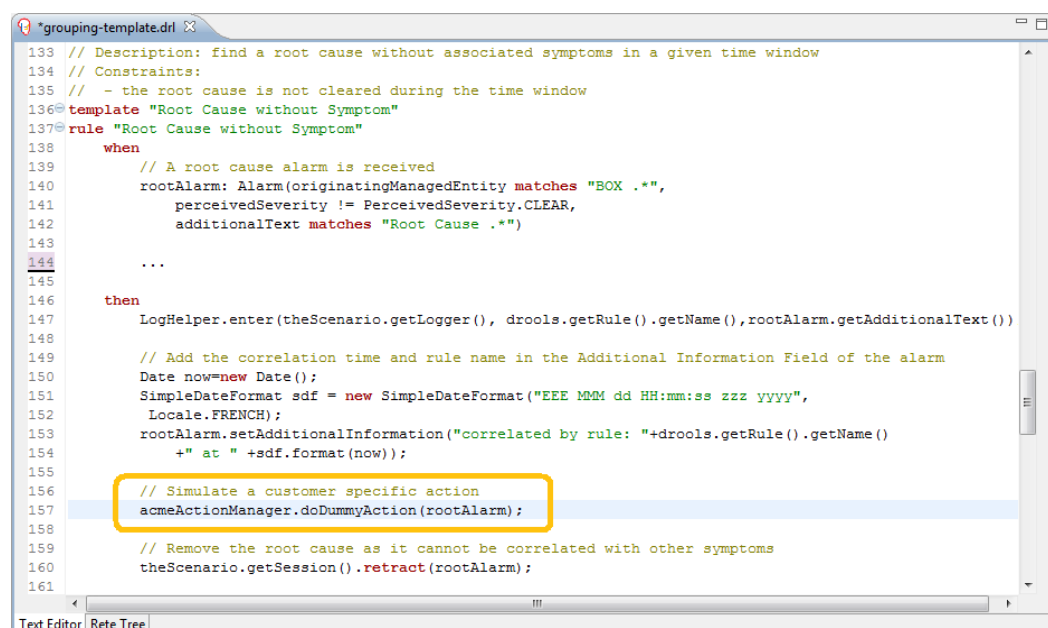
In the import section of your rules file, you need to add an "*import*" statement for the Java class of your Spring Bean:

```
import com.hp.uca.expert.vp.llef.action.AcmeActionManager;
```

Then you need to add a "global" statement creating a global variable for your Spring Bean:

```
global AcmeActionManager acmeActionManager;
```

Then you can use the global variable in your rules:

Figure 30 - Using global variables in rules files

5.2 Using the Flag Object

The UCA for EBC product provides a set of Flag Java object. These objects are useful to trigger rule execution in complex use cases or to trigger internal processing (Synchronization, etc...).

Please refer to [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide *Chapter Common Objects*, S*ection Flags*, for more information on how to use the Flag Object.

5.3 Alarm CustomFields

Alarm CustomFields is the standard x733alarm.CustomFields object. CustomFields attributes can be used in the rules "condition" part, whereas CustomFields methods can be called in the rules "action" part.

Please refer to [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide *Chapter Common Objects*, S*ection Alarm models used in the rules,* for more information on how to use the Custom Fields Object.

5.4 Alarm Raised Time

The AlarmRaisedTime field of an Alarm is using the Java type XMLGregorianCalendar, not easy to set. Hence, UCA for EBC provides a helper to set the AlarmRaisedTime field:

```
setTimeInMillisecond()
```

That sets all the time related fields.

Please refer to [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide, Chapter 5.1.1.2 General Attributes of Alarm for more information on how to deal with time fields.

5.5 Scenario specific configuration

The UCA for EBC provides a way to manage complex configuration based on XML file when the Customer Value Pack needs a complex specific configuration.

Please refer to [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide for more information on how to use the Specific Configuration, *Chapter Advanced UCA for EBC features, section Scenario Specific Configuration.*

5.6 Performing initialization at scenario startup

The UCA for EBC provides a way to initialize your Value Pack if it needs specific objects to be created at startup time. This is performed be defining a Java class in your Value Pack and setting it correctly in the configuration file.

Please refer to [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide for more information on how to perform initialization of customer object needed by a Value Pack.

5.7 WUI extensions for value packs

Since version 3.1 the UCA for EBC Web User Interface can be extended to host value pack's specific web applications or global web application

5.7.1 Extending the WUI at value pack Level

Any .war file delivered within value pack directory tree (usually in lib subdirectory) will be loaded through the UCA for EBC web server and visible through the Web User Interface.

When the value pack is started, the UCA for EBC Web UI makes this web application available from a new tab if the value packs' monitoring panel.

Example: the war file `MyVWebApp.war` dropped in `deploy/uca-topo-demo-3.2/lib` directory will lead to:
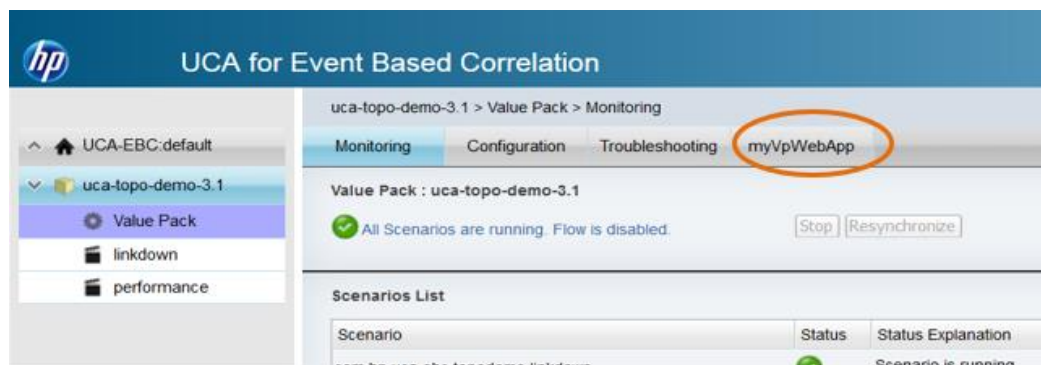
By default the UCA for EBC server binds the value pack web application at the following address:

http://localhost:8888/fullValuepackName-warFilename

For the example above this would give:

http://localhost:8888/uca-topo-demo-3.2-myVpWebApp

5.7.2 Extending the WUI at Global Level

In some cases the WUI extension is not directly linked to a specific value pack but may cover several value packs or a functionality global to the platform.

In such case it is useful to access this webapp from the global level (UCA-EBC). This is the role of the 'Extras' Submenu.

The 'Extras' sub-menu is displayed when you have optionally put some extra .war files under the $UCA_EBC_INSTANCE/webapps directory (**note the name of the directory with an 's' at the end**). This directory is optional and is not created by default.

Each .war file stored in this directory will be displayed by UCA for EBC UI under the following menu:

```
UCA-EBC:instanceName > Extras > <name of .war file>
```

As in the picture below:
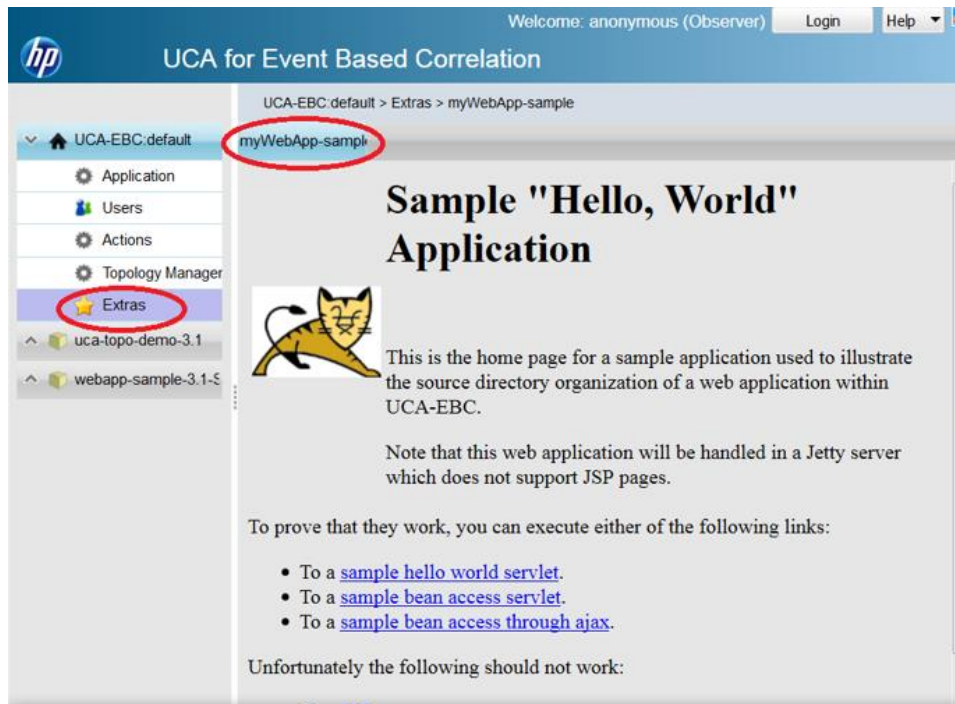
5.7.3 Web application extensions configuration

Some web application extensions may require some additional configuration in order for the UCA for EBC Web User Interface to build the expected URL.

Two possible configurations are offered:

- Defining the URL service Path
- Defining URL service parameters

5.7.3.1 Defining the URL service Path for extensions at value pack level

This is done by adding a property in the `uca-ebc.properties` file as shown below:

```
ValuepackFullname-warFileName-webapp-servicepath=your_path
```

Example:

For the value pack: uca-topo-demo (version3.2) with a war file named `myWebApp.war`, please define:

```
uca-topo-demo-3.2-myVpWebApp-webapp-servicepath=myService
```

This will lead to building the following URL:

http://localhost:8888/uca-topo-demo-3.2-myVpWebApp/myService

5.7.3.2 Defining the URL service Path for extensions at global level

This is done by adding a property in the `uca-ebc.properties` file as shown below:

```
warFileName-webapp-servicepath=your_path
```

Example:

For the war file named `myWebApp-sample.war`, please define:

```
myWebApp-sample-webapp-servicepath=myService
```

This will lead to building the following URL:

http://localhost:8888/myWebApp-sample/myService

5.7.3.3 Defining the URL parameters for extensions at value pack level

This is done by adding a property in the `uca-ebc.properties` file as shown below:

```
ValuepackFullname-warFileName-webapp-parameters= coma separated list of
parameters
```

Example:

For the value pack: uca-topo-demo (version3.2) with a war file named `myWebApp.war`, please define:

```
uca-topo-demo-3.2-myVpWebApp-webapp-parameters=param1=value1,param2=value2
```

This will lead to building the following URL:

http://localhost:8888/uca-topo-demo-3.2-myVpWebApp/?param1=value1&param2=value2#

5.7.3.4 Defining the URL parameters for extensions at global level

This is done by adding a property in the `uca-ebc.properties` file as shown below:

```
warFileName-webapp-parameters= coma separated list of parameters
```

Example:

For the war file named `myWebApp-sample.war`, please define:

```
myWebApp-sample-webapp-parameters=param1=value1,param2=value2
```

This will lead to building the following URL:

http://localhost:8888/myWebApp-sample/?param1=value1&param2=value2#

5.7.4 Inheriting the UCA for EBC logged user and role in the extended web application

Some web application may want to know which UCA user is logged (as well as his associated role) in order to adapt its processing depending on the user id or the role.

This is done by using placeholders in URL parameters as follow:

- `${user}` will represent the current logged user
- `${role}` will represent this user's role.

A typical definition would be:

```
uca-topo-demo-3.2-myVpWebApp-webapp-servicepath=
username=${user},userrole=${role}
```

5.8 Configuring the GUI filter tags editor

If your Value Pack is processing specific filters tags, it is possible to list them in a configuration so that the WUI will use that file to propose only those tags to be used for defining filters.

Please refer to [R2] Unified Correlation Analyzer for Event Based Correlation - Reference Guide and [R7] *UCA for Event Based Correlation – User Interface Guide* for more information on how to perform configuration to enable the GUI tags editor feature.

5.9 Editing Filter Files with the UCA for EBC eclipse filter editor

The UCA-EBC Development Toolkit provides a specific filter editor intended to ease the development of UCA-EBC filters.

This tool is mainly a checking tool that allows testing the filter against a sample of alarms. As a result the tool gives for each alarm, which Top-filter it passes or not, and if it passes a Top-filter, gives the associated tags (if any).

5.9.1 Editing a Filter

The UCA-EBC filter editor is available by right clicking on the Filter file as follow:
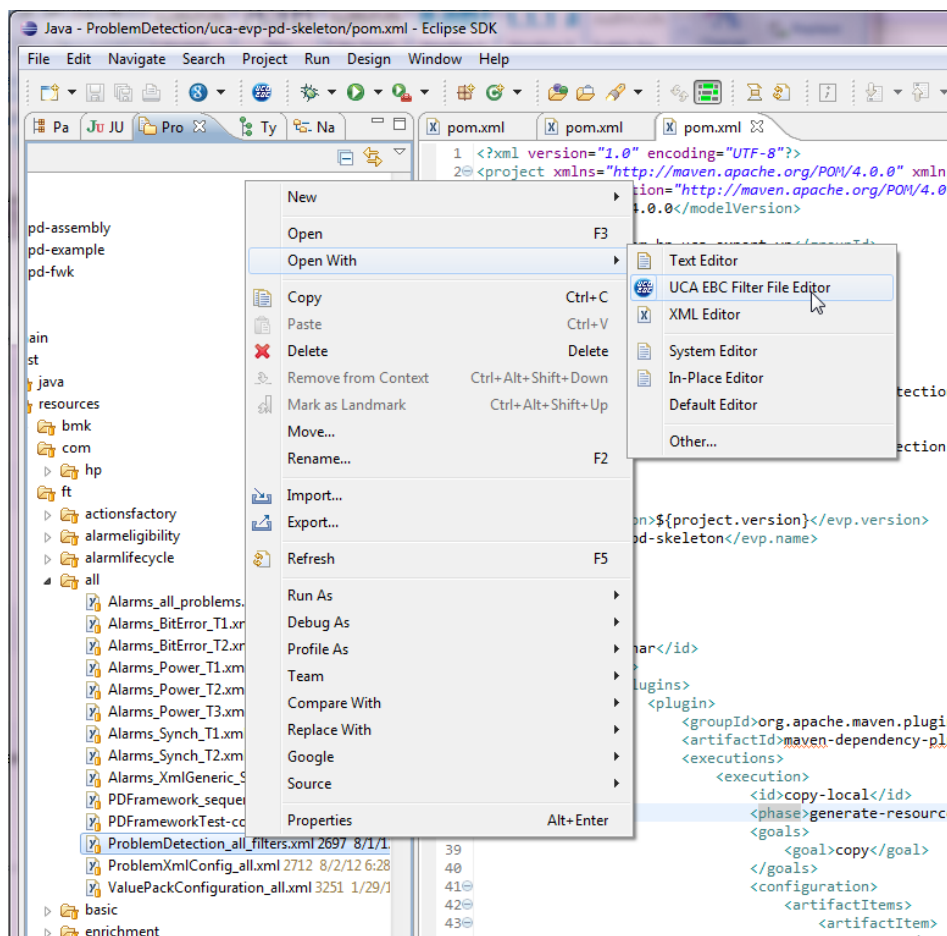
This launches the UCA-EBC filter editor.

At this stage the editor contains a single editor tab which is an XML editor allowing to edit/save the Xml Filter file:
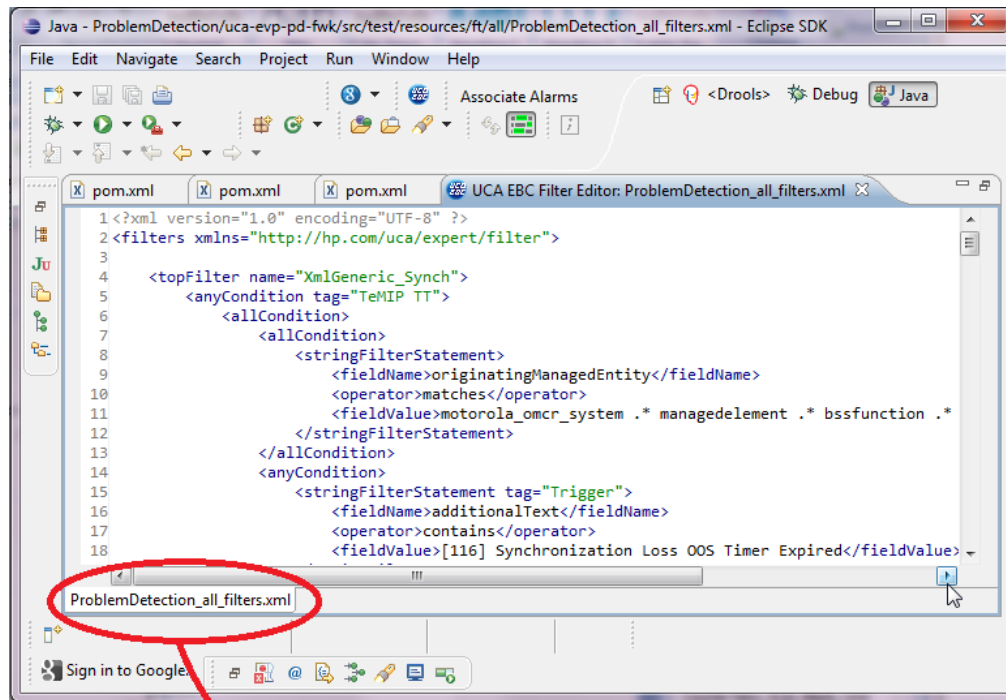
Single Tab Editor

5.9.2 Associating an Alarm File Sample to the Filter Editor

In Order to check the Filter against a set of alarms, the Xml Alarm file must be associated to the filter editor. This is done by left clicking on the Alarm File in order to select the file and the click on the 'Associate Alarms' button as follow:
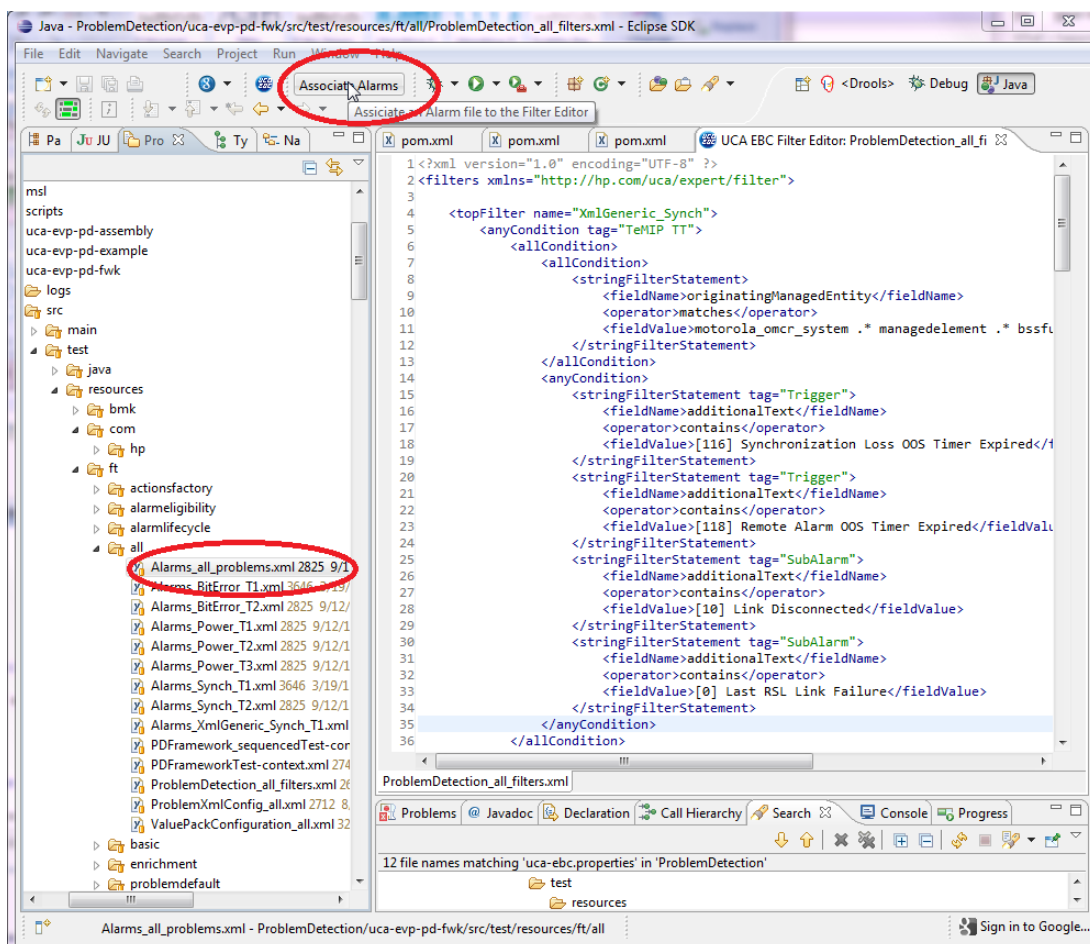
When the association is done, the editor turns itself into a multi-panel editor offering several edition panels:

- The Filter file editor panel, allowing to edit the Filter file
- The Aggregated View panel, giving an overview of the passing/blocked alarms
- The Alarm file editor panel, allowing to edit the Alarm File
- The Passed filter view, giving information on passed filters and tags.

As shown in the picture below:
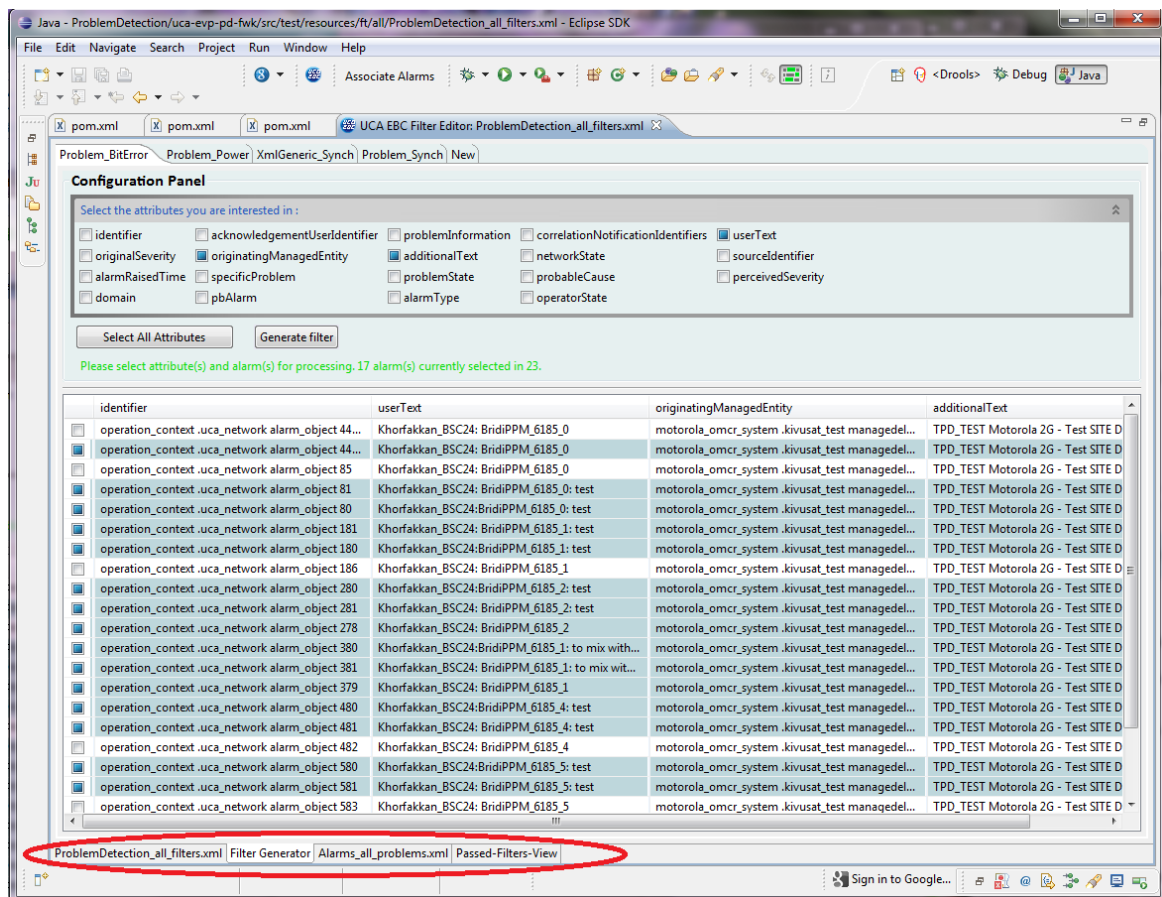
5.9.3 How to read the Filter editor aggregated view?

This view offers a panel per top filter as defined in the filter file.

You can switch from one top-filter to others by clicking on the top level panel selection:
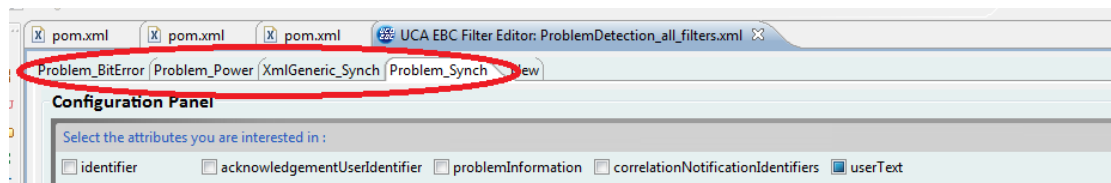
The configuration Panel area allows selecting the alarms attributes to be displayed in the Alarm table list.
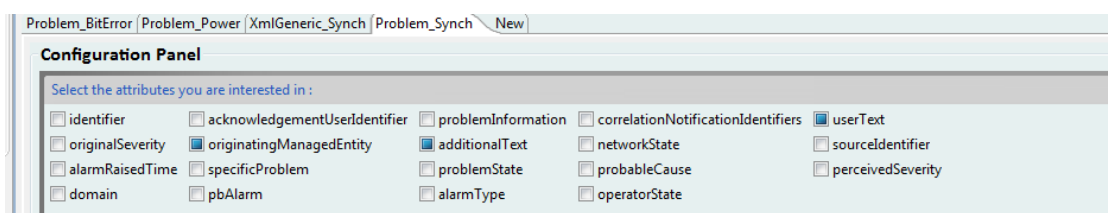
The Alarm table list shows the content of the alarm file as a table. Each table row is preceded by a check box indicating if the alarm is passing or not the given top-filter (A checked box and a green color indicate the alarm is passing the filter)

5.9.4 **How to read the 'passed filter' view?**

For a selected alarm, the 'passed filter' view gives the list of passed top-filters and the corresponding filter tags.

The passed filter view is a 3 parts window:

- The top part is the alarm picker, it allows selecting the alarm
- The left part displays the selected alarm content
- The right part gives the 'passed' top-filters and associated Tags.

5.9.5 How to use the filter to create a new top-filter?

The aggregate view offers the possibility to quickly create a new top-filter.

A top filter creation is a multi-step operation:

Step 1:   Create a new top-filter tab. This is done by clicking on the 'New' tab in the top-filter selection area:
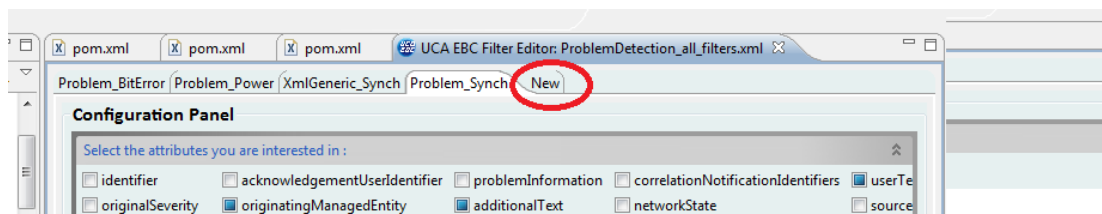
This creates a new Filter panel with a default name. This name can be changed by right clicking on the new filter tab:
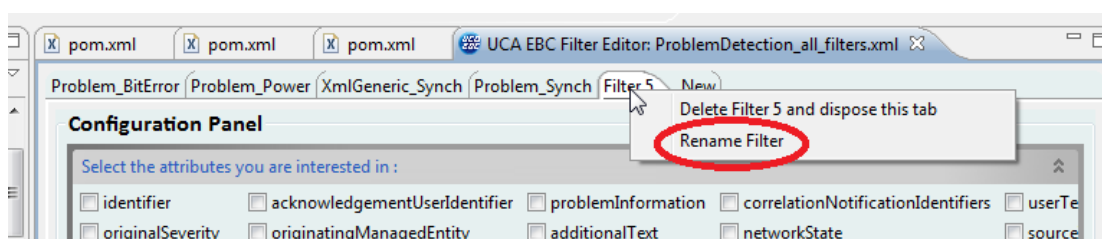
📝   NOTE: a Top-filter can also be deleted by clicking on the 'delete' option of the same menu.

Step 2:   select the alarm attributes that will play a role in the filtering in the "Configuration panel" section.

Example:
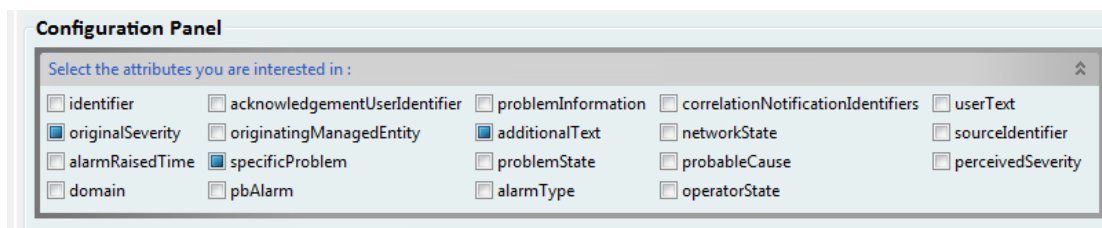
Step 3:   In the Alarm table, select those alarms that will pass the filter by selecting the checkbox.

Step 4:   generate the new filter by clicking the "Generate Filter" button.

Step 5: Click on the filter editor view and check the generated filter. You can manually edit the generated editor in order to make some fine tuning or changes.

Step 6: Control the result of the new filter in the "passed Filter" view

Step 7: save your changes

⚠️ CAUTION: The "Generate filter" Button can be used on an already existing filter in order to modify it. However by re-generating an existing filter, all the Tags defined in it will be lost. It is therefore not recommended to use the "Generate filter" button on existing filters.

5.10 Persisting alarms or events using the DB forwarder feature

This chapter provides technical information about the DB forwarder feature introduced in UCA-EBC 3.1.

It is intended to the UCA-EBC Value Pack developer that needs to set up that functionality within his VP.

Any DB coming with a JDBC driver can be supported by this feature. However, UCA-EBC brings 2 DBs with libraries already part of the UCA-EBC default libraries: H2 and HyperSQL.

5.10.1 Concepts

5.10.1.1 Storing alarms

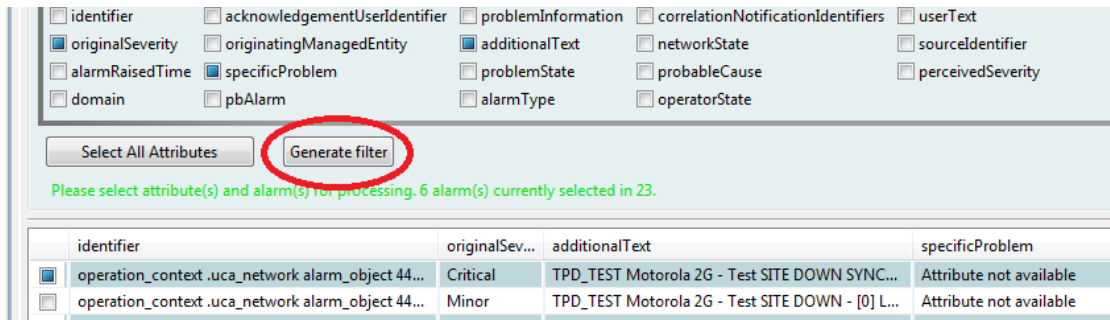To store alarms into a DB, the well-known alarm forwarder mechanism is used. In this particular case, a JDBC alarm forwarder is now provided to perform such actions.

Alarms that are stored into a DB follow also the same scheme of the alarms received through the mediation layer. Once stored in the DB, they are pushed back into the dispatcher of the Value Pack using the DB flow mechanism.

So if you want to recognize them from standard alarms, you will have to define a way to do it. This can done using a special identifier for the alarm, or by using a special custom field.

This is up to the Value Pack owner to decide which method is to be used.

5.10.1.2 Storing events

UCA-EBC (since version 3.2) brings new EventForwarder interface to handle Event objects (introduced in version 3.1 as well).

- `com.hp.uca.expert.event.EventForwarder`
- `com.hp.uca.expert.event.Event`

To store such Event objects into a DB, end-user can use a JDBC event forwarder based on the same concepts as the alarm forwarder described above.

- `com.hp.uca.expert.event.JDBCEventForwarder`

In the contrary of alarms, events stored into a DB do not have DB flow mechanism associated into it.

5.10.2 Getting started

To make use of the DB feature, this is just a question of configuring correctly your value pack. This is done by modifying the VP `context.xml file`[1].

Firstly, in this file, you will have to make use of the default JDBC settings by importing the provided file from the UCA classpath, as:

```
<import resource="classpath:jdbc/dependencies.xml" />
```

Those default settings bring mainly an AlarmDao bean (called *alarmDao*) and an AlarmNotifier bean (called *dbNotifier*). If you do not want to use default JDBC settings, you can do so by referring to the Advanced settings section below.

Then, still in context.xml, you will have to define at minimum 2 Spring beans:

- the datasource bean
- the DB forwarder bean

and optionally

- the DB store bean

5.10.2.1 Defining the datasource

The first thing to configure is the datasource. This is done by defining a new Spring bean. Spring offers a number of options for configuring a data sources via data source beans.

These sources include the following:

- Data sources that use JNDI
- Data sources that use JDBC drivers
- Data sources that pool connections

Below is an example using pool connections with Apache Commons DBCP[2], and with a H2 database[3].

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
      <property name="driverClassName" value="org.h2.Driver" />
      <property name="url" value="jdbc:h2:~/.uca/exampleDB" />
      <property name="username" value="sa" />
```

---

[1] You can also configure JDBC settings globally for all value packs in the conf/dependencies.xml file if needed.

[2] You could also use *"org.springframework.jdbc.datasource.DriverManagerDataSource"* or other of your choice

[3] You could also use HyperSQL DB. For other DBs, make sure to make the requested JDBC driver as part of your value pack libraries.

```
            <property name="password" value="" />
</bean>
```

5.10.2.2 Defining the DB store

The second thing to configure is the store used to persist alarms. Currently only a store of type SQL is supported. But still, in prevision of managing NOSQL stores, a bean is to be defined for specifying what that store is capable of. This setting is optional. The settable properties of an SQL store are:

Table 9 - Settable properties of an SQL store

Property	Type	Description	Defaults
name	string	Defines the name of the DB	
supportsCreate	boolean	tells if the DB can be created by the UCA-EBC engine if it does not exists	true except for "voltdb"
supportsIfNotExist	boolean	tells if the DB supports the SQL syntax "IF NOT EXISTS" at creation	true except for "hsqldb"
supportsUnlimitedVarChar	boolean	tells if the DB supports definition of VARCHAR without a numeric limit	true except for "vertica"
bigInt	string	defines the data type to use for big integers	"BIGINT" except for "oracle" that is "NUMBER"
useIndex	boolean	tells whether or not to create indexes at DB creation	true

Here below is a simple example:

```
<bean id="dbStore" class="com.hp.uca.expert.store.sql.SqlStore">
        <property name="name" value="h2" />
</bean>
```

5.10.2.3 Defining the DB forwarder

The next thing to configure is the DB forwarder itself, which is the thread that is going to use datasource and store defined previously to persist alarms.

The DB forwarder has 2 main properties: `alarmDao` and `store`. All possible properties are listed below:

Table 10 – Properties of a DB forwarder

Property	Type	Description
alarmDao	bean	the DB Alarm DAO bean
store	bean	the DB store bean
override	boolean	tells what to do when inserting an alarm that already exists in DB store (with same identifier). If false (default value): the new alarm is ignored. If true : the old alarm is deleted, the new alarm is inserted.
compress	boolean	tells whether or not to compress enqueued alarms with same identifier, for performance reasons. If true (default value): alarms are compressed.

Here below the typical configuration (The init-method is optional as the DB forwarder has an auto-start capability).

```
<bean id="dbForwarder" class="com.hp.uca.expert.alarm.JDBCAlarmForwarder"
init-method="start">
        <property name="alarmDao" ref="alarmDao" />
        <property name="store" ref="dbStore" />
</bean>
```

If you use a DB forwarder to forward Events instead of Alarms, you will need to configure as per example below (the eventDao bean needs to be configured too, as specified in Advanced settings section below).

```
<bean id="dbForwarder" class="com.hp.uca.expert.event.JDBCEventForwarder"
init-method="start">
        <property name="eventDao" ref="eventDao" />
        <property name="store" ref="dbStore" />
</bean>
```

5.10.2.4 Defining the DB flow

To be able to receive alarms changes coming from the DB as per any other alarm coming from a mediation flow, you will have to configure a DB flow in the `ValuePackConfiguration.xml` file.

The dbFlow has 2 main properties: `name` and `dbNotifierName`. All possible properties are listed below:

Table 11 – Properties of a DB flow

Property	Type	Description
name	string	the name of the DB flow. should be unique in case of multiple flows
dbNotifierName	string	refers to the name of the DB notifier on which to subscribe for notifications. This is explained in Advanced Setting section. Its default name is "dbNotifier".
automaticStart	boolean	flag indicating whether to automatically start the DB flow when the value pack is started or not. Default=true
lastEventReceivedFirst DuringResynchronization	boolean	attribute which tells if the DB notifier will notify existing alarms in reverse order (if true) upon resynchronization
eligibilityScope	string	element that specifies a Java evaluated boolean expression defining the eligibility of an alarm to pass through at flow resynchronization. default is "true" meaning all alarms present in DB are sent
sourceIdentifier	string	when alarm is coming through that flow, the sourceIdentifier is replaced by this value. default="DB
selfFeed	boolean	flag indicating whether to dispatch alarm creation messages generated by this value pack in standard mode (non-resynchronization. Default=false

A default configuration could be:

```
<dbFlows>
        <dbFlow name="exampleDbFlow" dbNotifierName="dbNotifier" />
</dbFlows>
```

5.10.3 Example

You can refer to the example part of the UCA-EBC Development Toolkit.

You can find it under `${UCA_EBC_DEV_HOME}/vp-examples/persistence-example`.

You can build this example as usual:

```
# ant all
```

Specifically, you can have a look at the files in the `src/main/resources/valuepack/conf` folder to see how to configure the DB feature elements (`context.xml`) and the DB flows (`ValuePackConfiguration.xml`)

5.10.4 Advanced settings

Advanced settings are optional and are only for those who do not want to use the default settings provided by the `jdbc/dependencies.xml` file. You can replace the following line:

```
<import resource="classpath:jdbc/dependencies.xml" />
```

And add each of the following beans directly in the `context.xml` file of the value pack.

5.10.4.1 Defining the SQL Session factory

The SQL session factory is the MyBatis[1] session factory bean. It has two properties:

Table 12 – Properties of the MyBatis SQL session factory bean

Property	Type	Description
dataSource	bean	the datasource bean
configLocation	string	the location of the MyBatis configuration file

The default configuration is:

```
<bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="configLocation" value="classpath:jdbc/mybatis-
config.xml"/>
</bean>
```

5.10.4.2 Defining the DB Alarm DAO

The DB DAO is the mapper interface used to instantiate the Java interface corresponding to the SQL commands stored in the file defined within the MyBatis configuration file. By default, the alarms mapper interface is defined in the `jdbc/sql-alarms-mapper.xml` file.

The DB DAO has two properties:

Table 13 – Properties of the DB Alarm DAO

Property	Type	Description
sqlSessionFactory	bean	the SQL session factory bean

---

[1] MyBatis is an Open Source software delivered as part of UCA-EBC 3.2 libraries.

`mapperInterface`	string	the Java interface for the DAO, which is defaulted to the one provided by UCA-EBC, i.e. `com.hp.uca.expert.alarm.store.AlarmDao`

The DB DAO is in turn used to configure the DB forwarder and the DB notifier beans.

The default configuration is:

```
<bean id="alarmDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
        <property name="sqlSessionFactory" ref="sqlSessionFactory" />
        <property name="mapperInterface"
value="com.hp.uca.expert.alarm.store.AlarmDao" />
</bean>
```

5.10.4.3 Defining the DB Notifier

The DB notifier is a component that listens to the DB for changes and notifies the value pack about these changes by feeding the DBFlow.

The DB Notifier has two properties:

Table 14 – Properties of the DB Notifier

Property	Type	Description
`alarmDao`	bean	the DB Alarm DAO bean
`checkTimer`	number	a timer in milliseconds representing the interval between two DB checkings for the changes

The default configuration is:

```
<bean id="dbNotifier" class="com.hp.uca.expert.alarm.store.AlarmNotifier"
scope="singleton">
        <property name="alarmDao" ref="alarmDao" />
        <property name="checkTimer" value="1000" />
</bean>
```

5.10.4.4 Defining the DB Event DAO

The DB Event DAO is the mapper interface used to instantiate the Java interface corresponding to the SQL commands stored in the file defined within the MyBatis configuration file. By default, the events mapper interface is defined in the `jdbc/sql-events-mapper.xml` file.

Table 15 – Properties of the DB Event DAO

Property	Type	Description
`sqlSessionFactory`	bean	the SQL session factory bean
`mapperInterface`	string	the Java interface for the DAO, which is defaulted to the one provided by UCA-EBC, i.e. `com.hp.uca.expert.event.store.EventDao`

The DB Event DAO is in turn used to configure the DB forwarder bean.

There is no default configuration available but it should be easily configurable as per below:

```
<bean id="eventDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
        <property name="sqlSessionFactory" ref="sqlSessionFactory" />
```

```
        <property name="mapperInterface"
value="com.hp.uca.expert.event.store.EventDao" />
</bean>
```

5.10.4.5 Defining the SQL Mapping interfaces

Alarms mapper:

The alarms mapper interface is defined by default in the `jdbc/sql-alarms-mapper.xml` file. This file defines the dynamic SQL mapping of the Java interface provided:
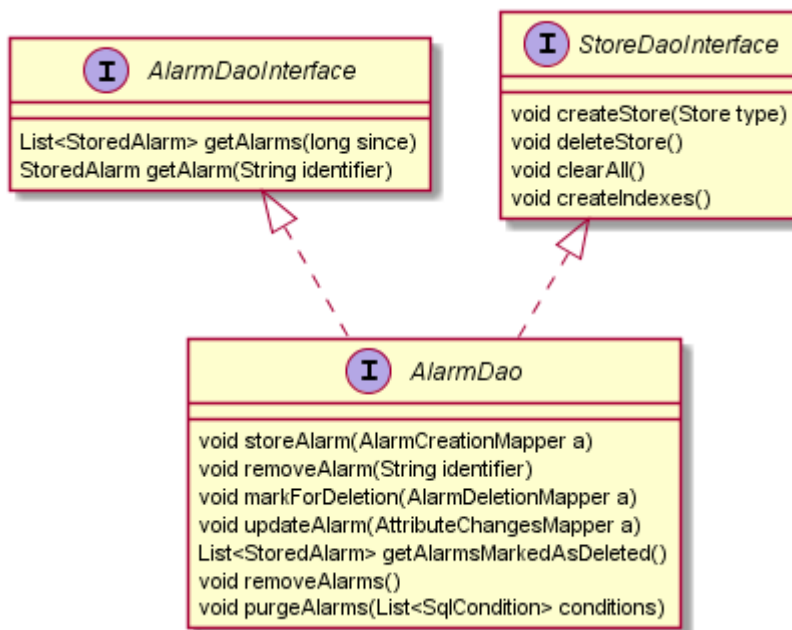

Figure 31 – AlarmDao Java interface

This interface is provided by default and can be replaced if necessary, in which case the mapping interface should be changed accordingly.

Events mapper:

The events mapper interface is defined by default in `jdbc/sql-events-mapper.xml` file. This file defines the dynamic SQL mapping of the Java interface provided:
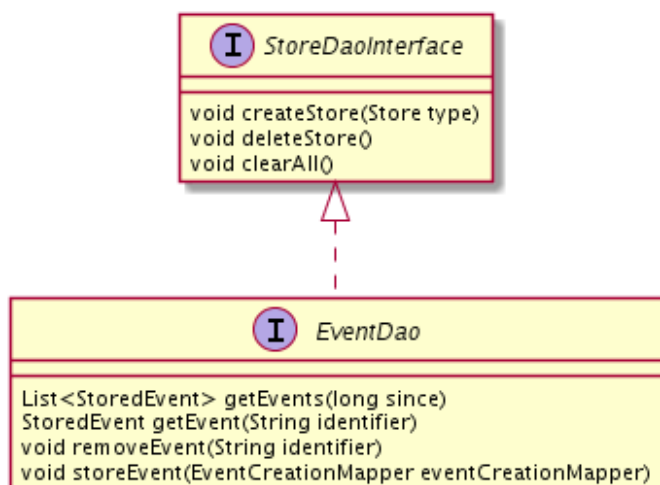

Figure 32 – EventDao Java interface

This interface is provided by default and can be replaced if necessary, in which case the mapping interface should be changed accordingly.

Appendix A
Ant build.xml targets

The value pack examples provided with UCA for EBC come with an Ant `build.xml` file that can build and package the project as described in this document.

Following is the full list of Apache Ant targets defined in the `build.xml` file that can be executed from the command line using the **ant** tool:

eclipse command:

```
# ant eclipse
```

Creates the .project and .classpath files used by eclipse when importing a project.

clean command:

```
# ant clean
```

Removes all files created during the build from the build directory.

compile command:

```
# ant compile
```

Compiles all Java files of the project.

test command:

```
# ant test
```

Runs the Junit tests defined in the project.

package command:

```
# ant package
```

Build the final, "ready to deploy" value pack ZIP file.

all command:

```
# ant all
```

Is equivalent to executing the following targets: "clean", "compile", "test" and "package".

Appendix B
Glossary

Table 16 - Glossary

Acronym	Definition
DRL	Drools Rule file
EBC	Event Based Correlation
EVP	UCA for EBC Value Pack
IDE	Integrated Development Environment
Inference Engine	Process that uses a Rete algorithm
JMS	Java Messaging Service
JMX	Java Management eXtension, used to access or process action on the UCA for EBC product
JNDI	Java Naming and Directory Interface
UCA	Unified Correlation Analyzer
XML	Extensible Markup Language
XSD	Schema of an XML file, describing its structure
X733	Standard describing the structure of an Alarm used in telecommunication environment
WUI	Web User Interface