PROGRESS Corticon

Corticon Studio: Extensions Guide

PROGRESS

Notices

Table of Contents

Preface

For details, see the following topics:

Progress Corticon documentation

The following documentation, as well as a *What's New in Corticon* document, is included with this Progress Corticon release:

Corticon Tutorials	
Corticon Studio Tutorial: Basic Rule Modeling	Introduces modeling, analyzing, and testing rules and decisions in Corticon Studio. Recommended for evaluators and users getting started. *See also the PowerPoint-as-PDF version of this document that is accessed from the Studio's* ***Help*** *menu.*
Corticon Studio Tutorial: Advanced Rule Modeling	Provides a deeper look into Corticon Studio's capabilities by defining and testing vocabularies, scope, collections, messages, filters, conditions, transient data, and calculations in multiple rulesheets that are assembled into a Ruleflow. *See also the PowerPoint-as-PDF version of this document that is accessed from the Studio's* ***Help*** *menu.*

Corticon Tutorial: Using Enterprise Data Connector (EDC)	Introduces Corticon's direct database access with a detailed walkthrough from development in Studio to deployment on Server. Uses Microsoft SQL Server to demonstrate database read-only and read-update functions.
Corticon Studio Documentation: Defining and Modeling Business Rules	
Corticon Studio: Installation Guide	Step-by-step procedures for installing Corticon Studio on computers running Microsoft Windows as a standalone installation and as a part of an existing Eclipse installation such as Progress Developer Studio for OpenEdge. Shows how to enable internationalization on Windows.
Corticon Studio: Rule Modeling Guide	Presents the concepts and purposes the Corticon Vocabulary, then shows how to work with it in Rulesheets by using scope, filters, conditions, collections, and calculations. Discusses chaining, looping, dependencies, filters and preconditions in rules. Presents the Enterprise Data Connector from a rules viewpoint, and then shows how database queries work. Provides information on versioning, natural language, reporting, and localizing. Provides troubleshooting and many *Test Yourself* exercises.
Corticon Studio: Quick Reference Guide	Reference guide to the Corticon Studio user interface and its mechanics, including descriptions of all menu options, buttons, and actions.
Corticon Studio: Rule Language Guide	Reference information for all operators available in the Corticon Studio Vocabulary. A Rulesheet example is provided for many of the operators. Includes special syntax issues, handling arithmetic and character precedence issues.
Corticon Studio: Extensions Guide	Detailed technical information about the Corticon extension framework for extended operators and service call-outs. Describes several types of operator extensions, and how to create a custom extension plug-in.
Corticon Server Documentation: Deploying Rules as Decision Services	
Corticon Server: Deploying Web Services with Java	Details installing the Corticon Server as a Web Services Server, and then deploying and exposing Decision Services as Web Services on the Pacific Application Server (PAS) and other Java-based servers. Includes SOAP and JSON/RESTful samples. Presents the features and functions of the browser-based Server Console. Provides administrative instructions for the Pacific Application Server.

Corticon Server: Deploying Web Services with .NET	Details installing the Corticon Server as a Web Services Server, and then deploying and exposing decisions as Web Services with .NET. Provides installation and configuration information for the .NET Framework and Internet Information Services (IIS) on various supported Windows platforms.
Corticon Server: Integration & Deployment Guide	An in-depth, technical description of Corticon Server deployment methods, including preparation and deployment of Decision Services and Service Contracts through the Deployment Console tool. Discusses relational database concepts and implementation of the Enterprise Data Connector. Goes deep into the server to discuss state, persistence, and invocations by version or effective date. Includes server monitoring techniques, performance diagnostics, and recommendations for performance tuning.

Overview of Progress Corticon

Progress® Corticon® is the Business Rules Management System with the patented "no-coding" rules engine that automates sophisticated decision processes.

Progress Corticon products

Progress Corticon distinguishes its development toolsets from its server deployment environments.

- **Corticon Studio** is the Windows-based development environment for creating and testing business rules:
  - When installed as a standalone application, Corticon Studio provides the complete Eclipse development environment for Corticon as the **Corticon Designer** perspective. You can use this fresh Eclipse installation as the basis for adding other Eclipse toolsets.
  - When installed into an existing Eclipse such as the **Progress Developer Studio** (PDS), our industry-standard Eclipse and Java development environment, the PDS enables development of Corticon applications in the **Corticon Designer** perspective that integrate with other products, such as Progress OpenEdge.

  Note: Corticon installers are available for 64-bit and 32-bit platforms. Typically, you use the 64-bit installer on a 64-bit machine, where that installer is not valid on a 32-bit machine. When adding Corticon to an existing Eclipse, the target Eclipse must be an installation of the same bit width. Refer to the *Corticon Studio: Installation Guide* to access, prepare, and install Corticon Studio.

  Studio Licensing - Corticon embeds a time-delimited evaluation license that enables development of both rule modeling and Enterprise Data Connector (EDC) projects, as well as testing of the projects in an embedded Axis test server. You must obtain Studio development licenses from your Progress representative.

- **Corticon Servers** implement web services for business rules defined in Corticon Studios:

- **Corticon Server for deploying web services with Java** is supported on various application servers, and client web browsers. After installation on a supported Windows platform, that server installation's deployment artifacts can be redeployed on various UNIX and Linux web service platforms as Corticon Decision Services. The guide *Corticon Server: Deploying web services with Java* provides details on the full set of platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage.
- **Corticon Server for deploying web services with .NET** facilitates deployment of Corticon Decision Services on Windows .NET Framework and Microsoft Internet Information Services (IIS). The guide *Corticon Server: Deploying web services with .NET* provides details on the platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage.

Server Licensing - Corticon embeds a time-delimited evaluation license that enables evaluation and testing of rule modeling projects on supported platform configurations. You must obtain server deployment licenses and server licenses that enable the Enterprise Data Connector (EDC) from your Progress representative.

1

Supported configurations

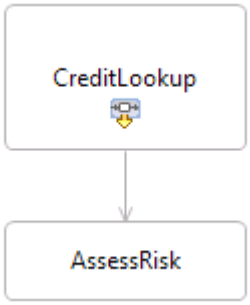
Software Component	Certified on Version
JDK	JDK 1.7.0_05
Eclipse (32-bit and 64-bit)	4.3.1
EMF	2.9.1
GEF	3.9.1 RC 1
GMF	1.7.0

2

Overview

You can extend the set of operators available to your users by writing your own Java classes.

Extension	Description	Example Invocation
Attribute Extended Operator	Extensions that you invoke against attributes of a given type. For example, you can define an extension that can be invoked against String attributes.	`Customer.zip.characterAt(2)` String Attribute Extended Operator `characterAt` returns a String consisting of the second character of `Customer.zip`.
Collection Extended Operator	Extensions that operate on collections. For example, you can define an extension that can be invoked against a collection of String attributes.	`Order.uni= orditm.sku->uniqueCount` Collection Extended Operator `uniqueCount` returns a count of the number of unique String values in collection `orditem.sku`.

Extension	Description	Example Invocation
Sequence Extended Operator	Extensions that operate on sequences. For example, you can define an extension that can be invoked against a sequence Decimal attributes.	`Trade.mavg= security.price->sortedBy(marketDate) ->mavg(30)` Sequence Extended Operator `mavg` returns the *moving average* of a sequence of price values in `security.price`.
Stand-Alone Extended Operator	Function you can use in any Rulesheet expression. A Stand-Alone extension can take any number of input parameters returns a value.	`Shape.circumference = SeMath.getCircumference (Shape.radius)` Stand-Alone Extension `SeMath.getCircumference` converts `Shape.radius` to `Shape.circumference`.
Service Call-out Extension	Extensions you can use in a Ruleflow. A Service Call-out is an extension that can directly access and update the universe of rule engine facts via the Service Call-out application programming interfaces.	 Service Call-out Extension `CreditLookup` retrieves credit data from a consumer credit agency and updates rule engine facts accordingly. The system forwards the updated facts to Rulesheet `AssessRisk` for analysis.

There are several ways to add your extension classes to the system. The approach you use depends on whether you are using Corticon in the Progress Developer Studio (Eclipse), or the Foundation APIs in a non-Eclipse setting.

3

Java class conventions

Your extension classes must conform to certain standards. Although you are free to choose any package and class names for your extensions, your classes must implement special *marker interfaces* to identify them as containers of extension logic:

Interface Name `com.corticon.services.extensions.`	Description
`ICcStringExtension`	Identifies your class as a container of String Attribute Extended Operators.
`ICcDateTimeExtension`	Identifies your class as a container of Date, Time and/or DateTime Attribute Extended Operators.
`ICcDecimalExtension`	Identifies your class as a container of Decimal Attribute Extended Operators.
`ICcIntegerExtension`	Identifies your class as a container of Integer Attribute Extended Operators.
`ICcCollectionExtension`	Identifies your class as a container of Collection Extended Operators.
`ICcSequenceExtension`	Identifies your class as a container of Sequence Extended Operators.

Interface Name com.corticon.services.extensions.	Description
ICcStandAloneExtension	Identifies your class as a container of Stand-Alone Extended Operators
ICcServiceCalloutExtension	Identifies your class as a container of Service Call-out Extensions.

These marker interfaces can be found in:

Usage Scenario	Location
Eclipse	Eclipse plug-in com.corticon.services
Foundation API	Corticon_eFoundation_API.jar

Corticon Studio is installed with source code examples that illustrate the proper implementation for each type of extension.

Refer to the source code examples in the Extended Operators and Service Call-outs folders.

4

Attribute extended operators

Attribute Extended Operators apply to specific attribute data types. Consider this example Rulesheet expression:

```
Customer.fullName = Customer.name.trimSpaces
```

Assuming that `Customer.name` is a String attribute, Attribute Extended Operator `trimSpaces` might perform the "trim" function on the value of attribute `name`, returning the trimmed value, which the rule engine will assign to `Customer.fullname`.

To create an Attribute Extended Operator, you must code a Java class, and your class must implement the marker interface corresponding to the attribute type. For example, to create a String Attribute Extended Operator, your Java class must implement interface `ICcStringExtension`. Example:

```
package com.acme.extensions;

import com.corticon.services.extensions.ICcStringExtension;

public class S1String implements ICcStringExtension
{
   public static String trimSpaces(String astrThis)
   {
      if (astrThis == null)
      {
         return null;
      }
      return astrThis.trim();
   }
}
```

Attribute Extended Operator methods must be declared `public static`.

The first positional parameter will always be a reference to the attribute upon which the function is being performed. In this example, the rules engine will pass the current value of `Customer.name` to extension method `trimSpaces` as parameter `astrThis`; thus, `astrThis` must be declared as type String.

An Attribute Extended Operator may accept any number of additional parameters as needed. Consider the following Rulesheet expression:

```
Customer.initial = Customer.name.characterAt(1)
```

In this example, your Java method would return the first character of `Customer.name`:

```
public static String characterAt(String astrThis, BigInteger abiIndex)
{
   if (abiIndex == null)
   {
      return null;
   }

   int liIndex = abiIndex.intValue() - 1;

   if (liIndex < 0 || liIndex >= astrThis.length())
   {
      return null;
   }

   return astrThis.substring(liIndex, liIndex + 1);
}
```

As always, the first positional parameter `astrThis` will contain a reference to the String upon which to operate (i.e., a reference to the value of `Customer.name`). The second parameter will contain the character index (i.e., literal `1`).

Your extension would return the character at the specified index as a String, and the rules engine will assign `Customer.initial` the value of your returned String.

For Attribute Extended Operators, you must limit your parameter and return types to the following:

Java Type	Corticon Type
java.math.BigInteger	Integer
java.math.BigDecimal	Decimal
java.lang.Boolean	Boolean
java.util.Date	Date, Time or DateTime
java.lang.String	String

Note that you may code any number of extension methods in a Java class, and you can supply one or more Java classes; regardless, the system will discover all of your methods via Java reflection.

5

Collection extended operators

Collection Extended Operators process a collection of input attribute values and return a single value. Consider this example expression:

```
ord.asteriskFlag = orditem.sku->allContain('*')
```

Assuming that `asteriskFlag` is a Boolean attribute and `orditem.sku` refers to a collection of String attributes, Collection Extended Operator `allContain` will return a Boolean value true if all instances of String attribute `sku` contain an asterisk.

To create a Collection Extended Operator, you must code a Java class, and your class must implement marker interface `ICcCollectionExtension`. Example:

```
package com.corticon.operations.extended.examples.operators.set1;

import java.util.HashSet;

import com.corticon.services.extensions.ICcCollectionExtension;

public class S1Collection implements ICcCollectionExtension
{
   public static Boolean allContain(String[] astrInputArray, String astrLookFor)

   {
      if (astrInputArray == null || astrInputArray.length == 0)
         return null;

      for (String lstrInput : astrInputArray)
      {
         if (lstrInput != null)
         {
            if (lstrInput.indexOf(astrLookFor) < 0)
            {
               return new Boolean(false);
            }
         }
      }

      return new Boolean(true);
   }
}
```

Collection Extended Operator methods must be declared `public static`.

In this example, the first positional parameter of method `allContain` is an array of String instances. The rules engine will populate this array with the String values in `orditem.sku` (i.e., the collection upon which the method operates).

The example Java method analyzes this array and returns a Boolean value. The rules engine will then assign `ord.asteriskFlag` the Boolean return value.

Note that you cannot code Collection Extended Operators for entity instances, nor can you code extensions for Collections or Sequences of entity instances. For example, the following expression is **not** allowed:

```
orditem->allContain('*')
```

In other words, you can only code Collection Extended Operators to process collections of *attribute* values.

When implementing Collection Extended Operators, you must limit your parameter and return types to the following:

Java Type	Corticon Type
`java.math.BigInteger`	Integer
`java.math.BigDecimal`	Decimal
`java.lang.Boolean`	Boolean
`java.util.Date`	Date, Time or DateTime
`java.lang.String`	String

The first positional parameter must be an array of one of these allowable types.

6

Sequence extended operators

Sequence Extended Operators process a sequence (list) of input attribute values and return a single value. Consider this example expression:

```
Trade.mavg=security.price->sortedBy(marketDate)->mavg(30)
```

Assuming that `Trade.mavg` is a Decimal attribute, `marketDate` a Date attribute, and `security.price` refers to a collection of Decimal attributes, Sequence Extended Operator `mavg` might return a Decimal value that is the moving average of the first 30 instances of sequence `security.price`.

To create a Sequence Extended Operator, you must code a Java class, and your class must implement marker interface `ICcSequenceExtension`. Example:

```
package com.corticon.operations.extended.examples.operators.set1;

import java.math.BigDecimal;
import java.math.BigInteger;

import com.corticon.services.extensions.ICcSequenceExtension;

public class S1Sequence implements ICcSequenceExtension
{
   private static final int DECIMAL_SCALE = 4;

   public static BigDecimal mavg(BigDecimal[] abdInputArray, BigInteger
abiElements)
   {
      if (abdInputArray == null || abiElements == null)
      {
         return new BigDecimal("0");
      }

      int liElements = abiElements.intValue();

      BigDecimal lbdSum = new BigDecimal("0").setScale(DECIMAL_SCALE);
      int liDenominator = 0;
      for (int liIndex = 0; liIndex < abdInputArray.length &&
       liIndex < liElements; liIndex++)
      {
         lbdSum = lbdSum.add(abdInputArray[liIndex]);
         liDenominator++;
      }

     BigDecimal lbdDenominator = new BigDecimal(String.valueOf(liDenominator));

      return lbdSum.divide(lbdDenominator, DECIMAL_SCALE,
BigDecimal.ROUND_HALF_UP);
   }
}
```

Sequence Extended Operator methods must be declared `public static`.

In this example, the first positional parameter of method `mavg` is an array of BigDecimal instances. The rules engine will populate this array with the Decimal values in `security.price` (that is, the sequence upon which the method operates).

Note: Collection **`security.price`** has been sorted into **marketDate** sequence via built-in operator **sortedBy**.

The example Java method analyzes this array and returns a Decimal value. The rules engine will then assign `Trade.mavg` the Decimal return value.

As with Collection Extended Operators, you cannot code Sequence Extended Operators for entity instances, nor can you code extensions for Collections or Sequences of entity instances.

When implementing Sequence Extended Operators, you must limit your parameter and return types to the following:

Java Type	Corticon Type
`java.math.BigInteger`	Integer
`java.math.BigDecimal`	Decimal
`java.lang.Boolean`	Boolean
`java.util.Date`	Date, Time or DateTime
`java.lang.String`	String

The first positional parameter must be an array of one of these allowable types.

7

Stand-alone extended operators

Stand-alone extended operators define functions that can be used in Rulesheet expressions. Unlike Attribute Extended Operators, they are not tied to Vocabulary attributes and are not associated with any particular data type.

Users invoke Stand-Alone extended operators by explicitly specifying a class name and method name in Rulesheet expressions.[1]

Stand Alone extension classes may contain any number of static methods.

Consider this expression:

```
Circle.circumference = SeMath.getCircumference(Circle.radius)
```

Assuming `Circle.circumference` and `Circle.radius` are both Decimal attributes, Stand-Alone Extended Operator `SeMath.getCircumference` might convert the radius to circumference, which the rule engine will assign to `Circle.circumference`.

To create Stand Alone Extended Operator, you must code a Java class, and your class must interface `ICcStandAloneExtension`. Example:

```
package com.corticon.operations.extended.examples.standalone.set1;

import java.math.BigDecimal;

import com.corticon.services.extensions.ICcStandAloneExtension;

public class SeMath implements ICcStandAloneExtension
{
   public static BigDecimal getCircumference(BigDecimal abdRadius)
   {
      BigDecimal lbdBigDecimal = abdRadius.multiply(new BigDecimal(2.0));
      lbdBigDecimal = lbdBigDecimal.multiply(new BigDecimal(Math.PI));
      return lbdBigDecimal;
   }
}
```

Stand-Alone Extended Operator methods must be declared `public static`. Any number of parameters may be specified. The Rulesheet expression parameter list and return value must match the extension class method signature; otherwise, the Rulesheet expression will be flagged as invalid.

In this example, the rules engine will pass the current value of `Circle.radius` to class `com.corticon.operations.extended.examples.standalone.set1.SeMath` method `getCircumference` as parameter `abdRadius`; thus, `abdRadius` must be declared as type BigDecimal.

Similarly to Attribute Extended Operators, you must limit parameter and return types to the following:

Java Type	Corticon Type
`java.math.BigInteger`	Integer
`java.math.BigDecimal`	Decimal
`java.lang.Boolean`	Boolean
`java.util.Date`	Date, Time or DateTime
`java.lang.String`	String

Note that you may code any number of extension methods in a Java class, and you can supply one or more Java classes; regardless, the system will discover all of your methods via Java reflection.

[1] The user specifies the unqualified part of the class name, namely the class name without the package name.

8

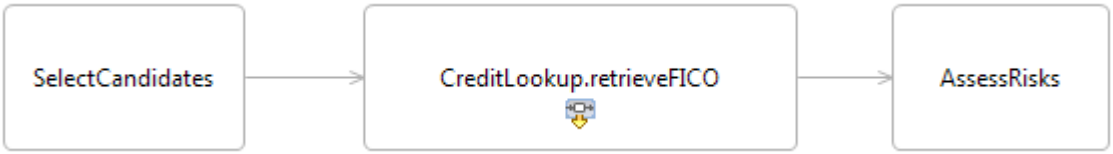
Service call-out extensions

Service Call-out Extensions are user-written functions that can be invoked in a Ruleflow.

In a Ruleflow, the flow of control moves from Rulesheet to Rulesheet, with all Rulesheets operating on a common pool of facts. This common pool of facts is retained in the rule execution engine's working memory for the duration of the transaction. Connection arrows on the diagram specify the flow of control. Each Rulesheet in the flow may update the fact pool.

When you add a Service Call-out to a Ruleflow diagram, you effectively instruct the system to transfer control to your extension class a specific point in the flow. Your extension class can directly update the fact pool, and your updated facts are available to subsequent Rulesheets.

Consider this example:



After the rule engine finishes processing Rulesheet `SelectCandidates`, it will transfer control to Service Call-out Extension class `CreditLookup`, method `retrieveFICO`.

Using the Service Call-out API, class `CreditLookup`, method `retrieveFICO` can create, retrieve, update and remove facts. For example, it might look up a customer's FICO score using an external web service, and update the facts accordingly.

When `CreditLookup.retrieveFICO` finishes, the system will transfer control to the next Rulesheet in the Ruleflow. Downstream Rulesheets (for example, `AssessRisks`) will evaluate and respond to new facts asserted by your Service Call-out.

9

Service call-out API

Your Service Call-outs use an API to retrieve and update facts. The API is comprised of these Java interfaces:

Interface	Purpose
`com.corticon.services.dataobject.ICcDataObjectManager`	Provides access to the entire fact pool. Allows you to create, retrieve and remove entity instances.
`com.corticon.services.dataobject.ICcDataObject`	Provides access to a single entity instance. Allows you to update the entity instance, including setting attributes and creating and removing associations.

These interfaces can be found in:

Usage Scenario	Location
Eclipse	Eclipse plug-in `com.corticon.services`
Foundation API	`Corticon_eFoundation_API.jar`

Your Service Call-out class must implement marker interface `ICcServiceCalloutExtension`.

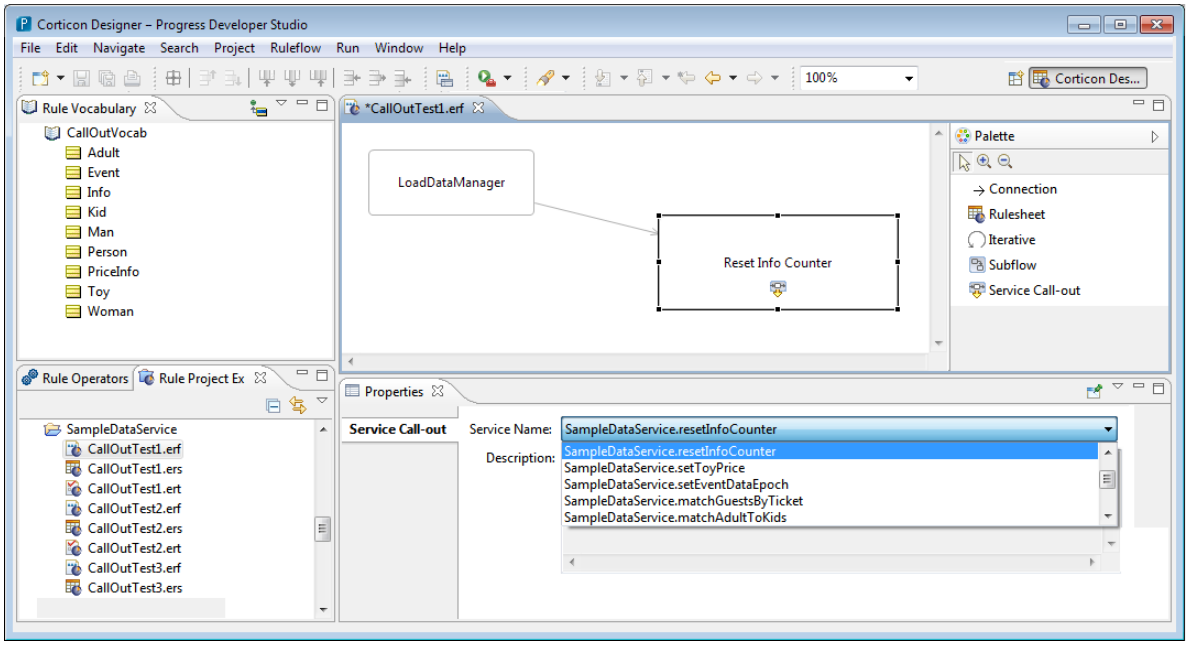
Example:

```
package com.corticon.operations.extended.examples.servicecallouts;

import com.corticon.services.dataobject.ICcDataObject;
import com.corticon.services.dataobject.ICcDataObjectManager;
import com.corticon.services.extensions.ICcServiceCalloutExtension;

public class SampleDataService implements ICcServiceCalloutExtension
{
   public static void retrievePersonInfo(ICcDataObjectManager aDataObjMgr)
   {
      for (ICcDataObject lPerson : aDataObjMgr.getEntitiesByName("Person"))
      {
         String lstrName = "Tom";
         Boolean lbMarried = new Boolean(true);

         lPerson.setAttributeValue("name", lstrName);
         lPerson.setAttributeValue("married", lbMarried);
      }
   }
}
```

Service Call-out methods must be declared `public static`.

The system will recognize your Service Call-out method if the method signature takes one parameter and that parameter is an instance of `ICcDataObjectManager`.

Recognized classes and methods will appear in the Ruleflow Properties View/Service Name drop-down list when a Service Call-out shape is selected in the Ruleflow diagram:



Example `SampleDataService.resetInfoCounter` illustrates how a Service Call-out can use the supplied `ICcDataObjectManager` to find and update entity instances.

`ICcDataObjectManager` method `getEntitiesByName` allows you to retrieve a Set of all entity instances with a given name. Each element of the returned Set is an instance of `ICcDataObject` which offers additional methods to access and update a specific entity instance.

The example finds all `Person` entities then iterates over them to set attributes `Person.name` and `Person.married` using generic method `ICcDataObject.setAttributeValue`.

The Service Call-out API provides your extension class complete access to the fact pool, allowing you to:

- Find entities in several ways
- Read and update entity attributes
- Create and remove entity instances
- Create and remove associations
- Post rule messages

Refer to Service Call-out Java source code examples in your Corticon Studio `/Samples/Extended Operators` folder, and see the *API Javadocs* for more information.

10

Creating an extension plug-in

You must create an Eclipse plug-in to encapsulate your extensions. The system will automatically recognize an optional plug-in named:

`com.corticon.eclipse.studio.operations.extended.core`

You can use your Eclipse IDE to create a *plug-in project* to develop your extensions and ultimately prepare them for deployment.

For details, see the following topics:

- Setting up your development Eclipse IDE
- Creating your plug-in project
- Creating the extension locator file
- Creating an extension class
- Coding your extension class
- Updating your plug-in manifest
- Organizing imports
- Exporting your plug-in
- Installing your plug-in
- Testing your extension
- Troubleshooting extensions
- Extension plug-in not resolved by Eclipse
- Extension locator file not set up correctly

- Extension classes not implementing marker interfaces
- Enabling logging to diagnose extensions issues

Setting up your development Eclipse IDE

To develop plug-ins, you must use the Java Development Kit (JDK), not just a Java Runtime Environment (JRE).

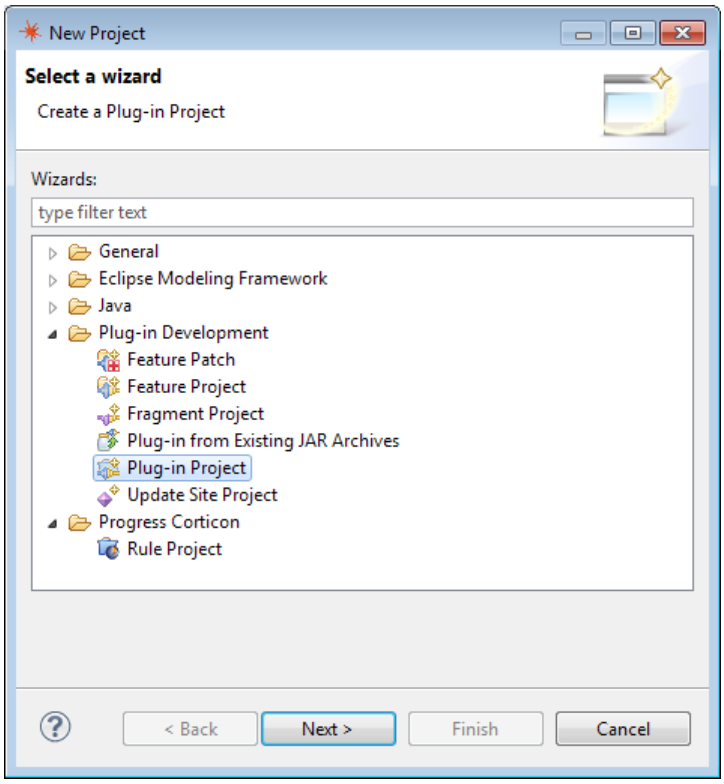
1. Download and install a supported JDK from www.oracle.com
2. Select **Window** > **Preferences** > **Java** > **Installed JREs**
3. Navigate to your downloaded JDK and select it as your default JRE.
4. Copy `tools.jar` from your JDK's `/lib` directory into `/jre/lib/ext` so that you can compile and test Rulesheets, .

Creating your plug-in project

To create a plug-in project:

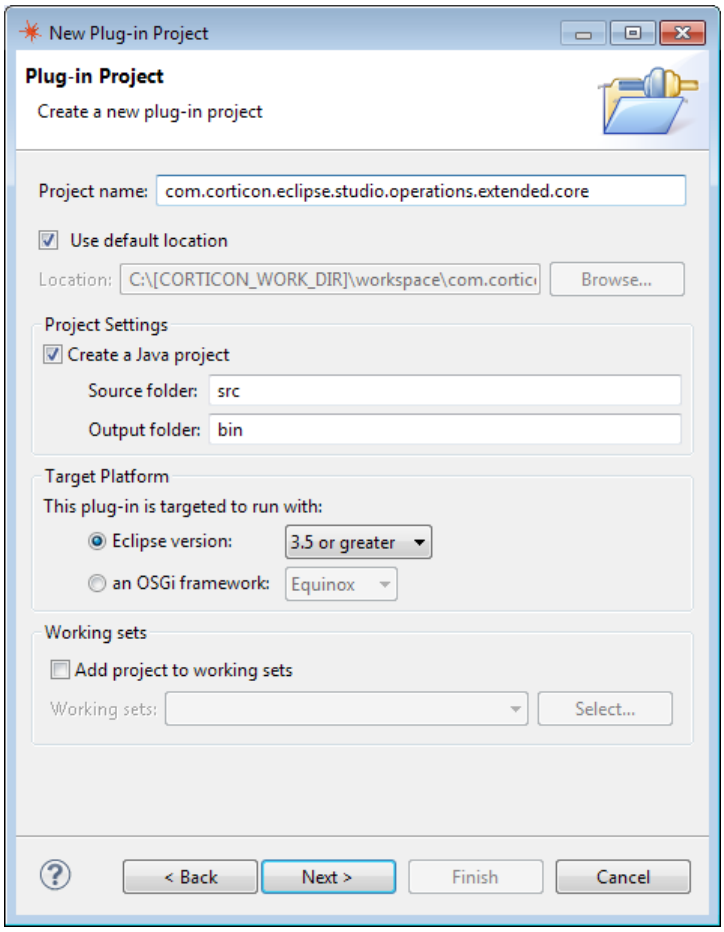
1. Using the Java Perspective select:

   Select **New** > **Project** > **Plug-in Project**.



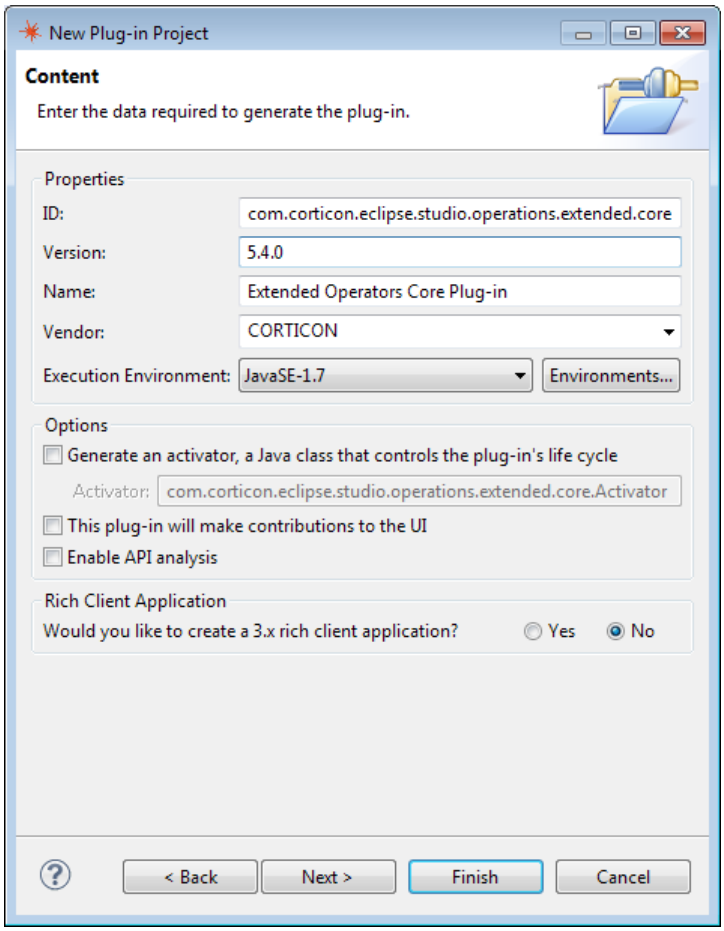
2. Click **Next**

3. Specify the plug-in **Project name** `com.corticon.eclipse.studio.operations.extended.core`.

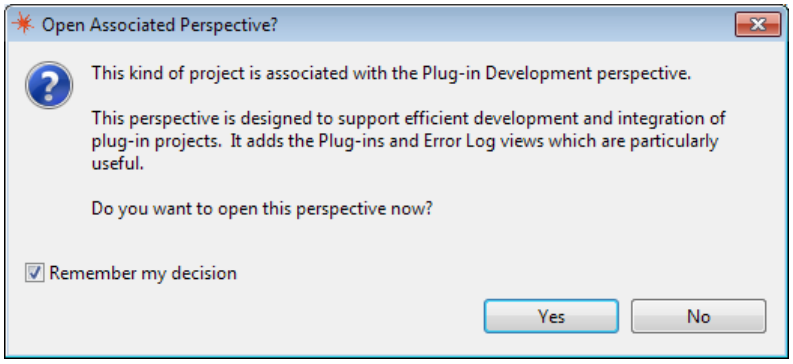


4. Click **Next**.

   The **Content** dialog opens.

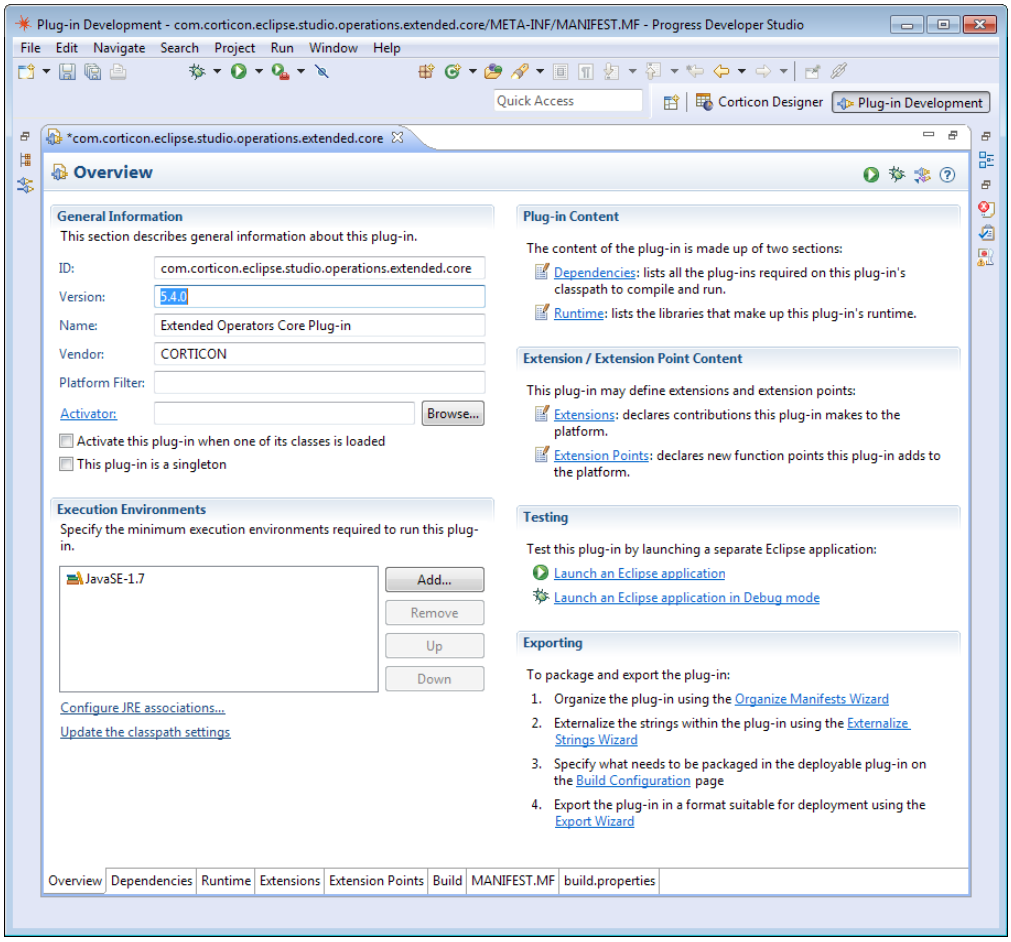


5. In the **Content** dialog:
   a) In the **Version** field, enter a version number for your plug-in. You should specify a version *greater* than the version of Studio you installed. For example, if you installed Studio Version 5.4.0, you might specify version **5.4.9**. This should ensure that your plug-in supersedes our example plug-in throughout the 5.4 release.
   b) In the **Name** field, enter `Extended Operators Core Plug-in`.
   c) In the **Provider** field, enter your company name.
   d) Uncheck **Generate an activator**.
   e) Uncheck **This plug-in will make contributions to the UI**.
   f) Click **Finish** to close the dialog.
6. When prompted to switch to the Plug-in Development perspective, check **Remember my decision**, as shown:



7. Click **Yes**.

   The system switches to **Plug-in Development** perspective, and then creates a new plug-in project in your workspace.
8. The *plug-in manifest* file (`MANIFEST.MF`) opens to let you configure the new plug-in:

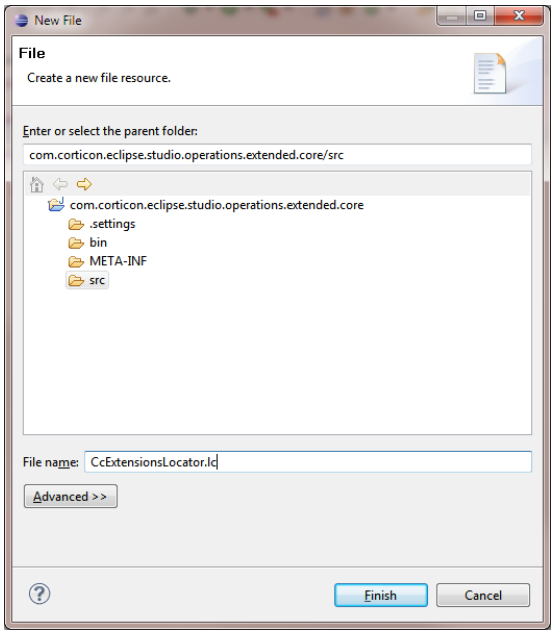


9. Close the plug-in manifest editor. (This task will be completed later.)

Creating the extension locator file

Right click on the `/src` node in the Package Explorer and select **New** > **File**

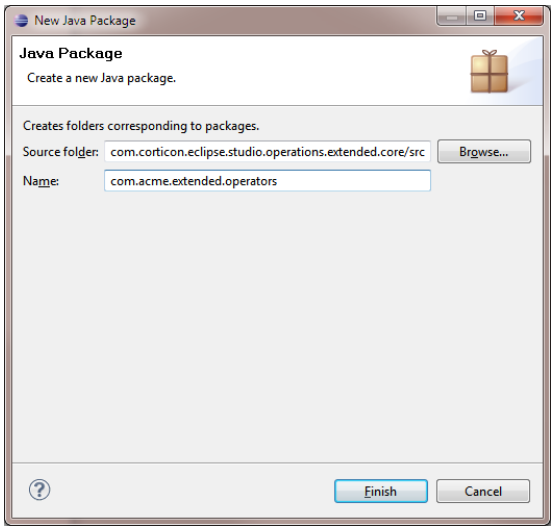
The system will prompt for a new file name. Specify `CcExtensionsLocator.lc` as shown:



Press **Finish** to proceed.

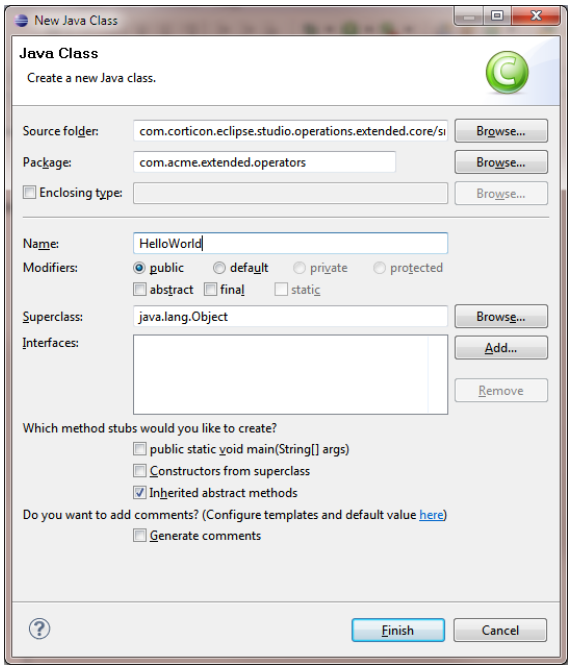
Creating an extension class

Right click on the `/src` node in the Package Explorer and select **New** > **Package**. Specify the package name for your new extension classes:

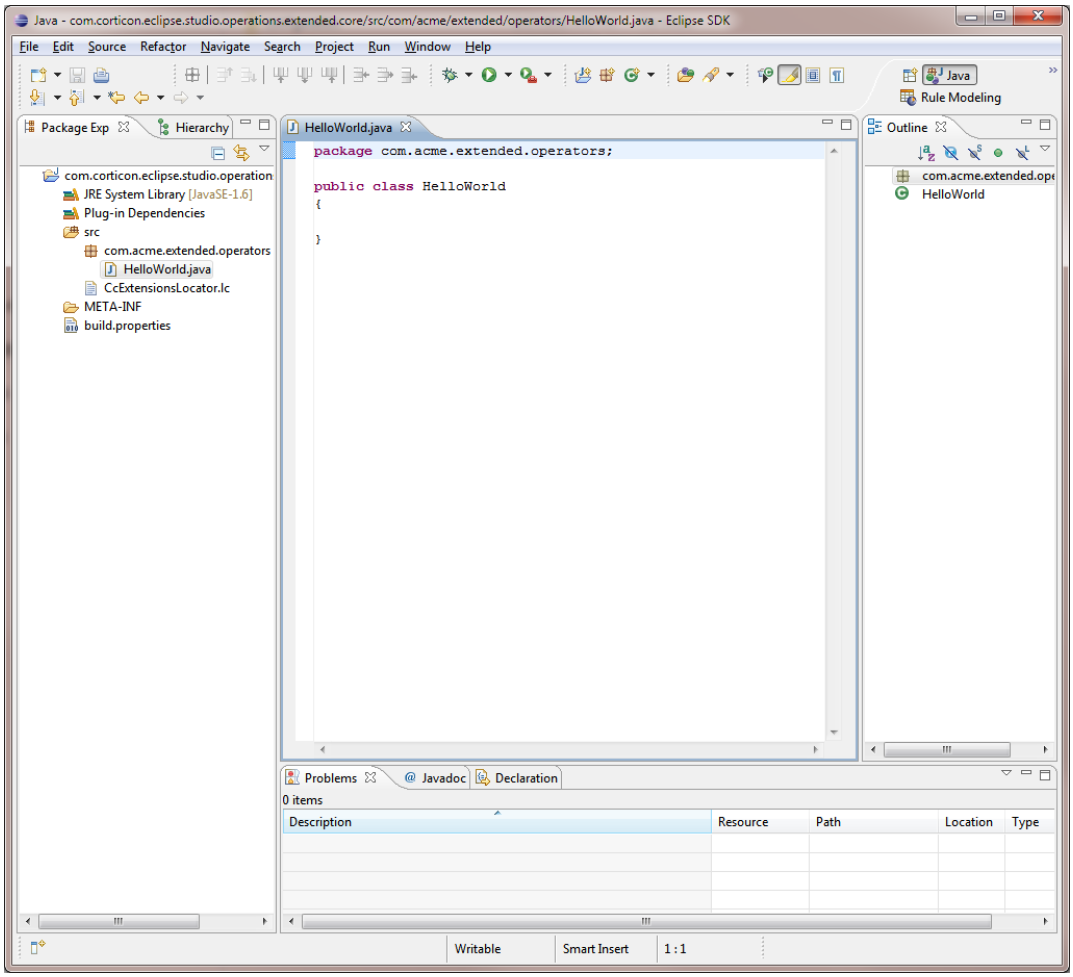


Press **Finish** to proceed.

Right-click on your new package and select **New** > **Class**:



Enter a name for your new Java class (for example, `HelloWorld`) and press **Finish**. The system will create an empty class definition:

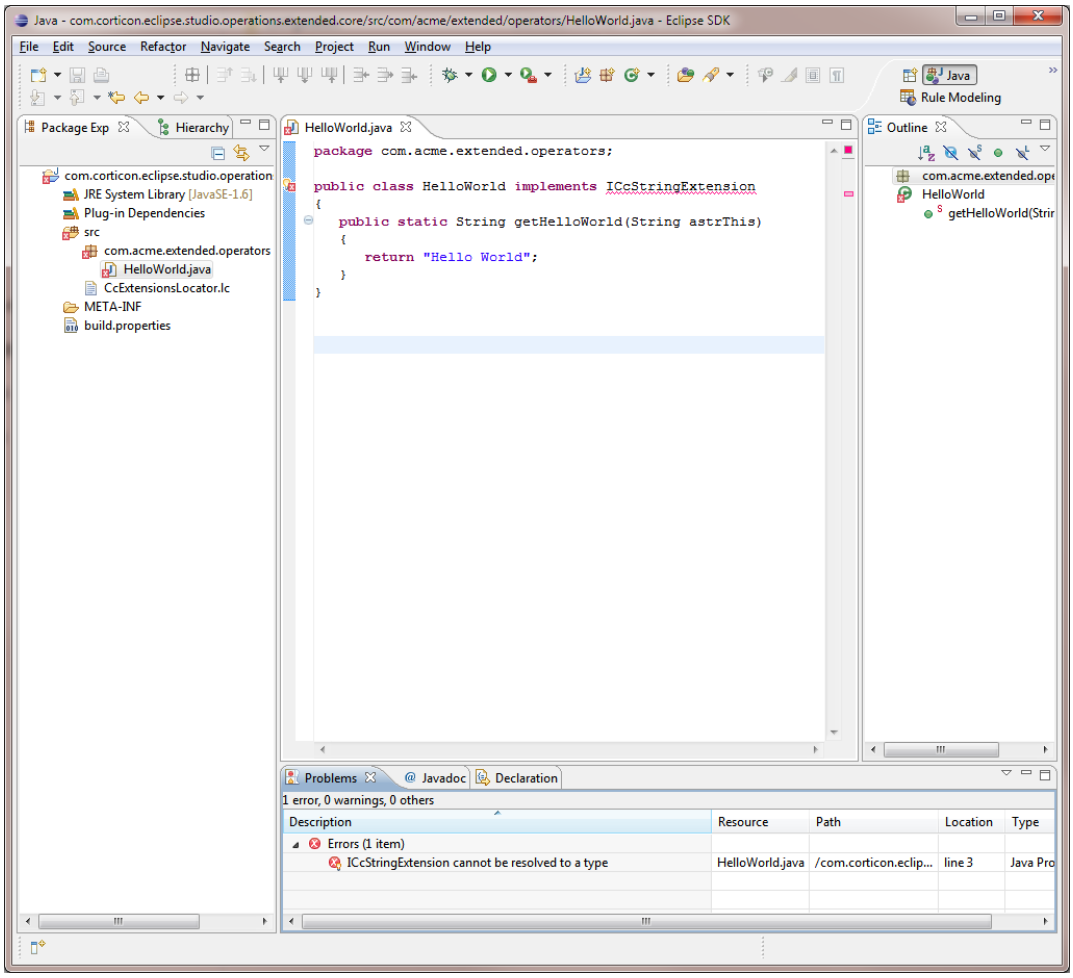


Coding your extension class

Code your extension class as shown:

```
package com.acme.extended.operators;

public class HelloWorld implements ICcStringExtension
{
   public static String getHelloWorld(String astrThis)
   {
      return "Hello World";
   }
}
```

Note that Eclipse cannot resolve interface `ICcStringExtension`:



Updating your plug-in manifest

To correct build errors, open `MANIFEST.MF` located in your new plug-in META-INF directory, and then select the **MANIFEST.MF** tab and update the manifest as shown in italics below:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Extended Operators Core Plug-in
Bundle-SymbolicName: com.corticon.eclipse.studio.operations.extended.core
Bundle-Version: 5.4.0
Bundle-Vendor: Your Company
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Require-Bundle: org.eclipse.core.runtime,
 com.corticon.legacy,
 com.corticon.services,
 com.corticon.thirdparty
Export-Package: .,
 com.acme.extended.operators
Eclipse-RegisterBuddy: com.corticon.legacy
```

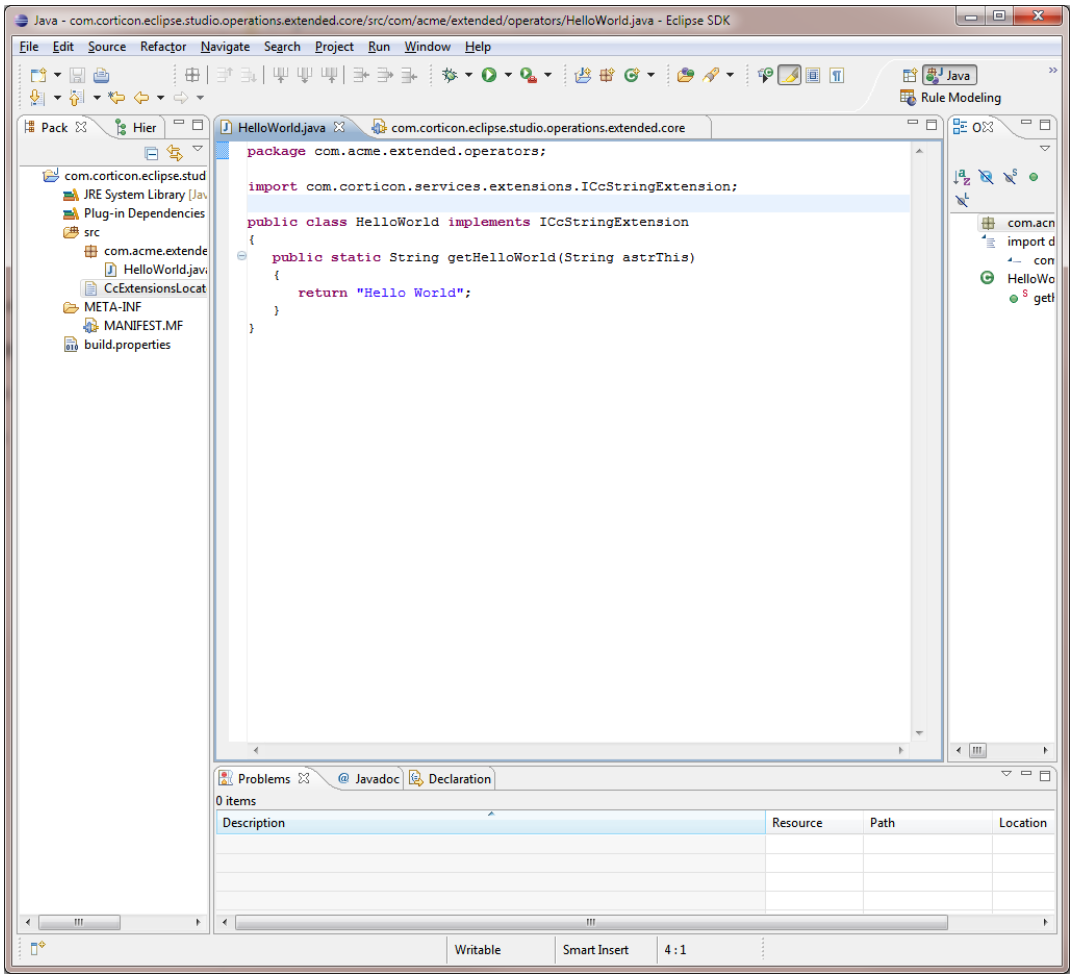
These changes will:

- Ensure that your plug-in has access to interface *`ICcStringExtension`* in `com.corticon.services`
- Ensure that your locator file `CcExtensionLocator.lc` and Java package *`com.acme.extended.operators`* are visible to the Corticon extensions subsystem
- Ensure that your plug-in is registered as a "buddy" with `com.corticon.legacy` to allow the Corticon class loader to find your extensions

Click **Save** to save your manifest changes.

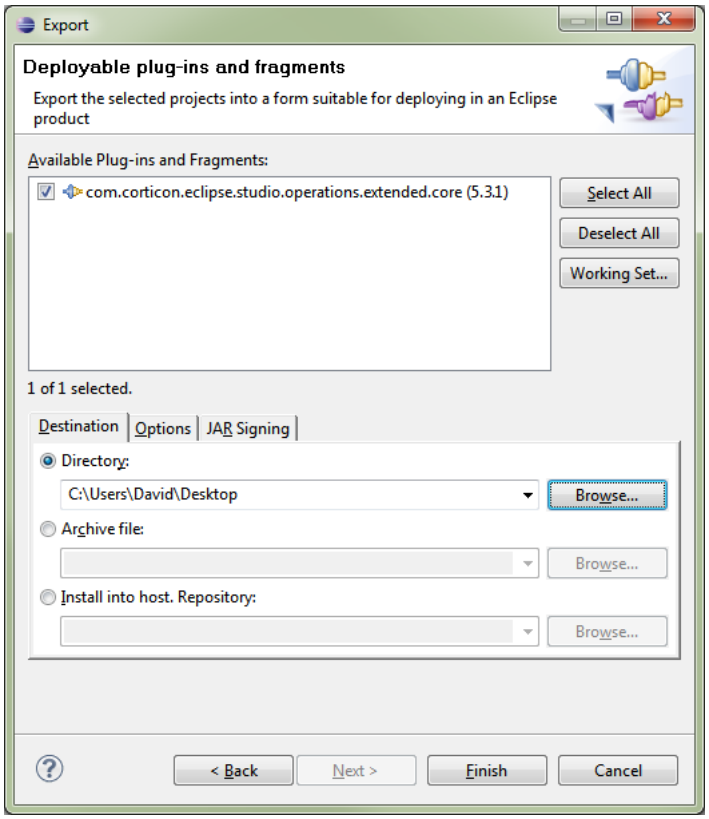
Organizing imports

Activate the `HelloWorld` Java editor and press CTRL-SHIFT-O to organize imports. The system will add import `com.corticon.services.extensions.ICcStringExtension` allowing your class to build without errors:



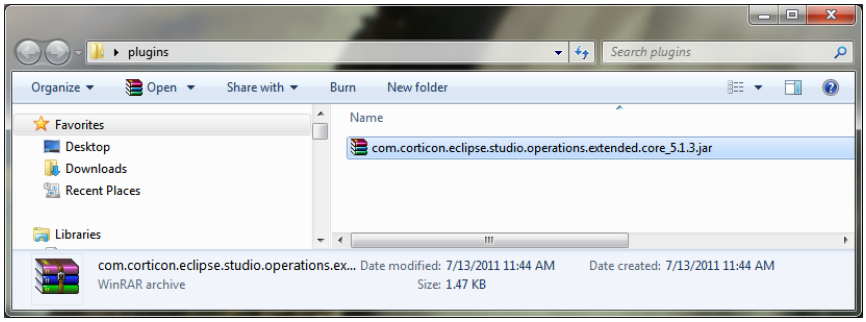
Exporting your plug-in

Close all open editors, right-click your Project in the Package Explorer view and select: **Export…** > **Plug-in Development** > **Deployable plug-ins and fragments**:



Navigate to your preferred output directory, and then click **Finish**.

An installable plug-in is created in your output directory:



Installing your plug-in

Copy your exported plug-in into the target Eclipse environment `/plugins` directory.

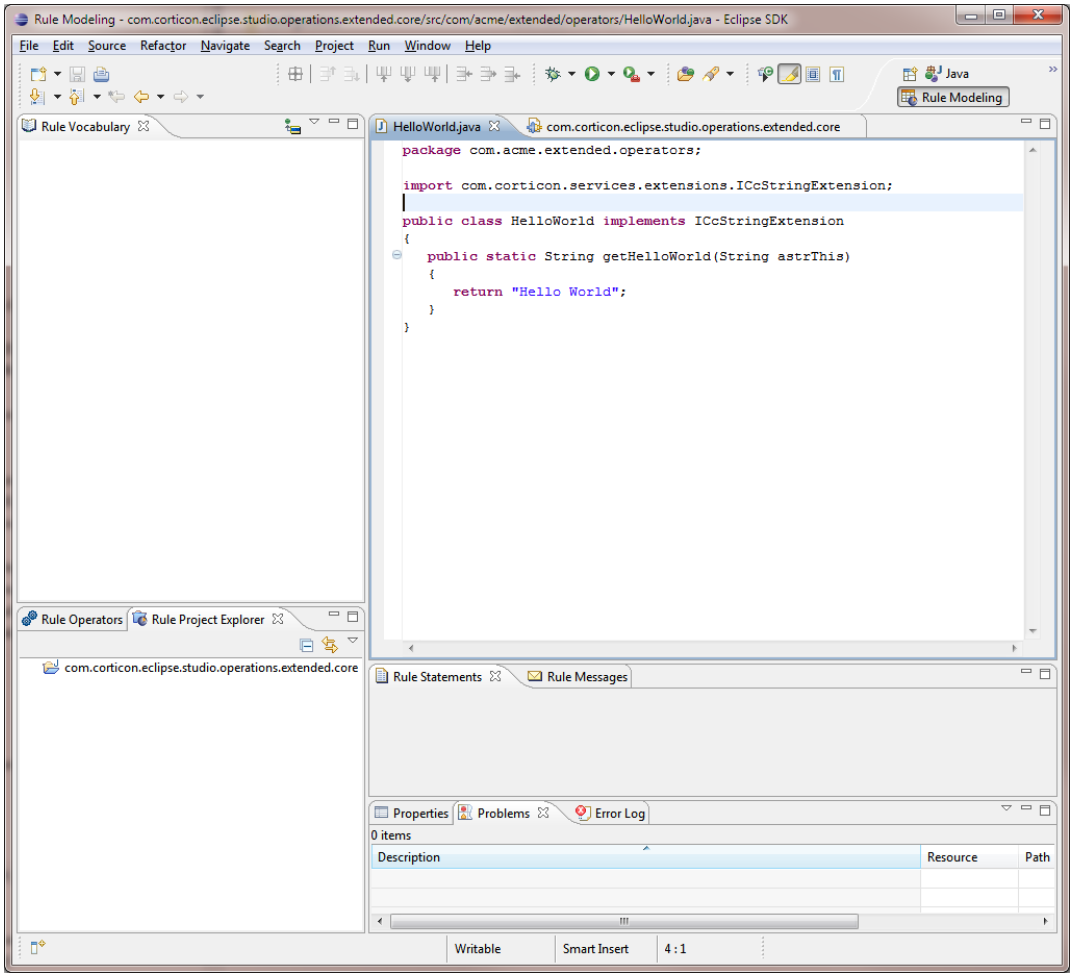
It is important that you restart your target Eclipse environment **with administrator privileges** whenever you update your plug-in so that the plug-in gets properly installed and registered.

It is a good practice to use the `-clean` command line option when you restart Eclipse to force the system to rebuild the bundle cache, ensuring that your changes take effect.

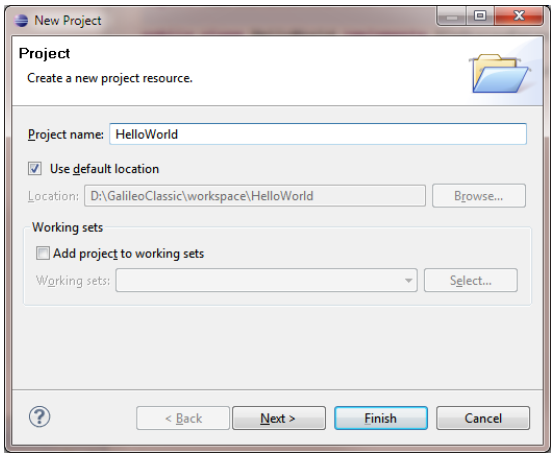
Testing your extension

Create a Rule Project as a home for your Corticon test assets. Switch to **Corticon Designer** perspective by selecting:

Window > **Open Perspective** > **Other** > **Corticon Designer**, and then click **OK**. The system rearranges the editors and views as shown:

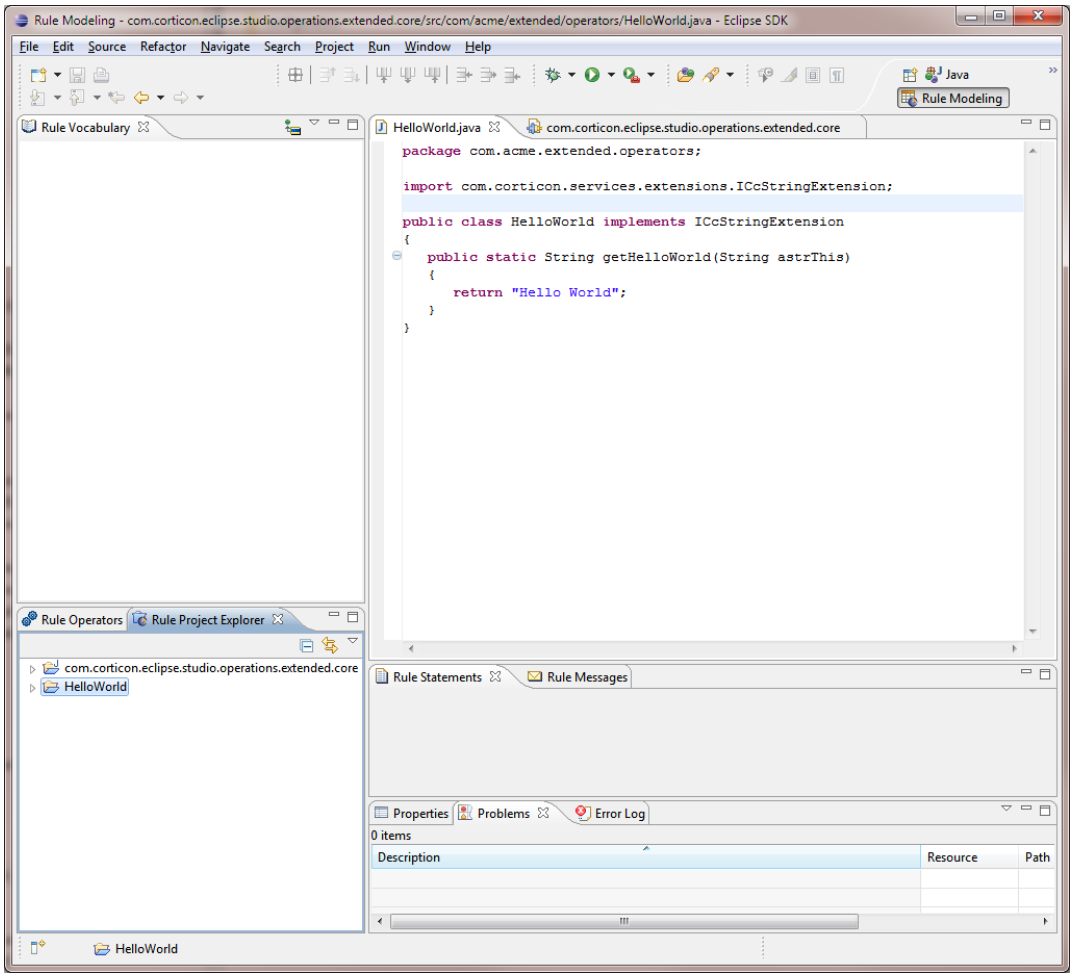


Right-click in the body of the **Rule Project Explorer** and select **New** > **Rule Project**

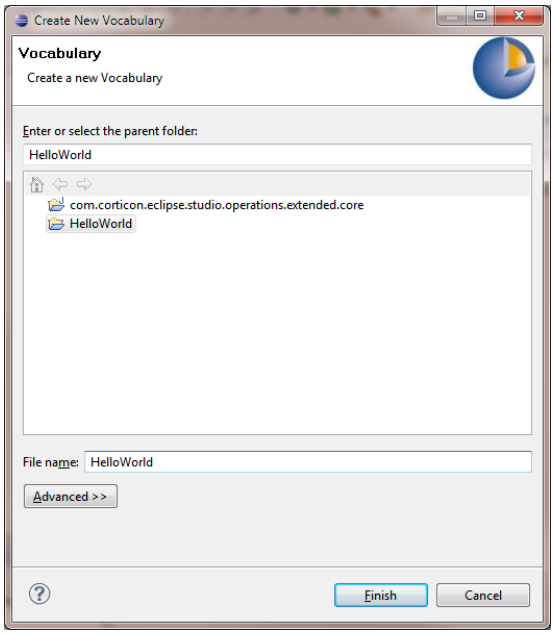


Enter `HelloWorld` in the **Project name** field and press **Finish**.

The system will create a new Rule Project visible in the **Rule Project Explorer**:

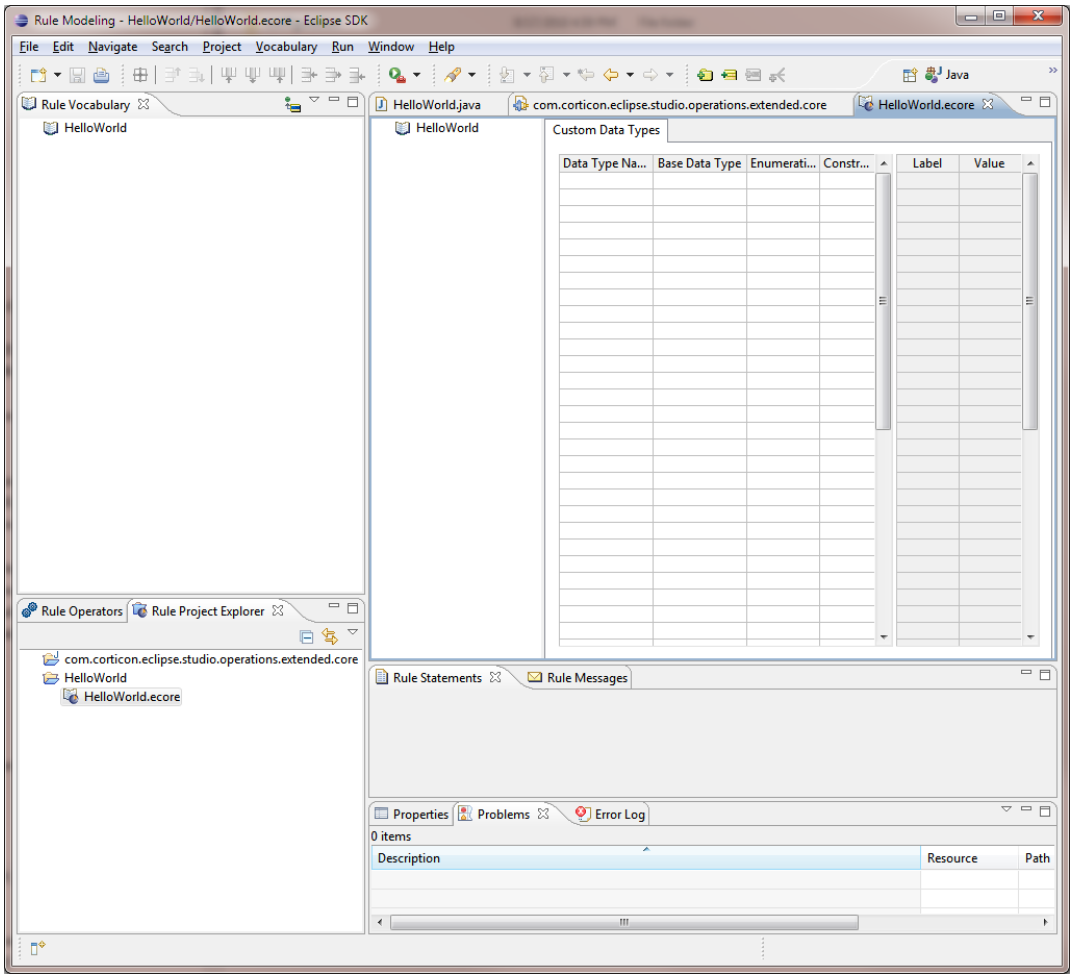


Right-click the **HelloWorld** project and select **New** > **Rule Vocabulary**:

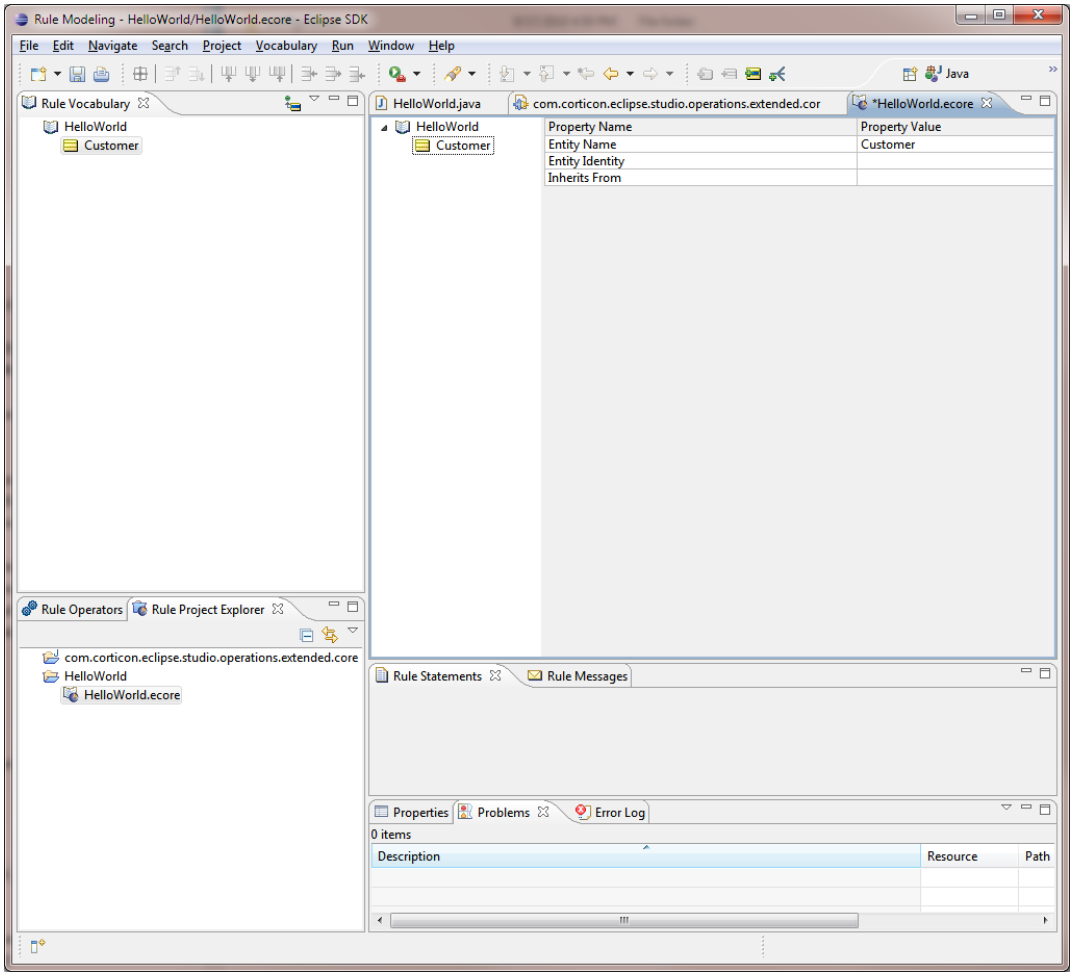


Enter `HelloWorld` in the **File name** field and press **Finish**.

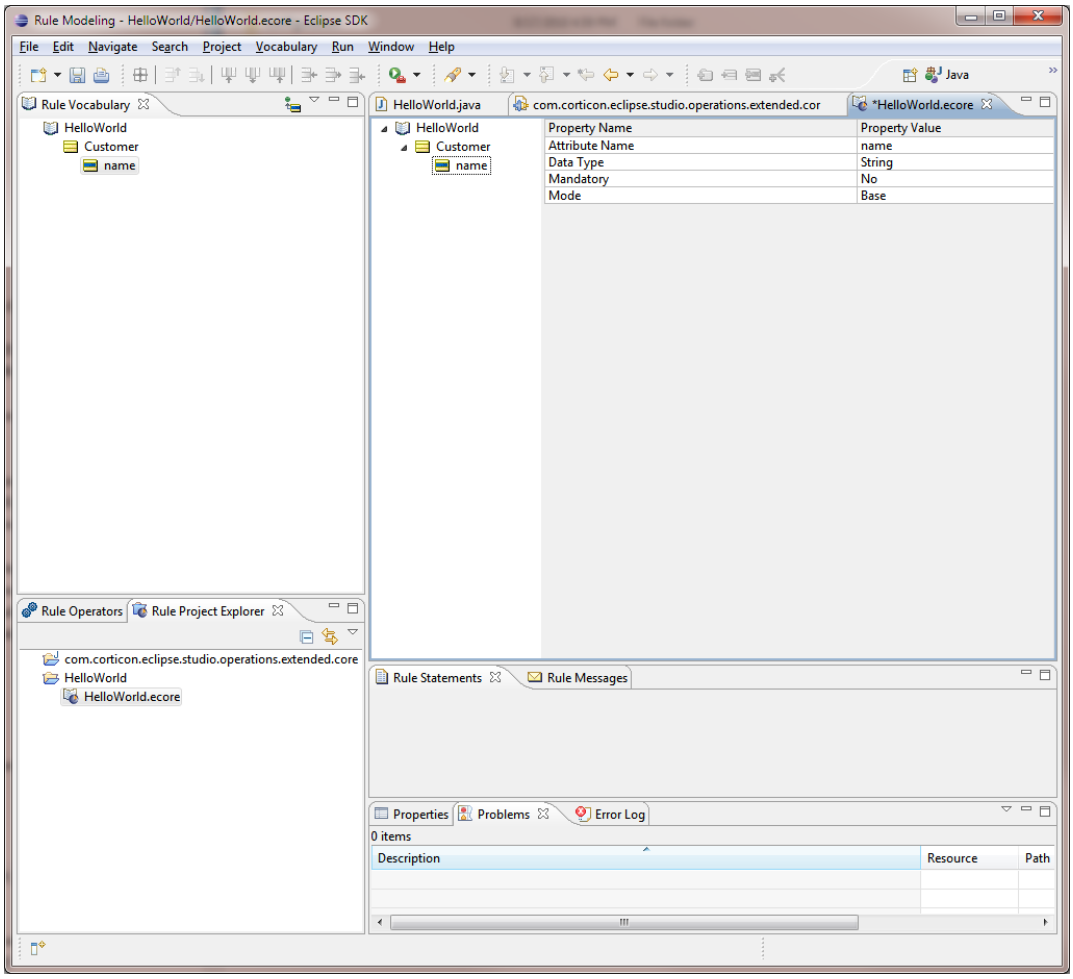
The system will create a new Vocabulary:



In the Vocabulary Editor, right-click the **HelloWorld** root node and select **Add Entity** to create a new entity `Customer`:

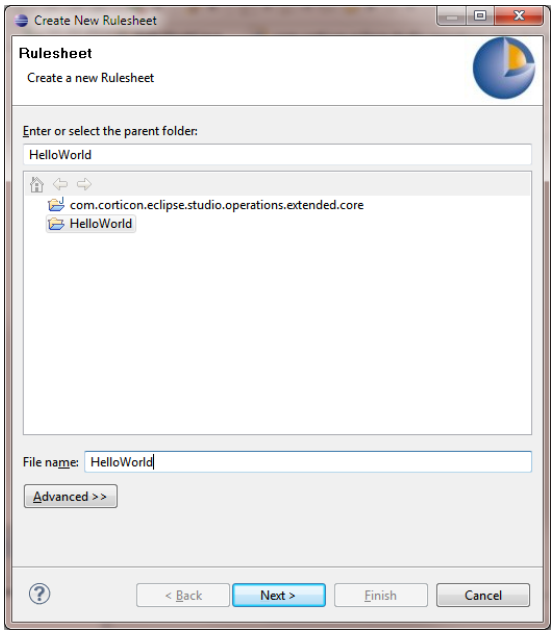


Right click **Customer** and select **Add Attribute** to create String attribute `name`:

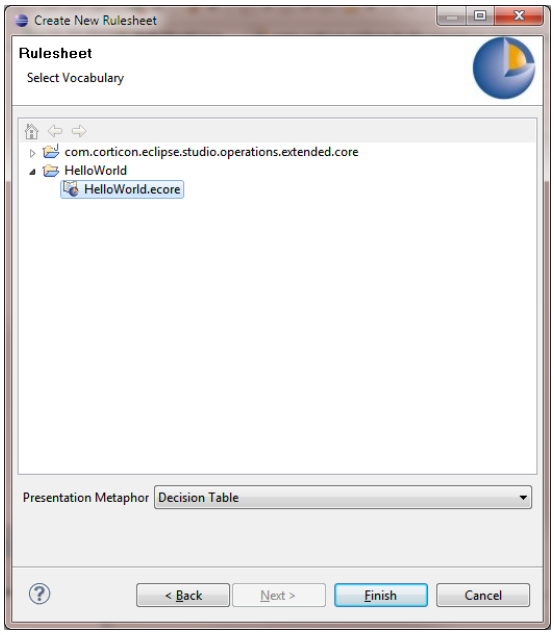


Press **Save** to save your new Vocabulary.

Select the **HelloWorld** project in the Rule Project Explorer and select **New** > **Rulesheet**:

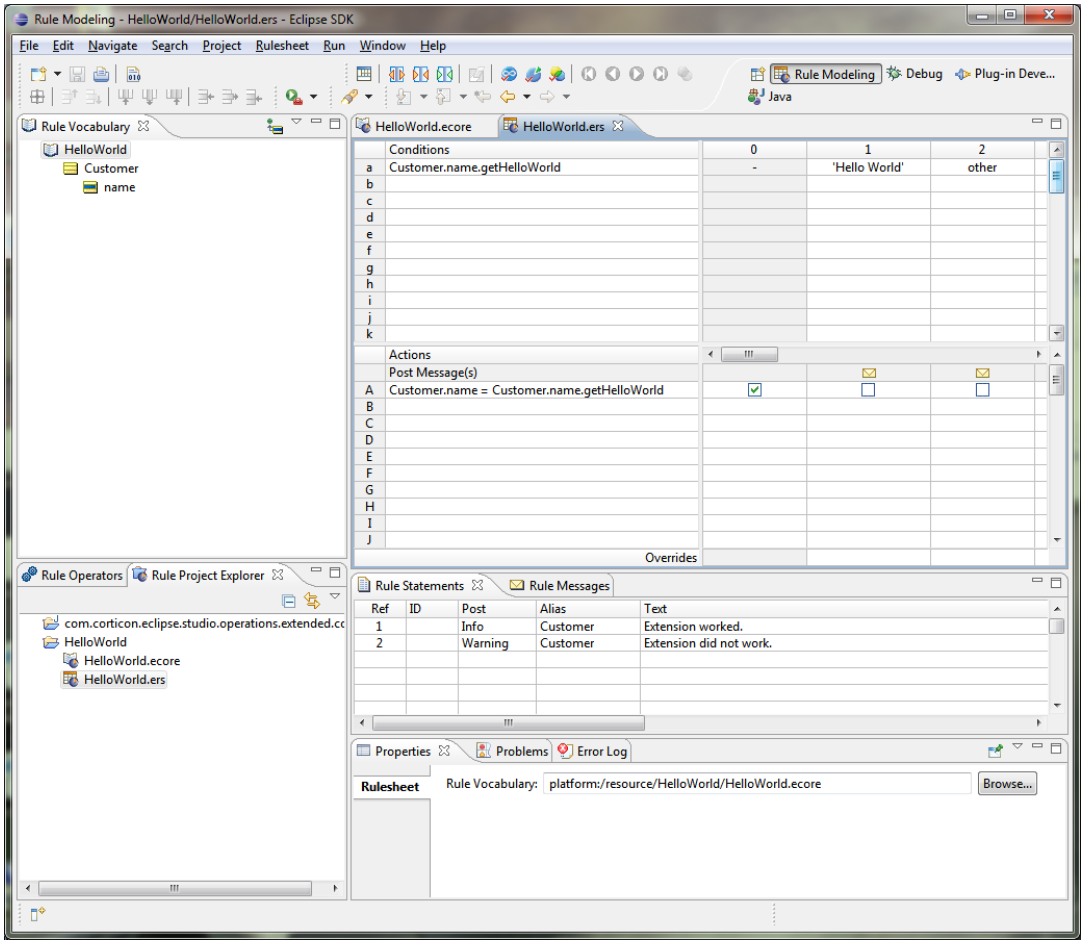


Press **Next** and then select the Vocabulary asset you created in the prior step:



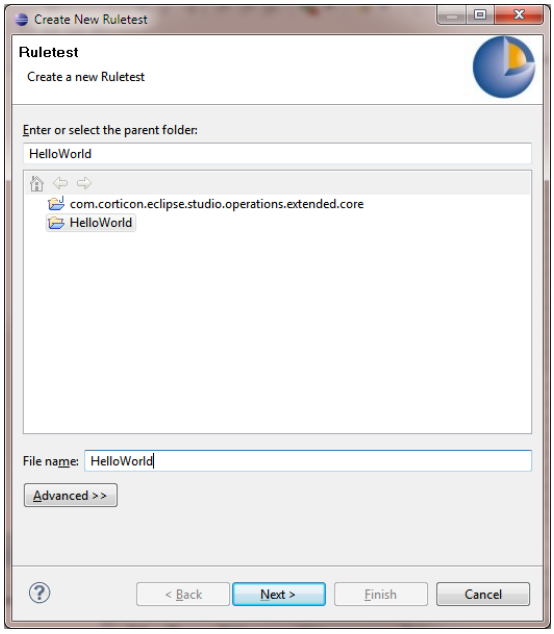
Press **Finish** and the system will create a new empty Rulesheet.

Update your Rulesheet as shown:



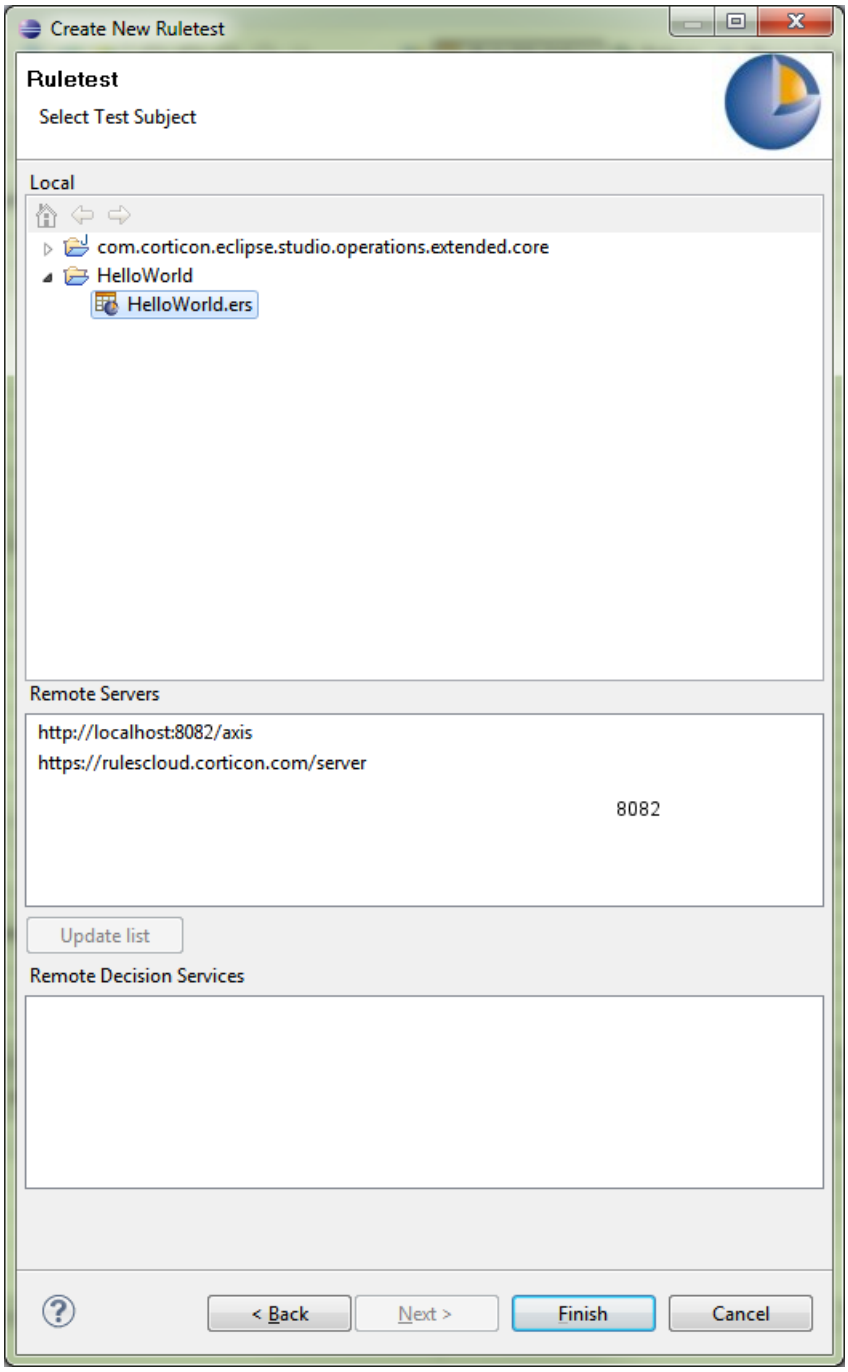
Press **Save** to save your Rulesheet.

Right click on your **HelloWorld** project and select **New** > **Ruletest**. Specify `HelloWorld` in the **File name** field:

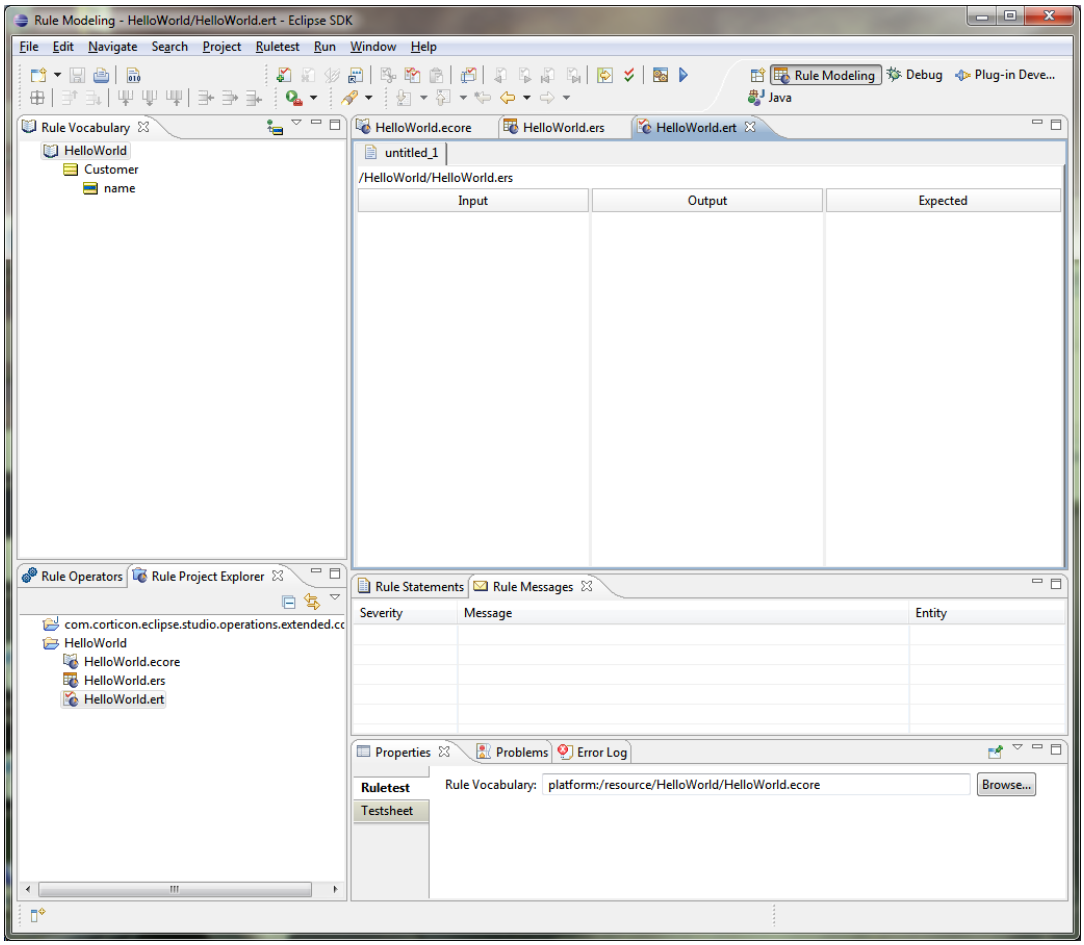


Press **Next**.

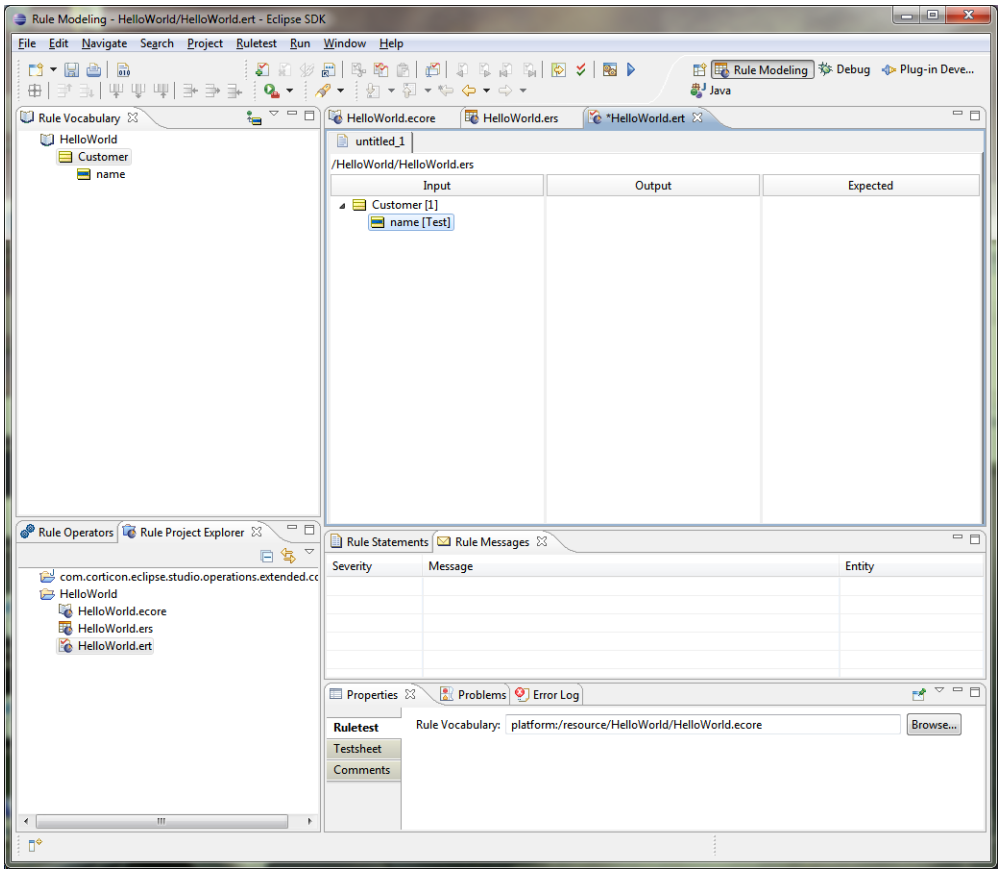
Select your Rulesheet **HelloWorld.ers** as the Test Subject:



Press **Finish** and the system will create an empty Ruletest asset.

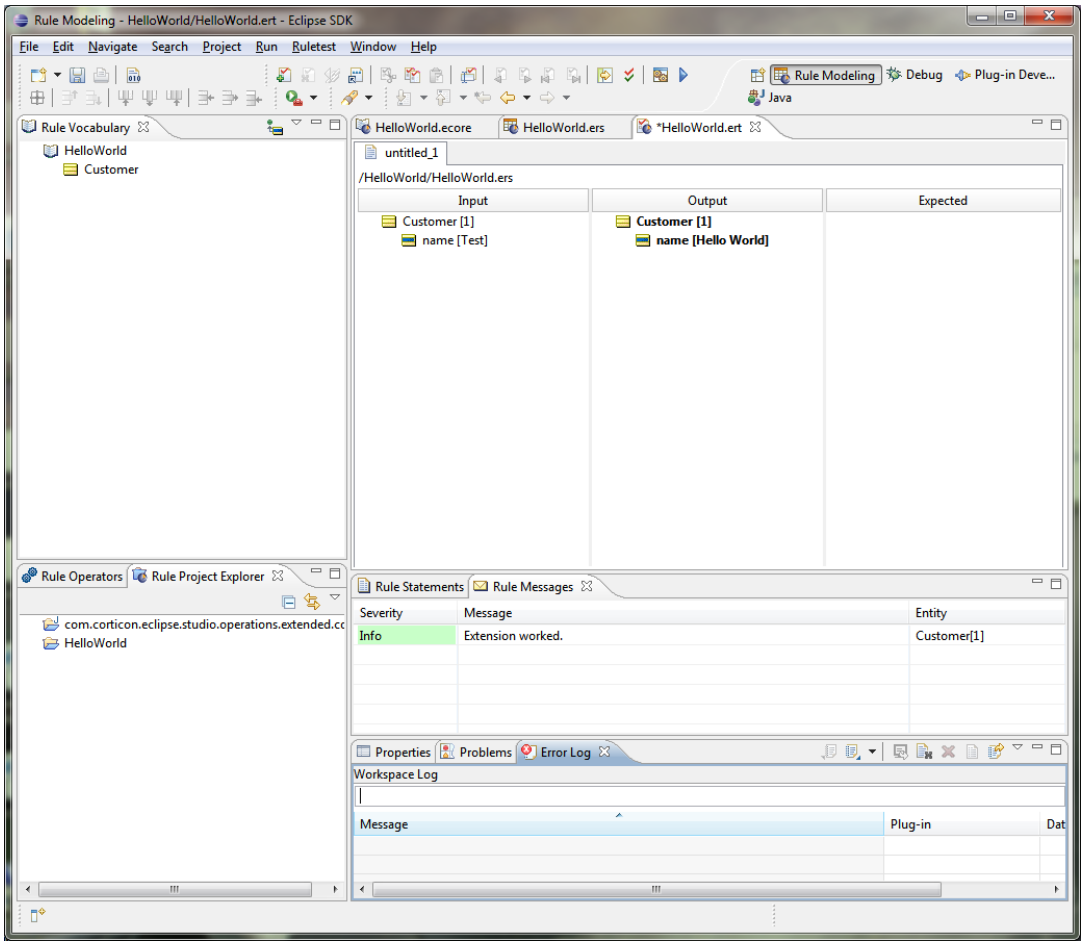


Drag entity `Customer` from the Vocabulary view to the Ruletest input tree:



Important: Double-click **Customer[1] name** and enter value `Test`.

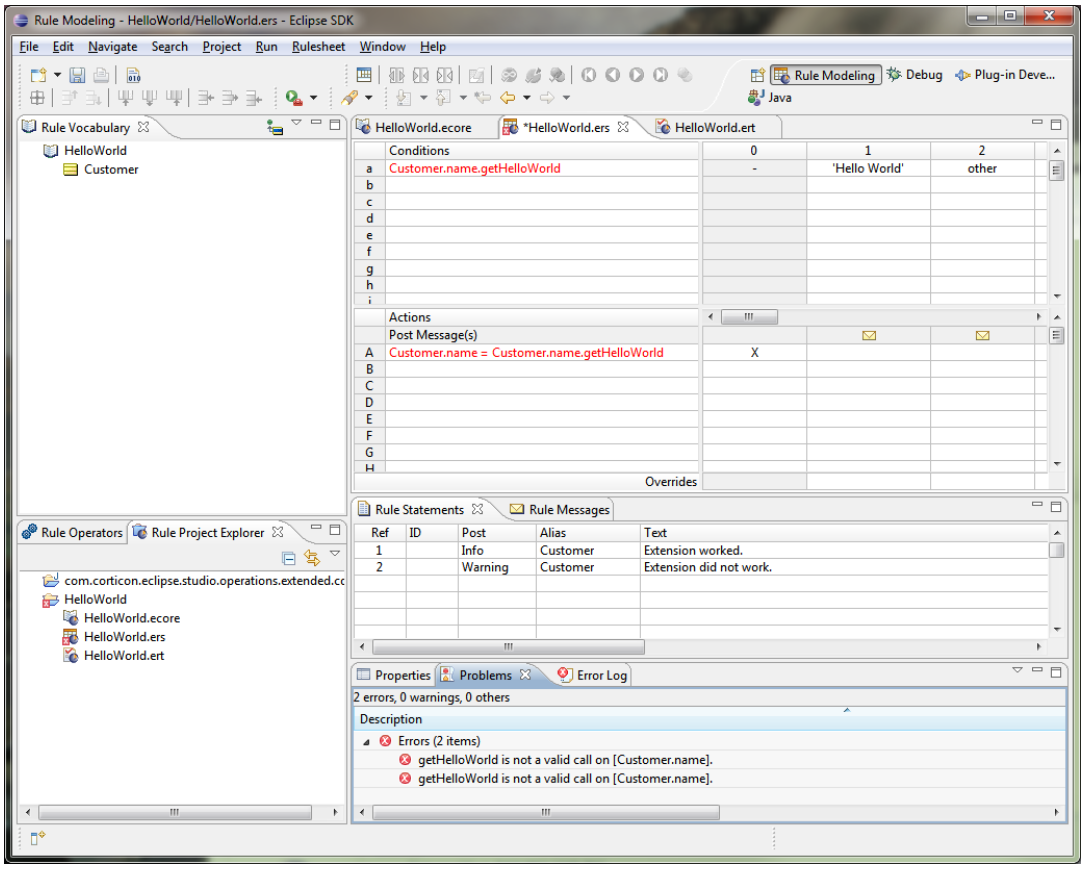
Select **Ruletest** > **Testsheet** > **Run Test** and you should see the following test results:



If you see `Hello World` in the output tree, congratulations! Your extensions are working properly.

Troubleshooting extensions

If your extensions are not properly written or deployed, the system will color Rulesheet expressions red and will display validation messages in the **Problems** tab:



For Attribute Extended Operators, validation messages will indicate that your method is not a valid call on an attribute. There are several possible causes for this type of error.

Extension plug-in not resolved by Eclipse

Your plug-in might not be properly started and resolved by Eclipse. One quick way to check this is to select:

Help > **About Progress Developer Studio** > **[Installation Details]** > **[Configuration]**

then check the `Plug-in Registry` to see the status of your extensions plug-in:

```
com.corticon.eclipse.studio.deployment.ui (5.4.0.0) "Deployment UI" [Starting]
com.corticon.eclipse.studio.drivers.core (5.4.0.0) "Drivers Core" [Starting]
com.corticon.eclipse.studio.drivers.datadirect.core (5.4.0.0) "Drivers
DataDirect Core" [Starting]
com.corticon.eclipse.studio.extension.core (5.4.0.0) "Vocabulary Extension
Core Plug-in" [Starting]
com.corticon.eclipse.studio.operations.extended.core (5.4.0.0) "Extended
Operators Core Plug-in" [Resolved]
com.corticon.eclipse.studio.extension.ui (5.4.0.0) "Vocabulary Extension UI
Plug-in" [Active]
com.corticon.eclipse.studio.importwizards.core (5.4.0.0) "Import Wizards Core"
 [Starting]
com.corticon.eclipse.studio.junit.core (5.4.0.0) "JUnit Core" [Starting]
```

In this example, the plug-in has been installed and properly `[Resolved]`.

If your plug-in is missing from this list, ensure that you have properly copied it into the Eclipse `/plugins` directory.

Remember to start Eclipse with the `-clean` commandline option. This forces the system to rebuild the bundle cache so that your new plug-in code is recognized.

If your plug-in is present in the list but not marked as `[Resolved]` there may be some problem with the plug-in manifest. Carefully review `MANIFST.MF` to ensure all of your specifications are correct.

Tips for troubleshooting bundle start failures can be found in various Eclipse online forums.

Extension locator file not set up correctly

If your plug-in has been correctly `[Resolved]`, ensure that your extension classes implement the correct marker interface(s) and that `CcExtensionsLocator.lc` is present in the root of your plug-in Jar.

Also, review the `MANIFEST.MF Export-Package` clause carefully. It should appear as follows:

`Export-Package`:

`com.acme.extended.operators`;

Note that in the `Export-Package` clause, you must export `period`(.) to ensure that `CcExtensionsLocator.lc` is properly exported. If this specification is in error or missing, the system will fail to locate your classes.

Extension classes not implementing marker interfaces

Ensure that your classes implement the proper marker interfaces (for example, `ICcStringExtension`); otherwise, the extensions subsystem will ignore your class.

Enabling logging to diagnose extensions issues

When you enable `INFO`-level logging, it will help diagnose extensions discovery issues. To change the log level, edit the brms.properties file at the root of the work directory to set the override:

```
loglevel=INFO
```

When you you restart, the extensions subsystem will log messages as it tries to locate your extension classes.

You might see this type of output if your plug-in fails to properly `export(.)`:

```
CcUtil.getAllLocationsOfFile() .. START
CcUtil.getAllLocationsOfFile() .. astrResourceName = CcExtensionsLocator.lc
com.corticon.extensions.CcExtensions|CcExtensions() loadClassesFromJars() ==
Start
com.corticon.extensions.CcExtensions|CcExtensions() loadClassesFromJars()
asetJarLocations == []
CcUtil|CcUtil.getAllLocationsOfFile() .. START
com.corticon.extensions.CcExtensions|CcExtensions() loadClassesFromDirectories()
 == Start
com.corticon.extensions.CcExtensions|CcExtensions() loadClassesFromDirectories()
 asetDirectoryLocations == []
```

In the log output above, the extensions subsystem is unable to find `CcExtensionsLocator.lc` either because it is missing or not properly exported.

In contrast, here is what you will see in the log if the extensions subsystem is able to find your locator and extension classes:

```
CcUtil.getAllLocationsOfFile() .. START
CcUtil.getAllLocationsOfFile() .. astrResourceName = CcExtensionsLocator.lc
CcUtil.getAllLocationsOfFile() .. lURLLocationPath =
  bundleresource://17.fwk25860399/CcExtensionsLocator.lc
CcUtil.getAllLocationsOfFile() .. lstrProtocol = bundleresource
CcUtil.getAllLocationsOfFile( AFTER RESOLVER ) .. lURLLocationPath =
 jar:file:/C:/Program Files/Progress/Corticon
5.4/Studio/eclipse/plugins/com.corticon.eclipse.studio.operations.extended.core

 _5.4.0.jar!/CcExtensionsLocator.lc
CcUtil.getAllLocationsOfFile(1) .. lstrPath =
 file:\C:\Program Files\Progress\Corticon
5.4\plugins\com.corticon.eclipse.studio.operations.extended.core
 _5.4.0.jar!\CcExtensionsLocator.lc
CcUtil.getAllLocationsOfFile(2) .. lstrPath =
 file:\C:\Program Files\Progress\Corticon
5.4\plugins\com.corticon.eclipse.studio.operations.extended.core
 _5.4.0.jar!\CcExtensionsLocator.lc
CcUtil|CcUtil.getAllLocationsOfFile() .. END =
 [file:\C:\Program Files\Progress\Corticon
5.4\plugins\com.corticon.eclipse.studio.operations.extended.core
 _5.4.0.jar!\CcExtensionsLocator.lc]
com.corticon.extensions.CcExtensions|CcExtensions() loadClassesFromDirectories()
 == Start
com.corticon.extensions.CcExtensions|CcExtensions() loadClassesFromDirectories()
  asetDirectoryLocations == []
```

As shown above, `getAllLocationsOfFile` is able to find your extensions. This is a positive sign and if you see such output in the log, it is very likely that your extensions will work properly.

11

Documenting your extensions

In order for your extensions to be visible in the Rule Operators tree view, they must be properly documented in a special file named `ExtendedOperators.operationsmodel`. This file is in EMF/XMI format and can be maintained using either a text editor an EMF-generated editor supplied with Corticon Studio. Refer to the example `ExtendedOperators.operationsmodel` file in:

`com.corticon.eclipse.studio.operations.extended.core`

During system initialization, the system will attempt to locate `ExtendedOperators.operationsmodel` on the Java class path; if the system finds this file, it will automatically merge the documented extended operators into the Rule Operators tree view.

`ExtendedOperators.operationsmodel` supports internationalization. The object model permits localization of folder names, extended operator names, parameter names and tooltips. This is accomplished via EMF "multi-valued" attributes, which are essentially lists (arrays) of values. Each "slot" in the array contains a particular localization (i.e., a string expressed in one of the supported languages).

The first localization in each "slot" is special and is referred to as the "base" localization. For class and method names, the "base" localizations must match the class and method names in your java extension classes.

The root object of the model (`ExtendedOperatorSet`) contains a list of Java `LanguageCode` instances that defines the set of languages supported. The order of language codes is important and must match the order of localizations expressed elsewhere in the file.

The system will merge typed extensions into the Rule Operator tree at the end of the built-in operators. For example, String Attribute Extended Operators will be appended to the "String" data type node of the Rule Operator tree after the built-in String operators.

The system will append Stand Alone extensions to the end of the Rule Operator tree. The Stand Alone extensions will be contained within a special folder whose name is declared in `ExtendedOperators.operationsmodel` (see *ExtendedOperatorSet.standAloneFolderName*). In our example, the name of the Stand Alone extensions folder is simply `Extended Operators`.

Our example `ExtendedOperators.operationsmodel` contains English descriptions of operators, and is set up to handle Japanese localizations as well, although the only Japanese localization specified is the Stand Alone folder name.

Refer to example `ExtendedOperators.operationsmodel` file as a guide documenting your own extensions.

12

Precompiling ruleflows with service call-outs

As described in the "*Deploying Corticon Ruleflows*" chapter of the *Server Integration & Deployment Guide*, you can pre-compile Ruleflows prior to deploying them. When pre-compiling Ruleflows with Service Call-outs (SCO), it is important to ensure the following:

1. Place the SCO JARs on the Deployment Console's classpath. This is accomplished by editing the `deployConsole.bat file` located in `[CORTICON_HOME]\Server` (or your chosen installation directory).
2. Once the Ruleflow has been compiled to an `.eds` file using the Deployment Console, add your `.eds` file, along with any other JARs required by the SCO, to the Server directory which holds `CcServer.jar`. In the default Corticon Server installation, this is `[CORTICON_HOME]\Server\pas\corticon\webapps\axis\WEB-INF\lib`.