ACTICO Platform - Modeler

Rules Java Integration Guide

Version 8.1.11

ACTICO GmbH

actico
Smarter Decisions

www.actico.com

Rules Java Integration Guide: Version 8.1.11

Rules Java Integration Guide, Version 8.1.11

Table of Contents

Chapter 1. Introduction

This manual is the "Rules Java Integration Guide". Its targeted audience are Java developers who want to learn how to integrate rules into Java applications, how to configure the Java code generation, how to make rules use an existing Java object model and how to extend ACTICO Modeler with custom functions, actions or services.

Chapter 2. Configuring Rule Projects and Code Generation

The ACTICO Modeler is based on the Eclipse platform. Consequently, many of the concepts of the Eclipse workbench can be found in ACTICO Modeler. For example, ACTICO Modeler uses the notion of projects to organize the files that make up rule models. Rule projects are a special kind of projects specifically tailored for rule modeling.

The ACTICO Modeler also makes extensive use of the JDT, the Java Development Tools of Eclipse. This is because prior to execution ACTICO Modeler always converts rules into Java code, using the JDT to compile and launch the Java rule code.

This chapter will explain the configuration options for rule projects and the code generation process.

If you are not familiar with the Eclipse workbench or the Java Development Tools, you may find it useful to first read the Workbench User Guide or the Java Development User Guide that come with the Eclipse platform. Both manuals can be found in the online help (menu item Help > Help Contents).

2.1. Concepts

2.1.1. Rule Project

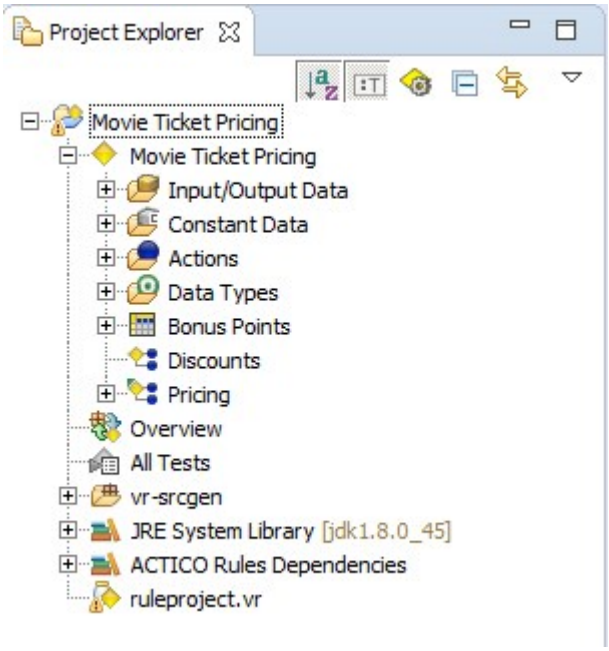
A rule project is a special kind of project in the workspace, represented by this icon ().

A rule project contains (at least) one rule model. Rule models are represented by a yellow diamond ().

It also contains a file named `ruleproject.vr`, which specifies unique identifiers for the rule project and a list of dependencies (other rule projects or libraries).

A rule project is always also a Java project, because whenever the rules are transformed into Java code, this code has to be compiled by the Eclipse Java compiler. The source folder named `vr-srcgen` is where the generated rule code goes. So whenever code is generated from the rule model, the generated code in this folder will be updated. The Eclipse Java compiler will then take the newly generated code and compile it. (The name and location of the source folder can be configured differently, `vr-srcgen` is just the default.)

By default, a rule project has the ACTICO Rules Dependencies classpath container on the project build path in order to compile and execute the rule code.



Related Concepts.

- Rule Project Identifiers, Dependencies and `ruleproject.vr`
- Rule Project Classpath
- Java Code Generation

Related Tasks.

- Setting Up the Classpath
- Configuring Java Code Generation

Related References.

- Java Code Generator Tab

2.1.2. Rule Project Identifiers, Dependencies and `ruleproject.vr`

A rule project can define dependencies to other rule projects or libraries (JARs) in order to reuse the rules, data types, services or other elements defined there. All dependencies of a rule project are defined in the file `ruleproject.vr`.

This file has two purposes:

- specify a unique identifier for the rule project itself, consisting of a Group Id, Artifact Id and a Version.
- specify the dependencies of the rule project, i.e. Group Ids, Artifact Ids and Versions of other rule projects (or rule artifacts, or libraries) required by this rule project

The Group Id, Artifact Id and Version are the coordinates that uniquely identify the rule project in the workspace of ACTICO Modeler, but also on the ACTICO Execution Server when deployed there.

It is best practice to have the Artifact Id relate to the project name. Per default, the Artifact Id is the same as the project name (with spaces replaced by dashes). The Group Id can either also relate to the project name or be used to indicate a group of interrelated projects, e.g. a domain name, department name or application name, like `com.example` or `marketing` or `CRM-system`.

The Version number should follow a numbering scheme with two or three components separated by dots and an optional qualifier separated by a dash, e.g. `0.0.1`, `2.0`, `3.1.6`, `1.0-ALPHA`, `7.11.3-SNAPSHOT`. The `SNAPSHOT` identifier is commonly used to identify a non-released version, i.e. a version that is currently under development and which has not been released to production.

Technically the `ruleproject.vr` file is a Maven `pom.xml`, and if you like, you can actually name the file `pom.xml`. You may want to do this, if you are using Maven in general and would like to use a Maven project layout. The main difference is that ACTICO Modeler provides a specific editor for the `ruleproject.vr` file, while for editing a `pom.xml` you must rely on other editors and tools.

Maven is a Java based build and project management tool that contains concepts and tools for the entire software development cycle. For more information about Maven visit the website http://maven.apache.org.

In addition to rule projects in the workspace, ACTICO Modeler also allows dependencies to other Maven projects and libraries that are made available through Maven repositories. By default ACTICO Modeler is able to download and use all libraries as dependencies that are contained in the public central repository of Maven at http://repo1.maven.org/maven2.

Related Concepts.

- Rule Project

Related Tasks.

- Setting Up the Classpath

Related References.

- Libraries Preferences

2.1.3. Rule Project Classpath

The classpath of a rule project must include all the classes necessary for the rule code to be compiled and - if you want to test or execute the rules - all the classes necessary for execution.

These are:

- the generated rule code (Java sources in `vr-srcgen`)
- the Rules Runtime libraries
- the Database Integrator Runtime library (if you are using Database Integrator)
- the rule project dependencies, i.e. other rule projects whose rule models are being reused (via Reuse Package definitions)
- implementation classes (JavaBeans) of the data types imported as a Java Type Library
- classes containing the implementations of custom functions, custom action types, custom service types
- any libraries your code depends upon

The recommended way to configure the classpath is to define dependencies in `ruleproject.vr`. The dependencies specified in the `ruleproject.vr` file are automatically reflected in the ACTICO Rules Dependencies classpath container, which every rule project has per default.

Related Concepts.

- Rule Project
- Rule Project Identifiers, Dependencies and `ruleproject.vr`
- Rules Runtime Library

Related Tasks.

- Setting Up the Classpath

Related References.

- Libraries Preferences

2.1.4. Java Code Generation

Rule models and the rules within are converted to Java code by the Java code generator. That generated code can be directly integrated into Java applications and the rules can be called through the Rule Execution API.

The Java code generator can be configured differently for every rule model on the Java Code Generator tab in the Properties. Most importantly, you can specify the output folder of the code generation. Per default this is a folder named `vr-srcgen` - the default Java source folder of a rule project. Whenever the code generator puts the generated source into that folder, Eclipse (more specifically the JDT) picks up the source files and compiles them into Java bytecode.

Related Concepts.

- Rule Project
- Rule Execution API

Related Tasks.

- Generating Java Code for a Rule Model
- Configuring Java Code Generation

Related References.

- Java Code Generator Tab

2.1.5. Copy-by-Reference / Copy-by-Value Semantics

The Java code generator of ACTICO Modeler can be configured to generate code that uses either copy-by-reference or copy-by-value semantics. This setting determines what happens when a structured value (an object) is assigned to a data element.

The setting is available on the Java Code Generator tab of the rule model and can be overridden for every rule package.

The Java Code Generator tab is not available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective.

In Java, whenever an identifier referring to an object is assigned to another identifier, that reference is merely copied to the other identifier. The object itself is not copied and remains unchanged. This results in two identifiers pointing to the same object. This behaviour is called copy-by-reference semantics and appears very natural to Java developers. However, it may be difficult to grasp for some non-technical users. They often expect a behaviour coined copy-by-value semantics.

In copy-by-value semantics not just the reference to the object, but the object itself is copied. Java does that only for primitive types (non-objects). In fact, Java has very limited support for copying objects. Objects may implement `Cloneable` to indicate and support copying, but very often they don't. So Java usually uses serialization and deserialization to create copies of objects. This is an expensive operation and works only if all participating objects are `Serializable`.

For example, let's assume a data type `Customer` which is a structure consisting of several attributes, including a `name`. There are two data elements `custA` and `custB` of type `Customer`. When one data element is assigned the value of the other data element, you may see different behaviour, depending on the configured semantics.

Copy-by-Reference Semantics	**Copy-by-Value Semantics**

`1 custA.name := "John"`
`2 custB := custA`
`3 custB.name := "Peter"`
	`1 custA.name := "John"`
	`2 custB := custA`
	`3 custB.name := "Peter"`

`custA.name` is now also `"Peter"`, simply because after line 2 `custA` and `custB` are referring to the same object. This is the default behaviour of Java.	`custA.name` is still `"John"`, because in line 2 a copy of `custA` was created and assigned to `custB`. Copying objects during assignments is not the default behaviour of Java and usually means a significant performance hit. However, ACTICO Modeler supports this behaviour in order to prevent some common misconceptions by the non-Java user.

The copy-by-reference or copy-by-value semantics also apply to a Repeat element when it is iterating over a collection ("Repeat for each of multiple elements"), because this also includes a (repeated) assignment of the respective element from the collection to the data element holding the current element.

Depending on the number and sizes of collections, and the size of their elements being processed by the rules, you may observe a dramatic performance decrease with copy-by-value semantics compared to copy-by-reference semantics. Sometimes rules may become a hundred or even a thousand times slower.

Therefore, it is recommended to use copy-by-reference semantics whenever possible, especially when the rules are "collection-intensive".

Related References.

- Java Code Generator Tab

2.1.6. Java Type Settings

Rule models have their own type system with the following basic data types: `Integer`, `Float`, `String`, `Date`, `Time`, `Timestamp` and `Any`.

The Java language has a much more fine-grained type system. For example, a data element of rules type `Float` may be represented by many different Java types (`float`, `double`, `java.lang.Float`, `java.lang.Double` or `java.math.BigDecimal`). ACTICO Modeler allows to specify additional settings for each data element to determine the exact Java type to be used for the code generation.

These settings can be done on the Java Implementation tab which is available for input/output data elements, internal data elements, constant data elements, as well as for attributes of structures.

Related Tasks.

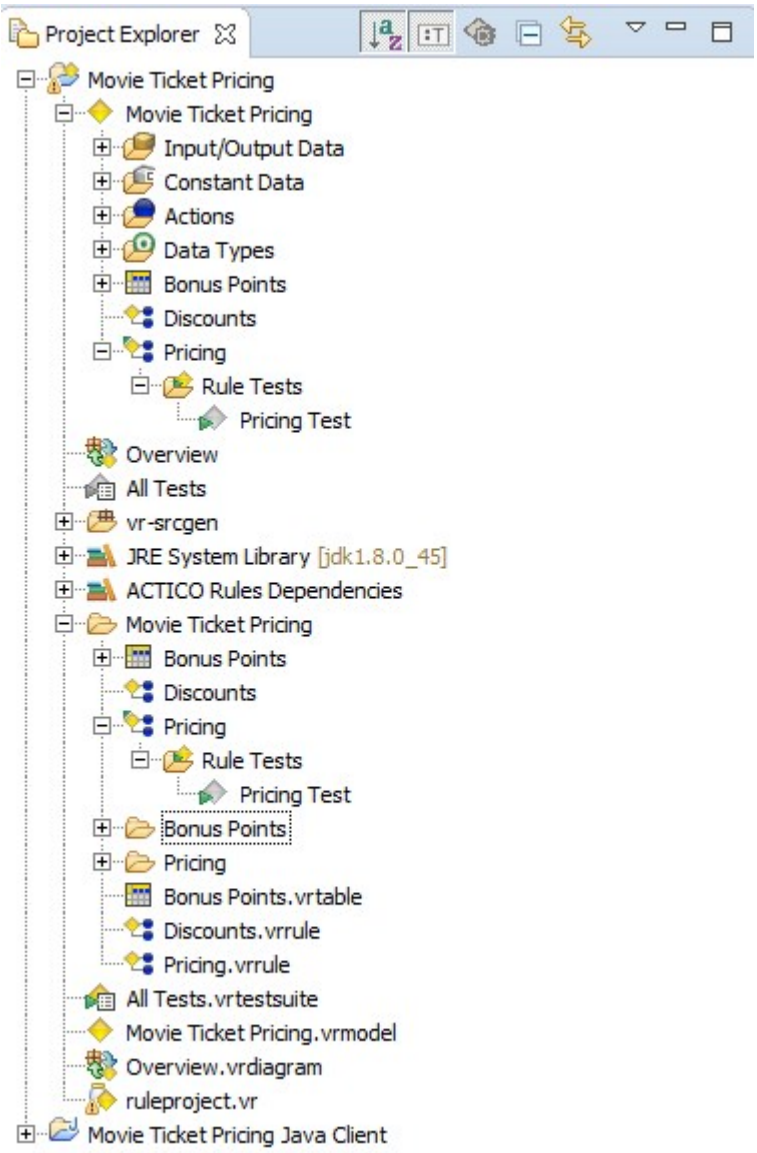
- Selecting Java Implementations for Data Elements

Related References.

- Java Implementation Tab

2.1.7. Rule Model File Layout

Rule models are stored on the file system as many individual files. The file representing the model itself has an extension of `.vrmodel`. Next to the `.vrmodel` file is a folder of the same name which contains the files that represent the rule model contents, like rules and packages.



The rule packages defined in the rule model are reflected on the file system by folders of the same name. This is mainly done so that the file system somewhat reflects the structure of the rule model. This is useful when

it comes to putting the files into a version control system (like CVS), because there the user will have to work with files and not with rule models. So having the file system follow the same structure as the rule model will make it easier to understand which part of the rule model is stored in what file.

The following table lists all the rule model elements which are stored in files of their own, along with the corresponding file extensions used.

Be sure to never change the names and locations of these files directly on the file system. That could render the rule models useless, i.e. ACTICO Modeler may ultimately not be able to load the rule model.

On the other hand, when you make structural changes to the rule model using the Rule or Project Explorer, like moving a rule from one package to another, or renaming a package, these changes will immediately be reflected on the filesystem automatically.

Table 2.1. File extensions of ACTICO Modeler rule model files

Rule Model Element	File extension
Rule Model	`.vrmodel`
Flow Rule	`.vrrule`
Decision Table	`.vrtable`
Rule Test (test input data and expected results)	`.vrtest`
Test Suite	`.vrtestsuite`
Execution Results, Test Results, Statistics	`.vrexecution`
Rule Package	`.vrpackage` (+ folder named like the package)

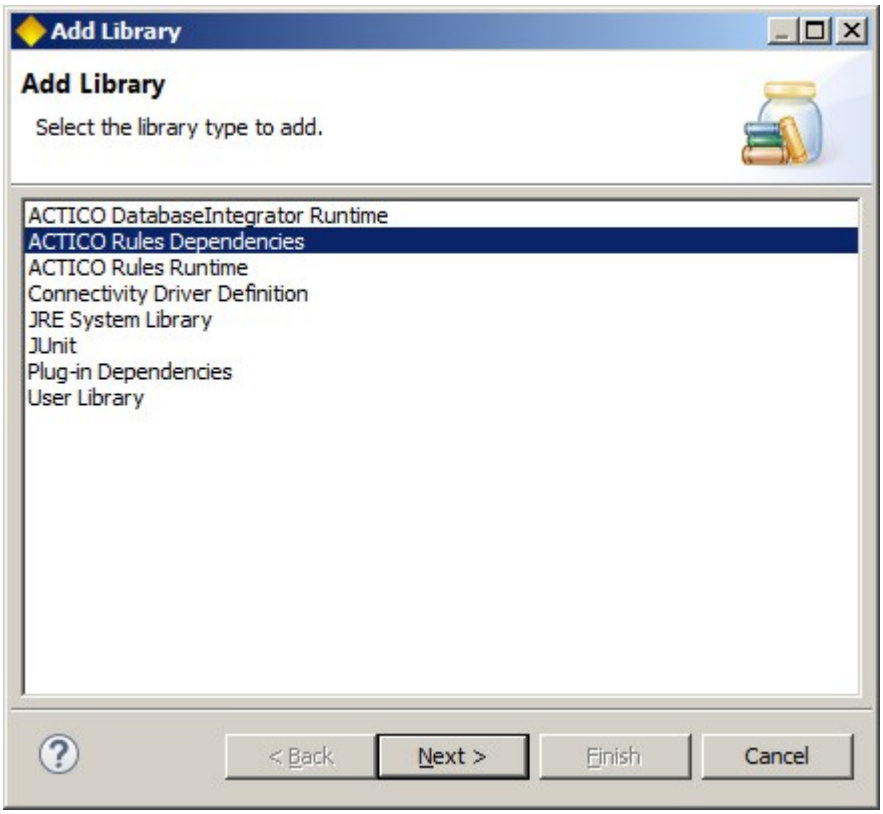
2.2. Tasks

2.2.1. Setting Up the Classpath

By default, a rule project has the ACTICO Rules Dependencies classpath container, which contains all the dependencies defined in `ruleproject.vr`. Consequently, you configure the classpath by defining the dependencies in `ruleproject.vr`. For further information, see the task *Defining Rule Project Dependencies* in the *Modeling Guide*.

You can manually add the ACTICO Rules Dependencies classpath container like this:

1. Right-click on the project and select Properties.
2. Go to the Java Build Path page.
3. Switch to the Libraries tab and click on Add Library... to add the ACTICO Rules Dependencies to the build path.



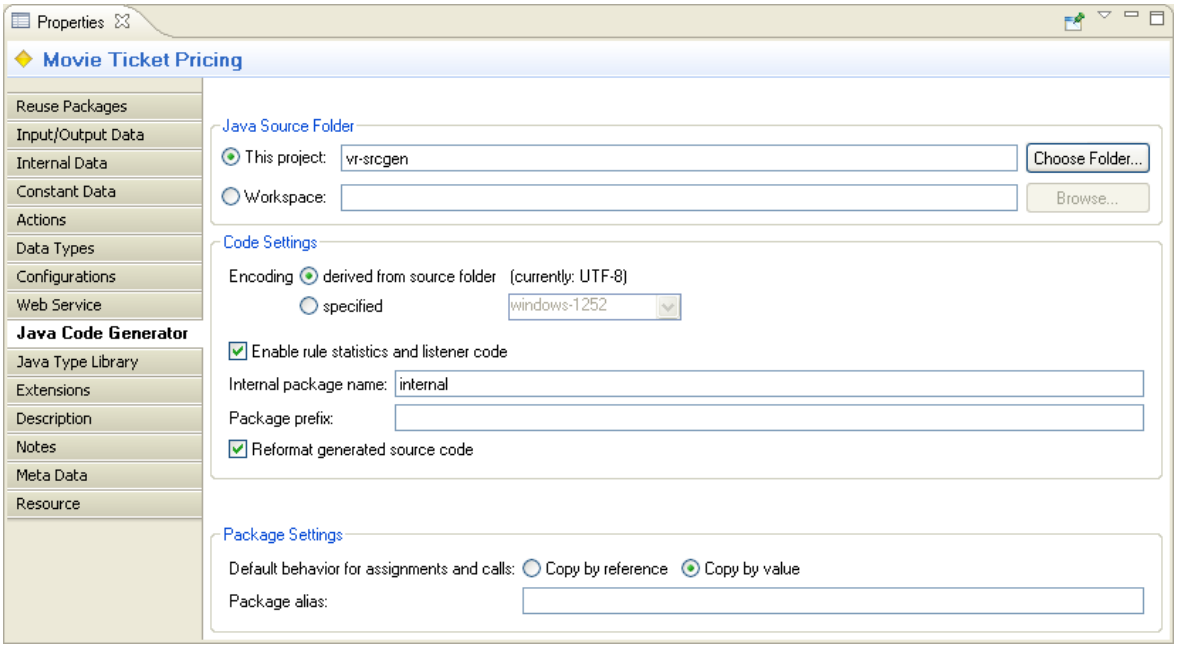
Related Concepts.

- Rule Project Classpath

2.2.2. Configuring Java Code Generation

General Java code generator settings are made at the rule model level. Some additional settings can also be defined for each rule package.

1. Select a rule model or rule package where you want to change the code generator settings.
2. In the Properties view activate the Java Code Generator tab.



The Java Code Generator tab is not available in the Rule Modeling perspective. Switch to another perspective, e.g. the Rule Integration perspective to have access to the technical tabs.

3. Depending on whether a rule model or just a rule package is selected, different settings are displayed and can be edited.

4. Make the appropriate changes to the settings. These will be used next time the code is generated.

Related References.

- Java Code Generator Tab

2.2.3. Generating Java Code for a Rule Model

You can manually generate the Java code for a rule model in the Project Explorer. Here the code can be generated either fully or incrementally.

The Java code is automatically generated incrementally, when for a rule included in the relevant rule model a test is executed.

To fully generate the code for a rule model, do one of the following:

1. Right-click on the rule model (or on any included rule package or rule) and select Generate Rule Code (fully) from the context menu.

The menu item Generate Rule Code (fully) is not available in the Rule Modeling perspective. Switch to another perspective, e.g. the Rule Integration perspective, if necessary.

or

2. Select the rule model (or on any included rule package or rule) and press Ctrl+Shift+C.

To incrementally generate the code for a rule model, do one of the following:

1. Select the rule model (or on any included rule package or rule) and and click on  in the header of the Project Explorer.

The button  is not available in the Rule Modeling perspective. Switch to another perspective, e.g. the Rule Integration perspective, if necessary.

or

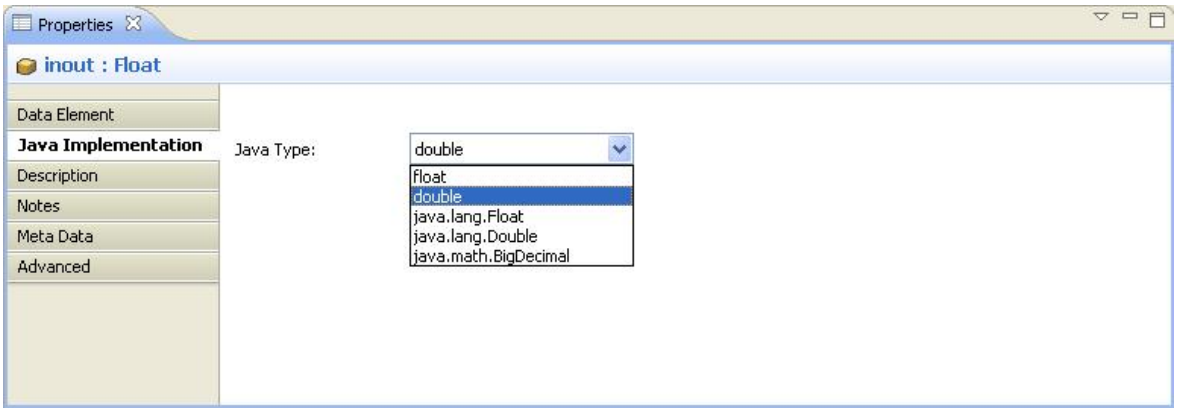
2. Select the rule model (or on any included rule package or rule) and press Ctrl+Shift+I.

2.2.4. Selecting Java Implementations for Data Elements

In order to define the Java type to be used for a specific data element or attribute, do the following:

1. Select a single data element (input/output, internal, constant, or attribute of a structure) in the Project Explorer.

2. In the Properties view activate the Java Implementation tab.



The Java Implementation tab is not available in the Rule Modeling perspective. Switch to another perspective, e.g. the Rule Integration perspective to have access to the technical tabs.

3. Here you can select the Java implementation (Java type) to be used in the generated code.

Related Concepts.

- Java Type Settings

Related References.

- Java Implementation Tab

Chapter 3. Executing Rules

Rules can be directly integrated into Java applications and called using the Rule Execution API. The following sections explain how to setup a Java project accordingly and how to use the Rule Execution API to call rules, including all the necessary data exchange between the calling application and the rules.

3.1. Concepts

3.1.1. Rule Execution API

The Rule Execution API is the API provided by the generated code and the Rules Runtime library. With this API you can

- create session and request objects for rule execution
- set input data for rules
- execute rules
- read results produced by rules (data and actions)
- dynamically query a rule's input and output interface
- activate/deactivate and configure the generation of statistics
- register and unregister listeners for rule execution events
- exchange action or service implementations

The API for calling rules is available in a specific and a generic version. The specific API is part of the generated code and provides type-safe getter and setter methods for the data and rules in the model. The method names use the actual names of the data elements and rules in the model and therefore are specific to each model. That is why this API is called the "specific" Rule Execution API.

In addition to the specific API there is also a generic Rule Execution API provided by the Rules runtime. The methods of the generic API are always the same, independently of the rule model. The methods use mostly strings to specify the names of data elements and rules.

Related Tasks.

- Calling Rules with the specific Rule Execution API
- Calling Rules with the generic Rule Execution API

3.1.2. Rule Session (ISession)

The rule session is a concept of the Rule Execution API. Before a client can call rules, it must first establish a rule session. The rule session is associated with the client and can be used to perform any number of individual rule requests. The lifecycle of a rule session is usually as follows:

1. The client obtains a rule session (`ISession`) from the `RuleSessionFactory`
2. The client creates any number of individual rule requests from the session and executes a rule each time
3. The client finally closes the rule session

A rule session should only be used by one thread at a time. (The rule code itself is thread-safe, i.e. multiple threads can use their individual rule sessions and rule requests to execute the same rules).

Often, a rule session only consists of just one request. For these cases, the Rule Execution API provides convenience methods that implicitely create a session, create a rule request, perform rule execution and finally close the session.

The rule session can be configured to collect statistics at different levels of granularity.

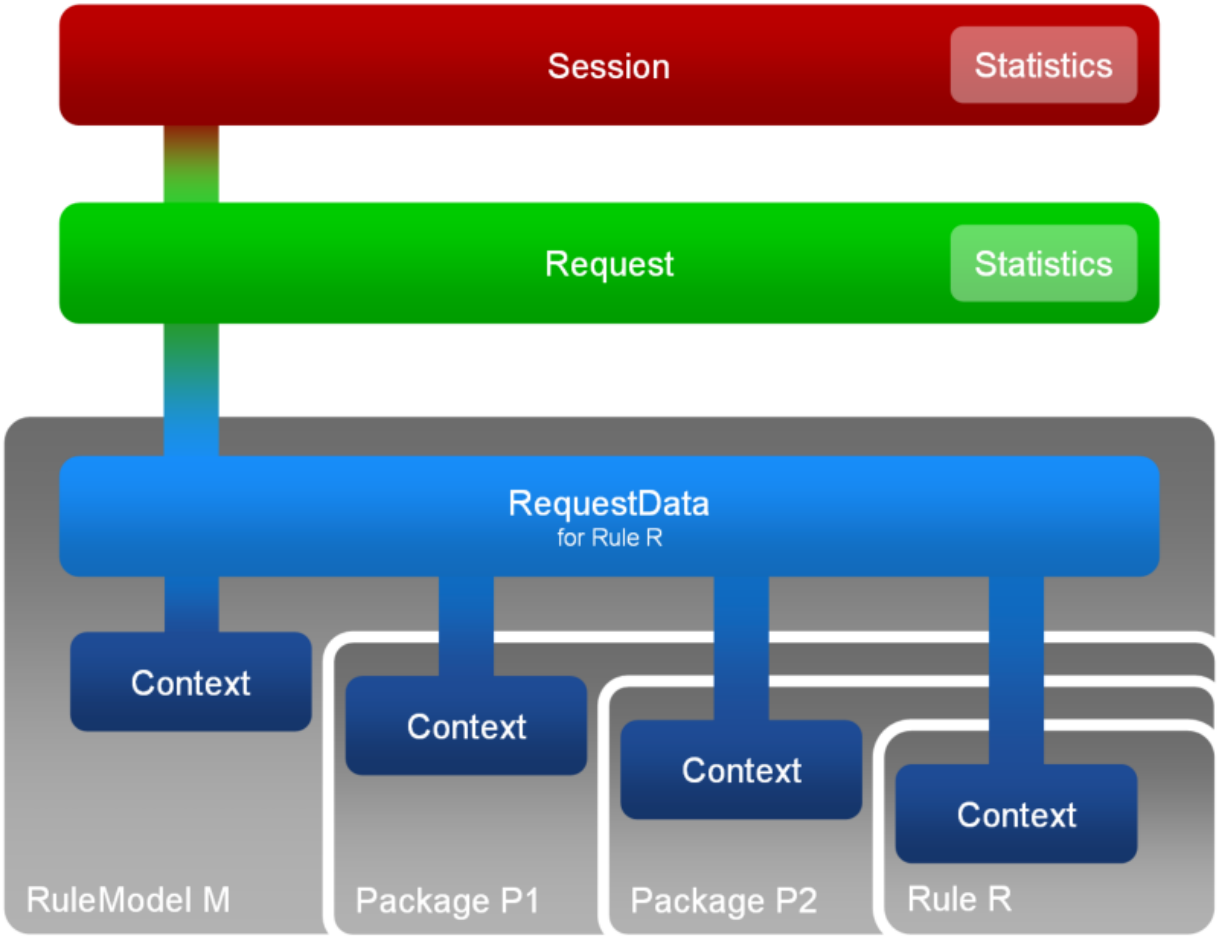


Figure 3.1. Relations between Session, Request, RequestData and Contexts

Related Concepts.

- Rule Request (IRequest)

Related Tasks.

- Calling Rules with the specific Rule Execution API

3.1.3. Rule Request (IRequest)

A rule request is a concept of the Rule Execution API and represents an individual call to a specific rule. A rule request always happens as part of a rule session. The rule request usually consists of five steps:

1. The client obtains a rule request object (`IRequest`) from the rule session (`ISession`)
2. The client sets up the input data (request data) for the rule and associates it with the rule request
3. The rule is called and fully executed, possibly triggering other rules
4. The rule returns control back to the client
5. The client inspects the results produced by the rules

These steps can be repeated any number of times during a rule session.

Like the rule session, a rule request can also be configured to collect statistical information.

Related Concepts.

- Rule Session (ISession)
- Request Data (IRequestData)

Related Tasks.

- Calling Rules with the specific Rule Execution API

3.1.4. Request Data (IRequestData)

Request data objects are part of the Rule Execution API. A request data object (`IRequestData`) is the data container for an individual rule request. The request data object associates all data objects that are accessible to the rule being executed. The data available to a rule consists of all data elements defined at the rule level, plus all data elements defined on the rule packages above and finally on the rule model.

Request data objects are specific for each rule. Everytime a rule calls another rule, a new request data object is created and reassociated with the same or new context objects, depending on whether the rules share the context.

A client calling a rule during a rule request has to do the following:

1. Create a new request data object specific for the rule to be called. This can be done by the specific `createXXXRequestData()` methods or the generic `createRequestData()` method on the rule model.
2. Populate the request data object by accessing the individual context objects and setting the input data elements accordingly using the specific `setXXX()` methods. There is also a generic `setInParameter()` method to do this without having to access the individual contexts.
3. Perform rule execution and passing along the request data object (and the request). This is done via the specific `performXXX()` methods or the generic `perform()` method on the rule model.

Related Concepts.

- Context
- Rule Request (IRequest)

Related Tasks.

- Calling Rules with the specific Rule Execution API

3.1.5. Context

Context objects are part of the Rule Execution API. A context represents the data that is defined on an individual rule, rule package or rule model. In other words, a context is a specific level within the hierarchy of the rule model. A rule can see its own context, the contexts of all packages it is contained within, and the rule model context.

A context object holds the values of all data elements defined in that individual context. It provides getters and setters for these values. Context objects are collected into request data objects.

A context is uniquely identified by a path-notation, which consists of the names of the contexts starting from the rule model and downwards to the context in question, e.g.

```
/My Rule Model/My Package/My Sub Package/My Rule
```

All names are separated by a slash ("/"). These paths are used in many places in the rule code and API to identify rule models, rule packages or rules.

Related Concepts.

- Request Data (IRequestData)

Related Tasks.

- Calling Rules with the specific Rule Execution API

3.1.6. Rules Runtime Library

The *Rules Runtime library* is a set of Java libraries that are required for the rule code to execute. The runtime library also contains the interfaces and classes of the Rule Execution API. Applications that want to integrate rules have to make the runtime library available on their classpath.

The Rules Runtime library consists of just one specific JAR named `visualrules-runtime-x.y.z.jar`. Where `x.y.z` stands for the concrete version, e.g. *visualrules-runtime-6.7.0.jar*. The Rules Runtime requires further JARs which are automatically added.

Related Concepts.

- Rule Execution API
- Database Integrator Runtime Library

Related Tasks.

- Setting up the Java Project

3.1.7. Database Integrator Runtime Library

The *Database Integrator Runtime library* is similar to the Rules Runtime library, as it contains interfaces and classes required to use the API of *Database Integrator* extension (available separately).

The library consists of one jar: `visualrules-dbcruntime-x.y.z.jar`, where `x.y.z` stands for the version of the *Database Integrator*. Note that *Rules Runtime library* is needed as well in order to execute rule code.

Related Concepts.

- Rules Runtime Library

3.1.8. Configurations

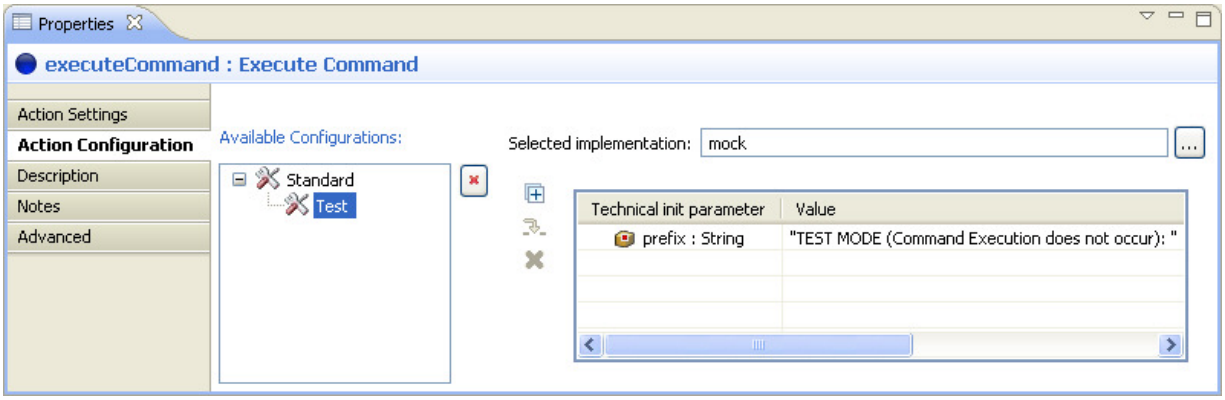
The Rule Execution API and the generated code support the concept of "named configurations". Multiple different configurations can be defined for any rule model and later be activated at runtime via the Rule Execution API. This is mainly used to switch between different behaviours for actions (it can also be used to configure resources like database connections differently for different environments).

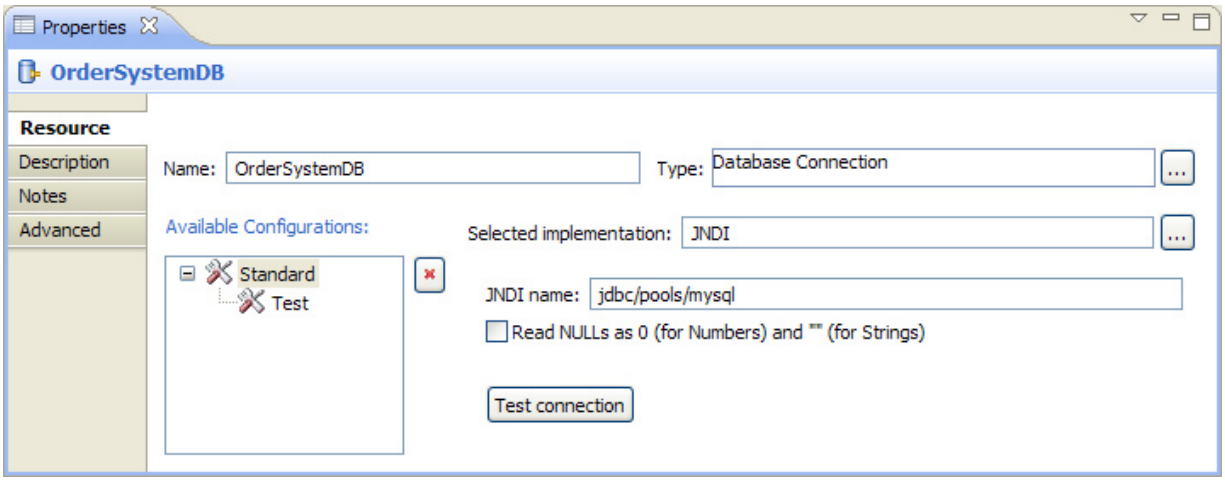
Actions can perform complicated tasks and interface with other systems, e.g. a database. Custom Java code can be introduced as custom action types to extend their capabilities. This is a very powerful feature, but at the same time it can make testing of rules harder, because actions need to behave differently or interface with other systems during testing than in production.

For example, an action may send e-mails to customers, but during testing we would rather prefer to write a log file of all the e-mails that would have been sent instead of actually sending them.

ACTICO Modeler supports these scenarios through the concept of multiple named configurations for actions. Individual actions can be configured differently for production, test, or other environments. The desired configuration can be selected during rule execution or rule testing simply by specifying its name. This will automatically reconfigure all actions that have a special configuration set up, and may even switch the action implementation entirely.

Configurations are defined on the Configurations tab of a rule model. Once multiple configurations have been defined, they can be referred to on the Action Configuration tab for actions or on the Resource tab for resources.





Related References.

- Configurations Tab

Related Tasks.

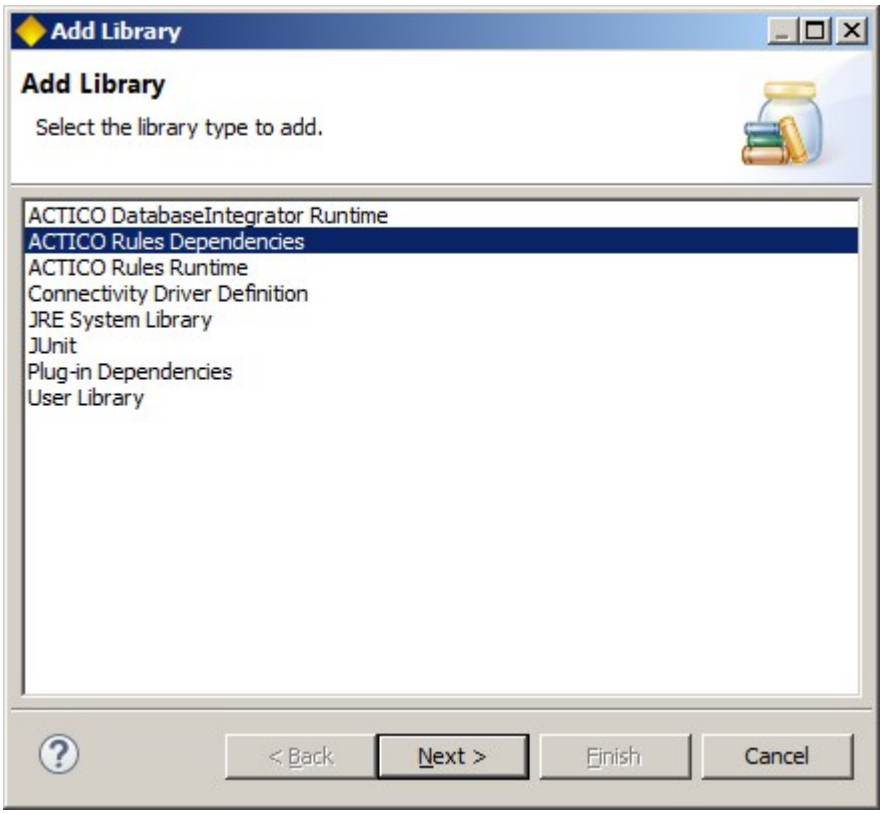
- Selecting Configurations

3.2. Tasks

3.2.1. Setting up the Java Project

A Java project that wants to call rules needs to have the rule code and the Rules Runtime library available on its classpath. The runtime library is a set of JAR files that are available as a built-in library. The rule code is usually referenced simply by adding the corresponding rule project to the build path. Here is how this is done:

1. Configure the Java build path of the Java project that should call the rules. Right-click on the project and select Properties.
2. Go to the Java Build Path page and select the Projects tab.
3. Click on Add... and select the project(s) containing the rule code (usually this is the rule project itself unless code generation was configured otherwise).
4. Switch to the Libraries tab and click on Add Library... to add the ACTICO Rules Runtime to the build path.



Related Concepts.

- Java Code Generation
- Rules Runtime Library

3.2.2. Calling Rules with the specific Rule Execution API

The specific Rule Execution API is part of the generated code and provides type-safe getter and setter methods (and others) for the data and rules in the model. The method names use the actual names of the data elements and rules in the model and therefore are specific to each model. That is why this API is called the "specific" Rule Execution API.

The names of rules, data elements etc. in the generated code match the names in the rule model. However, if the model name contains a blank, this will be replaced by an underscore, because Java doesn't support blanks in identifiers. For example, a rule model named "Example Rules" will appear as "Example_Rules" in the Java method names (see below).

There are additional cases where the name is changed. For example, names starting with numbers are preceded with an underscore to make them valid identifiers in Java.

To illustrate the specific Rule Execution API, we are using the rule model named "Example Rules" shown here. The rule model itself does not do anything useful but it is a suitable example to illustrate the API.

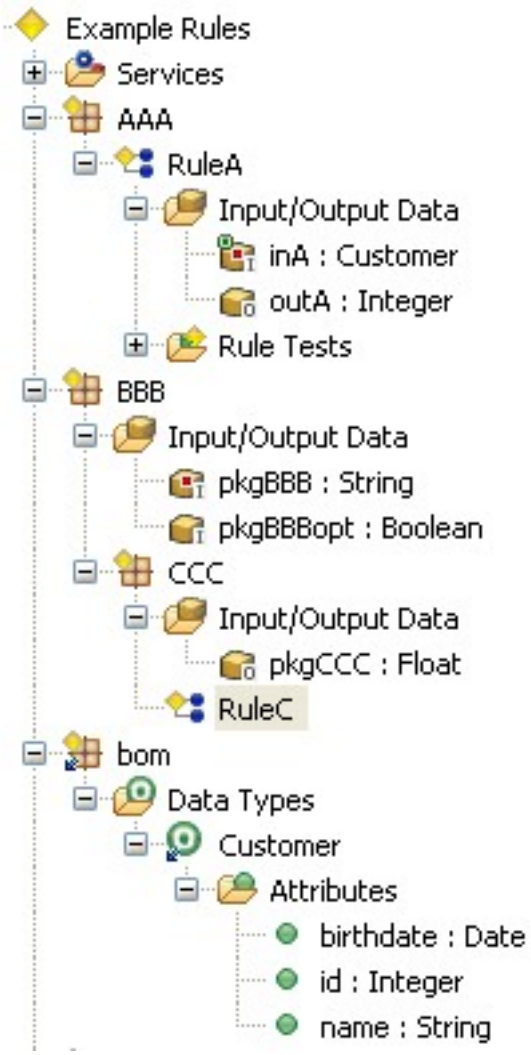


Figure 3.2. "Example Rules" model

- The rule model contains the rule packages "AAA" and "BBB". Rule package "BBB" has a sub-package named "CCC".
- The rule package "AAA" contains a rule named "RuleA".
- The rule "RuleA" has an input and an output data element named "inA" and "outA", respectively.
- The rule package "BBB" defines the input data elements named "pkgBBB" and "pkgBBBopt".
- The rule package "CCC" defines an output data element named "pkgCCC".
- The rule package "CCC" contains a rule named "RuleC".

The following code illustrates how a Java application can call the rules "RuleA" and "RuleC" in this model.

The examples here are taken from the "Rule Execution API Example" that comes with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace. See `ExampleSpecificAPI.java` to find the following code.


```
// create a rule session
ISession session = RuleSessionFactory.createSession();

try
{
   // --- call of rule "RuleA" ------------------------------------------------------
   // create a rule request
   IRequest request = session.createRequest();

   // create a request data object for the rule "RuleA"
   IRuleARequestData requestDataA = Example_RulesRuleModel.createRuleARequestData();

   // access the context of "RuleA" (contains all data elements defined on that rule)
   IContextRuleA contextRuleA = requestDataA.getContextExample_RulesAAARuleA();

   // set the input data element "inA"
   Customer customer = new Customer();
   customer.setId(7);
   customer.setName("Volker");
   customer.setBirthdate(new SimpleDateFormat("dd.MM.yyyy").parse("14.01.1970"));

   contextRuleA.setInA(customer);

   // execute the rule "RuleA"
   Example_RulesRuleModel.performRuleA(request, requestDataA);

   // read the output data element "outA"
   int outA = contextRuleA.getOutA();
   System.out.println("RuleA has returned a value of " + outA);

   // --- call of rule "RuleC" ------------------------------------------------------
   // create another rule request
   request = session.createRequest();

   // create a request data object for the rule "RuleC"
   IRuleCRequestData requestDataC = Example_RulesRuleModel.createRuleCRequestData();

   // access the context of rule model "Example Rules" (contains all global data elements)
   IContextExample_Rules contextExample_Rules = requestDataC.getContextExample_Rules();

   // access the context of rule package "BBB"
   IContextBBB contextBBB = requestDataC.getContextExample_RulesBBB();

   // set the input data element "pkgBBBopt"
   contextBBB.setPkgBBBopt(true);

   // set the input data element "pkgBBB"
   contextBBB.setPkgBBB("This is the value for input data element 'pkgBBB'");

   // execute the rule "RuleC"
   Example_RulesRuleModel.performRuleC(request, requestDataC);

   // access the context of rule package "CCC"
   IContextCCC contextCCC = requestDataC.getContextExample_RulesBBBCCC();

   // read the output data element "pkgCCC"
   double pkgCCC = contextCCC.getPkgCCC();
   System.out.println("RuleC has set the value of pkgCCC to " + pkgCCC);
}
finally
{
   // close the rule session
   session.closeSession();
}
```

Initially the client creates a rule session. The rule session can be used by a client to perform any number of individual rule requests using any number of rule models. In this case, only two rule requests are performed within the session.
The client creates a request object for the rule call.

In order to call a rule, the client needs to obtain a suitable request data object. The request data object contains the data that is passed from the application to the rules and vice versa. Because every rule has a different set of input/output data elements, the request data objects are specific for every rule. The rule model class contains methods to create a corresponding request data object for each rule. These methods are called `createXXXRequestData()` where `XXX` is the rule name. In this case the client creates the request data object for rule "RuleA".
Within the request data object there are multiple context objects, each for every level within the hierarchy of the rule model. These contexts are accessible via getter methods named `getXXX()` where `XXX` is the concatenated names of the contexts starting from the rule model down the hierarchy. This quasi fully-qualified path syntax for the context getter methods is due to the fact that two contexts may have the same name within a hierarchy (e.g. a rule package "ABC" may contain another package of the same name).
The context object provides the getter and/or setter methods for the data elements. The input data element "inA" of "RuleA" is of type "Customer". Accordingly, the client creates a "Customer" object and uses it as the value for "inA".
Now, the rule is executed using the corresponding `performXXX()` method on the rule model class. These methods are named `performXXX()` where `XXX` is the rule name, but only if the rule name is unique within the rule model. If the name is not unique, `XXX` is replaced by the concatenated names of the contexts starting from the rule model down the hierarchy including the rule name.

The `performXXX()` methods come in two flavors. One has two parameters: the request object and the request data object. This is the normal way to execute a rule and is used in the example above.

Additionally, there is a shorthand version of each `performXXX()` method which omits the request and only takes the request data object. This method implicitely creates a rule session, creates one rule request, executes the rule and finally closes the session again. This is useful in cases where a rule session only consists of one rule request. In that case, the client does not have to create the session nor the request itself, but only the request data.

After the rule is executed, the client can look at the results using the context objects' getter methods for the output data elements and actions. The client reads the value of the output element "outA" via the getter on the rule's context object.
Now, the client wants to call "RuleC". This involves the same steps as before, including the creation of a request object.
The rule "RuleC" itself does not define any input/output data but it lives in packages that do so. Accordingly, the client must access the corresponding context objects to set the value of the element "pkgBBBopt" or the element "pkgBBB" defined in the rule package "BBB".
After "RuleC" is executed the client accesses the context of package "CCC" to read the value of output data element "pkgCCC".
The client has to close the rule session after he is done with it.

All classes and interfaces in the `vrinternal` packages are not considered API and may change in an incompatible way in future releases.

Related Concepts.

- Rule Execution API
- Java Code Generation
- Rules Runtime Library

Related Tasks.

- Setting up the Java Project
- Calling Rules with the generic Rule Execution API

3.2.3. Calling Rules with the generic Rule Execution API

To illustrate the generic Rule Execution API, we are using the rule model named "Example Rules" shown here. The rule model itself does not do anything useful but it is a suitable example to illustrate the API.

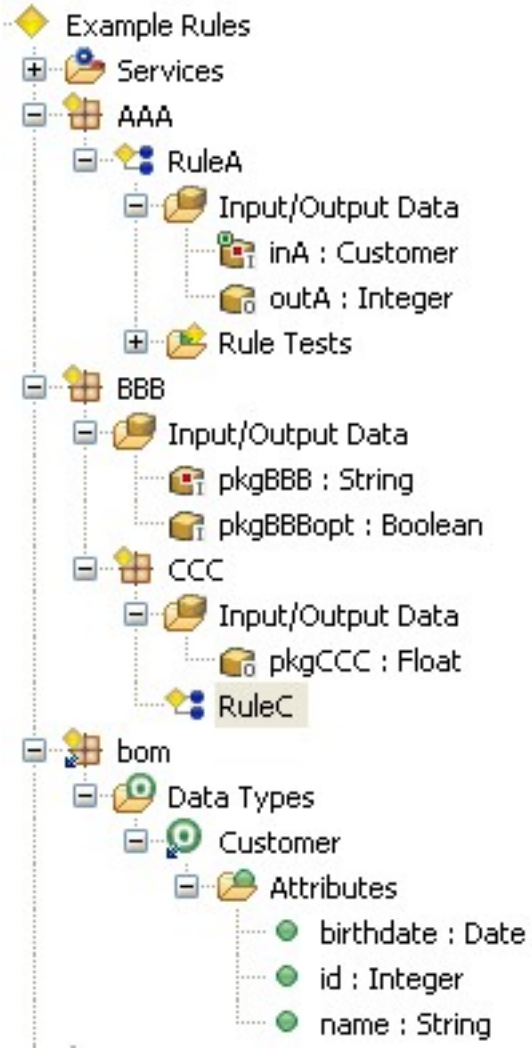


Figure 3.3. "Example Rules" model

- The rule model contains the rule packages "AAA" and "BBB". Rule package "BBB" has a sub-package named "CCC".
- The rule package "AAA" contains a rule named "RuleA".
- The rule "RuleA" has an input and an output data element named "inA" and "outA", respectively.
- The rule package "BBB" defines the input data elements named "pkgBBB" and "pkgBBBopt".
- The rule package "CCC" defines an output data element named "pkgCCC".
- The rule package "CCC" contains a rule named "RuleC".

The following code illustrates how a Java application can call the rules "RuleA" and "RuleC" in this model.

The example here is taken from the "Rule Execution API Example" that comes with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace. See `ExampleGenericAPI.java` to find the following code.


```
// create a rule session
ISession session = RuleSessionFactory.createSession();

try
{
   // --- call of rule "RuleA" ------------------------------------------------------
   // create a rule request
   IRequest request = session.createRequest();

   // create an instance of the rule model class to access the generic API
   IRuleModel ruleModel = new Example_RulesRuleModel();

   // create a request data object for the rule "RuleA"
   IRequestData requestData = ruleModel.createRequestData("RuleA");

   // set the input data element "inA"
   Customer customer = new Customer();
   customer.setId(7);
   customer.setName("Volker");
   customer.setBirthdate(new SimpleDateFormat("dd.MM.yyyy").parse("14.01.1970"));

   requestData.setInParameter("inA", customer);

   // execute the rule "RuleA"
   ruleModel.perform(request, requestData);

   // read the output data element "outA"
   Integer outA = (Integer) requestData.getOutParameter("outA");
   System.out.println("RuleA has returned a value of " + outA);

   // --- call of rule "RuleC" ------------------------------------------------------
   // create another rule request
   request = session.createRequest();

   // create a request data object for the rule "RuleC" using a fully-qualified path
   requestData = ruleModel.createRequestData("/Example Rules/BBB/CCC/RuleC");

   // set the input data element "pkgBBBopt"
   requestData.setInParameter("pkgBBBopt", Boolean.FALSE);

   // set the input data element "pkgBBB"
   requestData.setInParameter("pkgBBB", "This is the value for input data element
'pkgBBB'");

   // execute the rule "RuleC"
   ruleModel.perform(request, requestData);

   // read the output data element "pkgCCC"
   Double pkgCCC = ((Double)requestData.getOutParameter("pkgCCC"));
   System.out.println("RuleC has set the value of pkgCCC to " + pkgCCC);
}
finally
{
   // close the rule session
   session.closeSession();
}
```

Initially the client creates a rule session. The rule session can be used by a client to perform any number of individual rule requests using any number of rule models. In this case, only two rule requests are performed within the session.
The client creates a request object for each rule call.

In order to access the generic API, an instance of the rule model class must be created.

In order to call a rule, the client needs to obtain a suitable request data object. The request data object contains the data that is passed from the application to the rules and vice versa. Because every rule has

a different set of input/output data elements, the request data objects are specific for every rule. The rule model class contains a method `createRequestData()` to create a request data object for a specific rule. The name of the rule is passed to the method.

If the rule name is unique within the whole rule model, it is sufficient to just use the rule name. Otherwise a fully-qualified path must be used which consists of the names of the contexts starting from the rule model down the hierarchy, separated by slashes. This is illustrated further down in the lisitng for "RuleC". The input data element "inA" of "RuleA" is of type "Customer". Accordingly, the client creates a "Customer" object and uses it as the value for "inA".

The request data object provides the `setInParameter()`, `getOutParameter()`, `setInParameters()`, `getOutParameters()`, `getAction()` and `getActions()` methods to get and set the values of all input/output data elements and actions available to the rule. This includes all input/output elements and actions defined anywhere in the context hierarchy leading to the rule. Thus, when a client is using the generic API, it does neither have to access the context objects nor does it have to know exactly where elements are defined within the context hierarchy.
Now, the rule is executed using the `perform()` method of the rule model. The method receives the request and request data object (and by looking at the latter it knows which rule to call).

The `perform()` method comes in two flavors. One has two parameters: the request object and the request data object. This is the normal way to execute a rule and is used in the example above.

Additionally, there is a shorthand version of the `perform()` method which omits the request and only takes the request data object. This method implicitely creates a rule session, creates one rule request, executes the rule and finally closes the session again. This is useful in cases where a rule session only consists of one rule request. In that case, the client does not have to create the session nor the request itself, but only the request data.

After the rule is executed, the client can look at the results using the `getOutParameter()`, `getOutParameters()` or `getActions()` methods. The client reads the value of the output element "outA" via the `getOutParameter()` method.
Now, the client wants to call "RuleC". This involves the same steps as before, including the creation of a request object and the creation of a request data object, this time for "RuleC". Here we use the fully-qualified path to "RuleC" which consists of the names of the contexts starting from the rule model down the hierarchy, separated by slashes. The fully-qualified path syntax is only necessary to disambiguate two or more rules of the same name in a rule model.
The generic `setInParameter()` method is used to set the values for the data elements "pkgBBBopt" and "pkgBBB" defined in package "BBB".
After "RuleC" is executed the client reads the value of output data element "pkgCCC".

The client has to close the rule session after he is done with it.

When you are working with the generated rule code and the Rule Execution API, you should not use any classes from the `internal` or `vrinternal` packages. These are not considered API and may change in an incompatible way in future releases.

Related Concepts.

- Rule Execution API
- Java Code Generation
- Rules Runtime Library

Related Tasks.

- Setting up the Java Project
- Calling Rules with the specific Rule Execution API

3.2.4. Calling Rules using the Rule Execution Template

The class `RuleExectionTemplate` takes care about the rountine work around a rule call. Together with the class `AbstractRuleExecuitionAdvisor` it defines the typical procedure of a rule call.

Furthermore the template manages the reuse of a `ISession` instance belonging to the current thread.

The following code illustrates the usage of `RuleExectionTemplate`.

The example here is taken from the "Rule Execution API Example" that comes with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace. See `ExampleRuleExecutionTemplate.java` to find the following code.

```
 1       // Create the rule template. In this example the rule template will
 2       // create one session per perform call.
 3       RuleExecutionTemplate ruleExecutionTemplate = new RuleExecutionTemplate();
 4
 5       // Execute a rule
 6       IRequestData ruleExecutionResult = ruleExecutionTemplate.perform(new
AbstractRuleExecutionAdvisor()
 7       {
 8          public IRuleModel getRuleModel()
 9          {
10             return Example_RulesRuleModel.INSTANCE;
11          }
12
13          public String getRuleIdentifier()
14          {
15             return "/Example Rules/AAA/RuleA";
16          }
17
18          public void populateValues(IRequestData requestData) throws Exception
19          {
20             // set the input data element "inA"
21             Customer customer = new Customer();
22             customer.setId(7);
23             customer.setName("Volker");
24             customer.setBirthdate(new SimpleDateFormat("dd.MM.yyyy").parse("14.01.1970"));
25
26             requestData.setInParameter("inA", customer);
27          }
28       });
29
30       int outA = ((Integer)ruleExecutionResult.getOutParameter("outA")).intValue();
31       System.out.println("RuleA has returned a value of " + outA);
```

Creation of the `RuleExecutionTemplate`. In that example the default constructor was taken. So a new session will be created for each rule call. That session will be closed automatically after the rule call.

It's possible to pass a `true` to the parameterized constructor. In that case one session instance per thread will be created and reused. That mode requires the call of the static method `closeReusedSession()` or the instance methode `closeSession()` after the last rule call is done.

Having closed the reusable session of a thread a new instance will be created if necessary.
The `AbstractRuleExecutionAdvisor` is created as a anonymous inner class. It provides the information about the rule to call which are required by the `RuleExecutionTemplate`. Furthermore it offers the possibility to customize session and request creation. The population of the request data can be alse done.

When you are working with the generated rule code and the Rule Execution API, you should not use any classes from the `internal` or `vrinternal` packages. These are not considered API and may change in an incompatible way in future releases.

Related Concepts.

- Rule Execution API
- Java Code Generation
- Rules Runtime Library

Related Tasks.

- Calling Rules with the specific Rule Execution API
- Calling Rules with the generic Rule Execution API

3.2.5. Calling Rules based on Java Interface

As alternative to using Rule Execution API, it is also possible to call a Java interface. A rule must implement this interface but can then be used as any other interface. Keep in mind, that not all configuration options of Rule Execution API are possible with this approach. This is part of the broader concept described in Integration into the Java Service Landscape. The steps necessary to call a rule based on a Java interface can be found in Service Call.

3.2.6. Inspecting the Inputs and Outputs (Meta Data) of a Rule

The Rule Execution API allows a client to query a rule for the names and types of input and output parameters and actions. This way a client does not have to have any prior knowledge about the data interface of a rule. This allows for a rather loose coupling between a rule client and the actual rule code.

The method `getMetaData()` for querying the meta data about input/output data and actions can be found on the `IRequestData` interface, which is implemented by all the generated `RequestData` classes.

The `getMetaData()` method returns an object of type `IRequestDataMetaData` which has the methods `getParameterInfos()` and `getActionNames()`.

The `getActionNames()` method returns an array of all action names available in the request data object.

The `getParameterInfos()` method returns an array of `IParameterInfo` objects, each containing info about a parameter. This includes the name, direction (input and/or output), type information and whether it is a required parameter or not (see methods `getName()`, `isInput()`, `isOutput()` and `isRequired()`). The type information can be retrieved by the `getTypeDefinition()` method which returns an `ITypeDefinition` object.

`ITypeDefinition` has a method `getTypeClass()` which returns a `java.lang.Class` that indicates the Java type of the parameter. (Note: this is the type to be used for the second argument of the `setInParameter(String, Object)` method of `IRequestData`.)

If the parameter is a collection, the `getTypeClass()` method will accordingly return `java.util.Collection`, `java.util.List` or `java.util.Set`.

In that case it is possible to cast `ITypeDefinition` to `ICollectionTypeDefinition` and then use `getComponentTypeClass()` to determine the expected type of the collection's elements.

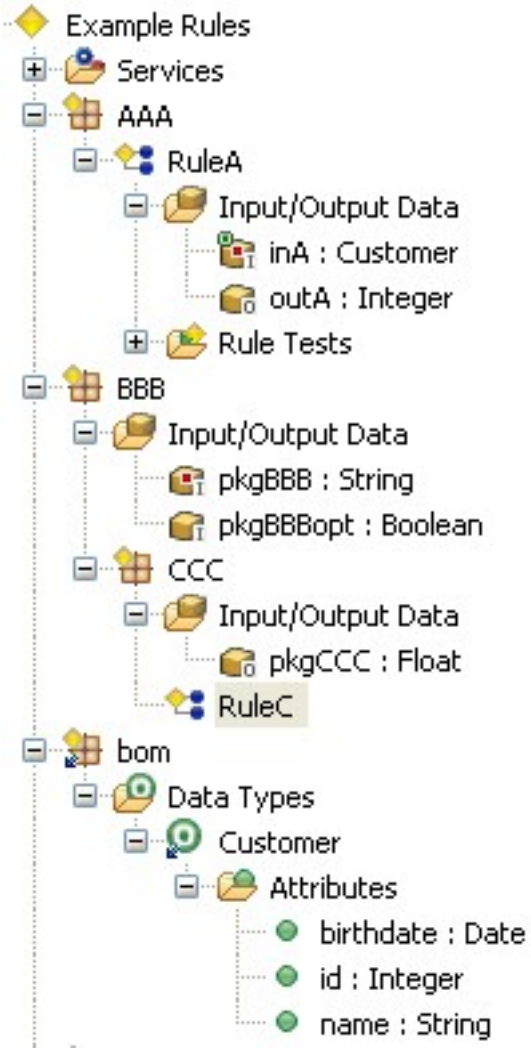
If the parameter is a map, the `ITypeDefinition` can be cast to `IMapTypeDefinition`. The method `getKeyTypeClass()` will return the type for the map's keys, while `getValueTypeClass()` will return the type of the map's values.

Finally, structured types are instances of `IStructuredTypeDefinition`. This contains information of the structure and it's attributes as defined in the model. This extends the possibilities of the current `IParameterInfo` by enabling the assignment of values based on the names in the model instead of using the Java Reflection API.

The structure is described by an `IStructureInfo` and each attribute by an `IAttributeInfo`, which again carries an `ITypeDefinition` to inspect the corresponding type in Java.

The following listing shows a client inspecting the parameters of "RuleA" in the example model.

The examples here are taken from the "Rule Execution API Example" that comes with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace. See `ExampleMetaData.java` to find the following code.




```
  public static void main(String[] args)
  {
     try
     {
        // create an instance of the rule model class to access the generic API
        IRuleModel ruleModel = new Example_RulesRuleModel();

        // create a request data object for the rule "RuleA"
        IRequestData requestData = ruleModel.createRequestData("RuleA");

        // retrieve meta data
        IRequestDataMetaData metaData = requestData.getMetaData();

        // look at the info for each input and output parameter ...
        IParameterInfo[] parameterInfos = metaData.getParameterInfos();

        for (IParameterInfo parameterInfo : parameterInfos)
        {
           // ... and print the info to the screen
           System.out.println(parameterInfoToString(parameterInfo));
        }
     }
     catch (Exception e)
     {
        e.printStackTrace();
     }
  }

  /**
   * Converts an {@link IParameterInfo} object into a human-readable string.
   */
  private static String parameterInfoToString(IParameterInfo parameterInfo)
  {
     StringBuffer infoString = new StringBuffer();
     String direction = "";

     // inspect parameter direction: either "in", "out" or "in/out"
     if (parameterInfo.isInput() && parameterInfo.isOutput())
     {
        direction = "in/out";
     }
     else if (parameterInfo.isInput())
     {
        direction = "in    ";
     }
     else if (parameterInfo.isOutput())
     {
        direction = "out   ";
     }

     infoString.append(direction);
     infoString.append(" ");

     // inspect if the parameter is required
     infoString.append(parameterInfo.isRequired() ? "*" : " ");

     // inspect name
     infoString.append(parameterInfo.getName());

     // inspect type
     infoString.append(" : ");
     ITypeDefinition typeDefinition = parameterInfo.getTypeDefinition();

     infoString.append(typeDefinition.getTypeClass().getName());

     // inspect additional type info in case it is a collection ...
     if (typeDefinition instanceof ICollectionTypeDefinition)
     {
        ICollectionTypeDefinition collectionTypeDefinition = (ICollectionTypeDefinition)
typeDefinition;

        infoString.append("<" + collectionTypeDefinition.getComponentTypeClass().getName() +
">");
     }
     // ... or a map
     else if (typeDefinition instanceof IMapTypeDefinition)
     {
        IMapTypeDefinition mapTypeDefinition = (IMapTypeDefinition) typeDefinition;

        infoString.append("<" + mapTypeDefinition.getKeyTypeClass().getName() + ", "
```

This program lists all parameters of "RuleA" as follows. Shown is the direction, required flag (*), name and type information. For structured parameters, it also describes the attributes of the structure.

```
out     outA : int
in     *inA : bom.Customer
        ------------------------
        structure Customer
          id : int
          name : java.lang.String
          birthdate : java.util.Date
        ------------------------
```

Related Tasks.

- Inspecting custom Meta Data
- Calling Rules with the generic Rule Execution API

3.2.7. Inspecting custom Meta Data

Rule models, rule packages, rules, input/output data elements, constant data element, enumerations, exceptions, structures and attributes can have custom meta data associated with them. The Rule Execution API gives access to that meta data at runtime.

To inspect custom meta data of a rule model, rule package or rules, use the `getMetaDataInfo()` method of `IRuleModel`. Specify the model path of the rule model, rule package or rule, whose custom meta data you would like to retrieve. `getMetaDataInfo()` returns an `IMetaDataInfo`. Method `getMetaData()` of `IMetaDataInfo` returns a `Map` which contain the keys and values of the custom meta data.

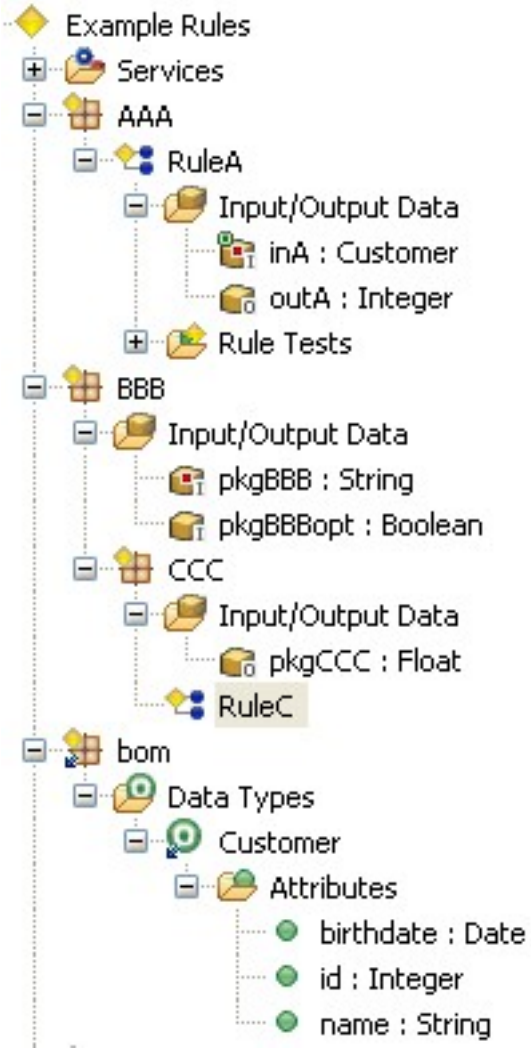
In order to inspect custom meta data associated with an input/output data element of a rule, use the `getMetaData()` method of `IParameterInfo`. Custom meta data of structures, exceptions or enumerations can be accessed with the method `getMetaData()` of `IStructuredTypeDefinition`. Custom meta data of attributes is accessed with the method `getMetaData()` of `IAttributeInfo`.

See Inspecting the Inputs and Outputs (Meta Data) of a Rule on how to access `IParameterInfo`, `IStructuredTypeDefinition`, and `IAttributeInfo`.

Finally, custom meta data of constant data elements is retrieved with `getMetaData()` of `IConstantInfo`. Use method `getConstantInfos()` of `IRuleModel` to retrieve infos about all the constants in the rule model.

You will see use of this API in the following example:

The examples here are taken from the "Rule Execution API Example" that comes with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace. See `ExampleCustomMetaData.java` to find the following code.




```
   public static void main(String[] args)
   {
      try
      {
         // create an instance of the rule model class to access the generic API
         IRuleModel ruleModel = new Example_RulesRuleModel();

         // inspect custom meta data for RuleA
         Map ruleMetaData = ruleModel.getMetaDataInfo("/Example Rules/AAA/
RuleA").getMetaData();
         System.out.println("RuleA \tmeta data: " + ruleMetaData + "\n");

         // create a request data object for the rule "RuleA"
         IRequestData requestData = ruleModel.createRequestData("RuleA");

         // retrieve meta data
         IRequestDataMetaData metaData = requestData.getMetaData();

         // look at the info for each input and output parameter ...
         IParameterInfo[] parameterInfos = metaData.getParameterInfos();

         for (IParameterInfo parameterInfo : parameterInfos)
         {
            // ... and print the info to the screen
            System.out.println(parameterInfoToString(parameterInfo));
         }
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }

   /**
    * Converts an {@link IParameterInfo} object into a human-readable string.
    */
   private static String parameterInfoToString(IParameterInfo parameterInfo)
   {
      StringBuffer infoString = new StringBuffer();
      String direction = "";

      // inspect name
      infoString.append(parameterInfo.getName()).append(":\t");

      // inspect custom meta data
      infoString.append("meta data: ");
      infoString.append(parameterInfo.getMetaData());

      ITypeDefinition typeDefinition = parameterInfo.getTypeDefinition();

      // inspect additional type info in case it is a collection ...
      if (typeDefinition instanceof ICollectionTypeDefinition)
      {
         ICollectionTypeDefinition collectionTypeDefinition = (ICollectionTypeDefinition)
 typeDefinition;

         infoString.append("<" + collectionTypeDefinition.getComponentTypeClass().getName() +
 ">");
      }
      // ... or a map
      else if (typeDefinition instanceof IMapTypeDefinition)
      {
         IMapTypeDefinition mapTypeDefinition = (IMapTypeDefinition) typeDefinition;

         infoString.append("<" + mapTypeDefinition.getKeyTypeClass().getName() + ", "
            + mapTypeDefinition.getValueTypeClass().getName() + ">");
      }
      // ... or it is structured
      else if (typeDefinition instanceof IStructuredTypeDefinition)
      {
         // inspect structure
         IStructuredTypeDefinition structuredTypeDefinition = (IStructuredTypeDefinition)
 typeDefinition;
         IStructureInfo structureInfo = structuredTypeDefinition.getStructureInfo();

         infoString.append("\n\t------------------------\n\tstructure
 ").append(structureInfo.getName()).append("\n\t");

         // inspect custom meta data of the structure
         infoString.append("meta data: ");
```

As you can see in the output, the example rule model has some custom meta data associated with `RuleA`, `outA`, `inA`, structure `Customer` and its attribute `id`.

```
RuleA   meta data: {layout.font.bold=true, layout.render=false}

outA:   meta data: {layout.render=true}
inA:    meta data: {layout.render=false, layout.color=green}
        ------------------------
        structure Customer
        meta data: {render.background=grey}
          id:   meta data: {id=true}
          name: meta data: {}
          birthdate:    meta data: {}
        ------------------------
```

Related Tasks.

- Inspecting the Inputs and Outputs (Meta Data) of a Rule
- Calling Rules with the generic Rule Execution API

3.2.8. Handling Exceptions

Rules have built-in support to produce and handle exceptions. Flow rules may explicitly contain Return Exception elements that will cause a Java exception to be thrown at runtime. Flow rules can also contain Handle Exception elements that allow to catch and handle these exceptions. If an exception is not handled by the rules themselves, it will be thrown back to the rule client calling the rules.

This is the reason why the `IRuleModel.perform()` methods have a `throw Exception` clause. This is not very specific, but effectively really any checked or unchecked exception may be thrown, because ACTICO Modeler can import any of your domain specific exception classes and these can be thrown with Return Exception elements.

Apart from the exceptions that are explicitely thrown by the rules, there is also the possibility for any kind of runtime exception to occur, like `NullPointerException`, `ArithmeticException` etc. All these unchecked exceptions can also be handled by Handle Exception elements or - if they are not handled by the rules - will be propagated back to the client.

A client should always make sure that the rule session (`ISession`) is properly closed even when exceptions occur. So it is often a good idea to have the `session.closeSession()` statement in a `finally` block.

3.2.9. Configuring Statistics

Rules can produce statistical information when they are executed. When statistics are turned on, a rule execution trace will be produced. This trace information represents the sequence of rule execution down to the level of single rule elements (called 'nodes' in the API). In the end, the statistics will contain counters for each rule element that indicate how often each rule element was executed. These counters can then be visualized in the rule editor, like this:

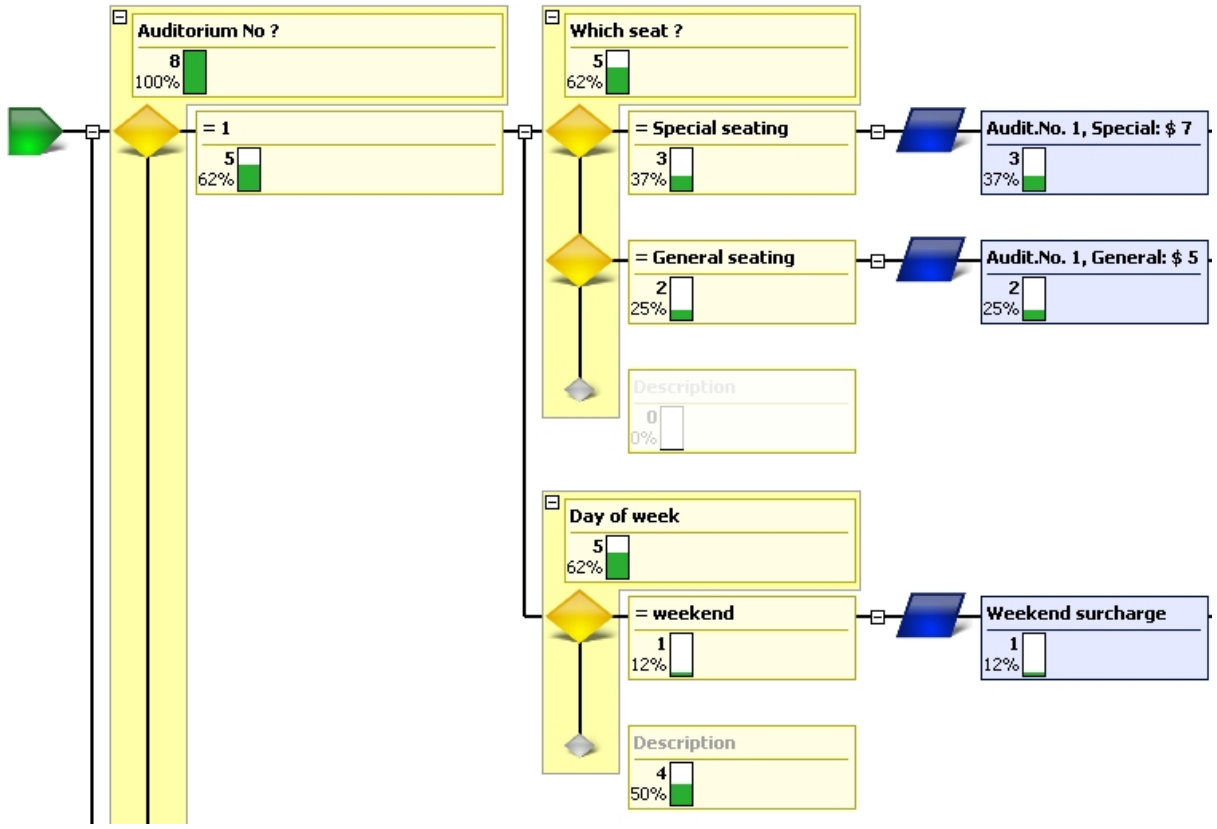


Figure 3.4. Rule editor displaying statistics

A client can activate statistics on either the rule session (`ISession`) or on the rule request (`IRequest`) or both. This is done with the `setStatisticsEnabled()` and `setStatisticsLevel()` methods available on both `ISession` and `IRequest`.

The statistical information is retrieved via the `getStatistics()` method from either `ISession` or `IRequest`. It will return either `ISessionStatistics` or `IRequestStatistics`. Statistics on request level reflect what happened in that single request, while statistics on the session will be an accumulation of what happened during the whole session.

Statistics can be produced at different levels of detail. The method `setStatisticsLevel()` on `ISession` or `IRequest` allows to set the desired level. `ISession` and `IRequest` contain predefined constants for these levels. Look at the following table to see which information is collected for the different levels.

Table 3.1. Statistics Levels

Level (from ISession or IRequest)	Collected Statistics
QUIET	Only session/request start and end time (default)
LOW	additionally execution counters for each rule element
MEDIUM	additionally total execution time for each rule element
HIGH	additionally minimum and maximum execution times for each rule element

The code generator can be configured to omit the code necessary for producing statistics (see Java Code Generator tab for the rule model). If rule code was generated like this, activating statistics will have no effect.

Related Tasks.

- Persisting Statistics

3.2.10. Persisting Statistics

Statistical information collected on the session or request is available via the `getStatistics()` methods on both `ISession` and `IRequest`.

The Rule Execution API provides a utility class named `StatisticsPersitenceUtil` that can be used to save statistics in an XML format. The following listing illustrates how this is done.

The example here is taken from the "Rule Execution API Example" that comes with ACTICO Modeler. See `ExampleStatistics.java`.

```
ISession session = RuleSessionFactory.createSession();

// ... execute some rules ...

// retrieve the statistics from the session
ISessionStatistics statistics = session.getStatistics();

// create a file
File statisticsFile = new File("statistics.xml");
fileWriter = new FileWriter(statisticsFile);

// write the statistics XML into the file
StatisticsPersistenceUtil.persistStatistics(statistics, fileWriter);
```

Related Tasks.

- Configuring Statistics

3.2.11. Selecting Configurations

When multiple configurations have been defined on the rule model, a client may select the desired configuration by calling the `setActiveConfigurationName(String name)` method on `ISession`. This must happen prior to calling any rule within that session. Consequently, once a session has started and rules have been executed, the configuration cannot be changed anymore.

The following listing shows how a client selects the configuration named "Test" and then executes a rule.

The example here is taken from the "Execute Command Action" example that comes with ACTICO Modeler. See `ExampleChangeConfiguration.java`.

```
// create a session
ISession session = RuleSessionFactory.createSession();

// set the active configuration to "Test"
// this will cause the mock implementation to be used, like it is configured in the rule
model
session.setActiveConfigurationName("Test");

// create the request
IRequest request = session.createRequest();

// ... execute rules ...
```

Related Concepts.

- Configurations

3.2.12. Activating, deactivating and exchanging Actions

A client can reconfigure any actions defined in the rule model prior to execution. This includes deactivation of specific actions, so that whenever the action is fired, the corresponding action implementation will not be executed (however, the action value itself will still be set to `true`).

Another option for a client is to exchange the action implementation (`IAction`) for specific actions, so that an implementation different from the one configured in the rule model will be executed when the action is fired.

There are two ways to achieve this. The first is by using the `registerAction()` method of the session (`ISession`), the second is by registering an `IActionDefinitionFactory`, which is explained later on.

The first parameter to the `registerAction()` method is the action id, which is a model path for the action. The model path is the concatenation of all context names (rule model, rule packages, rule) separated by slashes (/) indicating where the action is defined, e.g. `"/My Rule Model/My Package1/My Package2/My Rule/my_action"`. The model paths of actions are available as constants in the context interfaces (`IContextXXX`).

The second parameter is an `IActionDefinition`. An action definition is not the actual action implementation, but contains the information about which implementation (`IAction`) should be instantiated when the action is fired. `ActionUtil` can be used to create such an `IActionDefinition`.

The following example illustrates how a client can exchange an action implementation.

The examples here are taken from the "Execute Command Action" example that comes with ACTICO Modeler. See `ExampleReplaceAction.java` and `ExampleDeactivateAction.java`.

```
import de.visualrules.example.action.DebugAction;

   // create a session
   ISession session = RuleSessionFactory.createSession();

   // retrieve the id (model path) of the action to be exchanged from the corresponding context
   final String actionID = IContextExecute_Command_Action.VR_ACTION_ID_EXECUTECOMMAND;
   // replace the action implementation with a custom ActionDefinition pointing to the
 DebugAction (IAction)
   session.registerAction(actionID,
 ActionUtil.createActionDefinition(DebugAction.class.getName()));

   // create a request
   IRequest request = session.createRequest();

   // ... execute rules ...
```

There is a another way to exchange an action implementation by registering an `IActionDefinitionFactory` on the `ISession`.

The interface defines a `createActionDefinition()` method which has various parameters including the class name of the implementation, the action id and the path for the action type (see the Javadoc for details). It can return an `IActionDefinition` which will be used to instantiate the `IAction`.

This approach allows a more flexible exchanging, e.g. replacing all implementations of a certain action type. Using `registerAction()` is still possible and an `IActionDefintion` registered by this will be preferred over the `IActionDefinitionFactory`.

The following code illustrates how an action is deactivated. Basically, the "Standard" action type is used as its implementation. "Standard" actions don't do anything.

```
   // an action is "deactivated" by replacing it with the standard action (which does nothing)
   session.registerAction(actionID, ActionUtil.getStandardActionDefinition());
```

3.2.13. Exchanging Service Implementations

Rules can call services that are defined in the rule models. When a custom service type is used, it often is necessary to dynamically exchange the service implementation used at runtime. To achieve this, a client can register a `AbstractServiceFactory` on the rule session (`ISession`) prior to the execution of the rules. A service implementation implements the interface `IService2` or its predecessor `IService`.

Using a `AbstractServiceFactory` allows creating service instances that require an external component or a specific configuration from the environment the rules are executed in.

The following example illustrates how a client can exchange a service implementation using a `AbstractServiceFactory`:

This example here is taken from the "XPath Service" example that comes with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace.

```
session.registerServiceFactory(new AbstractServiceFactory() {

    @Override
    public IService2 createService2(String serviceId, String serviceClassImpl,
     String serviceTypePath, ClassLoader optionalClassLoader) {
     if (XPathService.class.getName().equals(serviceClassImpl))
     {
        // can use external configuration:
        return new DebugXPathService2(System.out);
     }

     // may return null
     return null;
     }
});
```

The `AbstractServiceFactory` is called when a `IService2` implementation needs to be created. For this, a lot of information about the service is passed in as parameters to the `createService2(..)` method. See the method's JavaDoc for more information. Based on this, a client can easily decide for which services the implementation is exchanged. For compatibility reasons, it is also possible to create older `IService` interface implementations with the `AbstractServiceFactory`. A client can choose which implementation to create, by implementing according methods. Note that it is not necessary to return an implementation as the runtime will then create one.

There are other ways to exchange a service implementation, which exist for compatibility reasons. These can only be used for `IService` implementations.

The following example illustrates how a client can register a service implementation on the `ISession` by its id.

This example here is also taken from the "XPath Service" example.

```
   // retrieve the id (model path) of the service whose implementation should be exchanged
   String serviceId = IContextXPath_Service.VR_SERVICE_ID_XPATH;

   // replace the service implementation with another implementation
   session.registerService(serviceId, DelegateXPathService.class.getName());

   // create a request
   IRequest request = session.createRequest();

   // ... execute rules ...
```

It is possible to use `registerService()` and an `IService` registered by this will be preferred over the `IServiceFactory`.

Note that the `IServiceFactory` is deprecated and should no longer be used.

Related Concepts.

- Custom Service Type

3.2.14. Monitoring Rule Execution

The Rule Execution API allows several kinds of listeners to be registered with either the session (`ISession`) or the request (`IRequest`). This mechanism allows an application to be informed about different events happening during rule execution. You can use this to produce custom traces for debugging or profiling purposes or to introduce any kind of custom behaviour into the rule execution.

There are three different kinds of listener interfaces available:

`ISessionListener`
: This listener can be registered with the session and is informed when the session is started, the session is closed and everytime a request is created.

`IRequestListener`
: This listener can be registered with the request and is informed when an individual node (rule element) of a rule is entered and left. Listeners of this type will *not* work if the code was generated with the Enable Rule Statistics and Debugging option of the Java code generator turned off.

`ILifeCycleListener`
: This listener can be registered with the request and is informed when the request is entered and left, and when each individual context (rule model, rule package or rule) is entered and left.

See the following example code that implements and registers each of these listeners. The sample implementations for the listeners simply print out the event on the console.

This example is taken from the "Rule Execution API Example" project that comes with ACTICO Modeler. See `ExampleListener.java`.

```
public static void main(String[] args)
{
   // create a new rule session
   final ISession session = RuleSessionFactory.createSession();
   // add a session listener
   ExampleSessionListener exampleSessionListener = new ExampleSessionListener();
   session.addSessionListener(exampleSessionListener);
   try
   {
      // create a new rule request; the first event the session listener will
      // receive
      IRequest request = session.createRequest();

      // add a lifecycle listener
      request.addLifeCycleListener(new ExampleLifeCycleListener());
      // add a request listener
      request.addRequestListener(new ExampleRequestListener());

      // create a request data object for the rule "Pricing"
      IRuleARequestData requestData = Example_RulesRuleModel
         .createRuleARequestData();

      // access the context of "RuleA" (contains all data elements defined on
      // that rule)
      IContextRuleA contextRuleA = requestData
         .getContextExample_RulesAAARuleA();
      // set the input data element "inA"
      Customer customer = new Customer();
      customer.setId(7);
      customer.setName("Volker");
      customer.setBirthdate(new SimpleDateFormat("dd.MM.yyyy")
         .parse("14.01.1970"));

      contextRuleA.setInA(customer);

      // execute rule "Pricing"
      Example_RulesRuleModel.performRuleA(request, requestData);

   }
   catch (Exception e)
   {
      e.printStackTrace();
   }
   finally
   {
      session.closeSession();
   }
}

/**
 * This listener shows the events of rule elements that are involved in the
 * lifecycle of the request. Additionally the duration of the request can be
 * measured.
 */
private static class ExampleLifeCycleListener implements ILifeCycleListener
{
   private long startTime;

   public void lifeCycleNotification(long timestamp, int discriminator,
      IRequest request, IRequestLifecyleInvolved involved)
   {
      switch (discriminator)
      {
         case ILifeCycleListener.REQUEST_ENTERED :
            startTime = timestamp;
            System.out.println("Entering request at: " + timestamp);
            break;
         case ILifeCycleListener.REQUEST_LEFT :
            System.out.println("Leaving  request at: " + timestamp + " ("
               + (timestamp - startTime) + "ms)");
            break;
         case ILifeCycleListener.REQUEST_SCOPE_ENTERED :
            System.out.println("Entering scope "
               + involved.getClass().getName());
            break;
         case ILifeCycleListener.REQUEST_SCOPE_LEFT :
            System.out.println("Leaving  scope "
               + involved.getClass().getName());
            break;
      }
```

This will produce the following output.

```
Creating request at: 1206537733386
Starting session at: 1206537733417
Entering request at: 1206537733417
Entering scope example_rules.internal.example_rules.ContextExample_Rules
Entering scope example_rules.internal.example_rules.aaa.ContextAAA
Entering node 0 in rule /Example Rules/AAA/RuleA/ClassifyCustomer at: 1206537733433
Entering node 4 in rule /Example Rules/AAA/RuleA/ClassifyCustomer at: 1206537733433
Leaving  node 4 in rule /Example Rules/AAA/RuleA/ClassifyCustomer at: 1206537733433
Leaving  node 0 in rule /Example Rules/AAA/RuleA/ClassifyCustomer at: 1206537733433
Leaving  scope example_rules.internal.example_rules.aaa.ContextAAA
Leaving  scope example_rules.internal.example_rules.ContextExample_Rules
Leaving  request at: 1206537733433 (16ms)
Closing  session at: 1206537733433 (16ms)
```

Related Concepts.

- Rule Session (ISession)
- Rule Request (IRequest)

Related References.

- Java Code Generator Tab

3.2.15. Saving Request Data

Request data can be saved to XML. Thus you can configure your application so that the end user can save his data with a single button press and send it to the rule developers. With another button press in the rule test editor, the rule developer converts the data stored in the file to new test cases. For further information, see the task *Importing Test Cases* in the *Rule Modeling Guide*.

In each of the following examples the execution result of a rule is saved to an XML file and loaded again.

It's possible to store more than one record by calling the append method multiple times.

The following examples are taken from the "Rule Execution API Example" project that comes with ACTICO Modeler. See `ExampleStoreValuesForTestImport` or `ExampleRequestDataPersistence.java`, respectively.

Example 1: Saving Input Data and Output Data in separate Files


```
  private static final int INPUT_DATA = 1;
  private static final int OUTPUT_DATA = 2;

  public static void main(String[] args) throws Exception
  {
     // create a request data object for the rule "Pricing"
     IRuleARequestData requestData = Example_RulesRuleModel.createRuleARequestData();

     // populate input data
     fillInputData(requestData);

     // store input data
     File inputDataFile = storeRequestData(requestData, "Test_Case_Input_Data", INPUT_DATA);
     System.out.println("Input data file location: " + inputDataFile.getPath());

     // execute rule "Pricing"
     Example_RulesRuleModel.performRuleA(requestData);

     // store output data
     File outputDataFile = storeRequestData(requestData, "Test_Case_Output_Data",
OUTPUT_DATA);
     System.out.println("Output data file location: " + outputDataFile.getPath());

     // Now you can import the created files as new test case in test "RuleA Test".
  }
  /**
   * Stores the given request data into a temporary file.
   *
   * @param requestData the {@link IRequestData} to store
   * @param fileName name of the file to create
   * @param type INPUT_DATA to store rule input data, OUTPUT_DATA to store rule output data
   * @return the file where the request data is stored
   * @throws IOException
   * @throws VRException
   * @throws XMLStreamException
   */
  private static File storeRequestData(IRequestData requestData, String fileName, int type)
throws IOException,
     XMLStreamException, VRException
  {
     // Preparing the output stream writer
     File tempFile = File.createTempFile(fileName + ".", ".vrdata");

     FileOutputStream fos = new FileOutputStream(tempFile);
     OutputStreamWriter osw = new OutputStreamWriter(fos);

     // Create the saver and store the request data
     RequestDataXmlSaver xmlSaver;

     if( type == INPUT_DATA )
     {
        xmlSaver = RequestDataXmlSaver.createInParameterSaver(osw);
     }
     else
     {
        xmlSaver = RequestDataXmlSaver.createOutParameterSaver(osw);
     }

     xmlSaver.append(requestData);

     // Tell the saver that no more request data will be appended
     // (there can be multiple request data instances appended)
     xmlSaver.finish();

     // cleanup of the output writer
     osw.flush();
     osw.close();

     return tempFile;
  }
  private static void fillInputData(final IRuleARequestData requestData) throws
ParseException, Exception
  {
     // access the context of "RuleA" (contains all data elements defined
     // on that rule)
     IContextRuleA contextRuleA = requestData.getContextExample_RulesAAARuleA();

     // set the input data element "inA"
     Customer customer = new Customer();
     customer.setId(7);
```

Example 2: Saving Input Data and Output Data in one File


```
  public static void main(String[] args) throws Exception
  {
     // create a request data object for the rule "Pricing"
     IRuleARequestData requestData = Example_RulesRuleModel.createRuleARequestData();

     // execute the rule to store some data in the requestData instance
     executeRule(requestData);

     // store the request data
     File requestFile = storeRequestData(requestData);

     // load the request from the file
     loadAndPrintRequestData(requestFile, requestData.getClass());
  }
  /**
   * Load the request data(s) of the given type.
   *
   * @param requestFile file containing the request data
   * @param clazz the type of the expected request data
   * @throws IOException
   * @throws FactoryConfigurationError
   * @throws XMLStreamException
   */
  private static void loadAndPrintRequestData(final File requestFile, final Class clazz)
throws IOException,
     XMLStreamException, FactoryConfigurationError
  {
     // setup of the xml stream reader
     FileReader fr = new FileReader(requestFile);
     XMLStreamReader xmlReader = XMLInputFactory.newInstance().createXMLStreamReader(fr);

     // create a xml loader instance.
     RequestDataXmlLoader xmlLoader = RequestDataXmlLoader.createRequestDataLoader(clazz,
xmlReader);

     int requestCt = 0;
     System.out.println("Reading requests from file: '" + requestFile.getPath() + "'.");

     // load all request data
     while (xmlLoader.hasNext())
     {
        requestCt++;
        IRequestData next = (IRequestData) xmlLoader.next();
        System.out.print("Request no. " + requestCt + ": " + next.getParameters());
     }

     xmlReader.close();
  }

  /**
   * Stores the given request data into a temporary file.
   *
   * @param requestData the {@link IRequestData} to store
   * @return the temporary file where the request data is stored
   * @throws IOException
   * @throws VRException
   * @throws XMLStreamException
   */
  private static File storeRequestData(IRequestData requestData) throws IOException,
XMLStreamException, VRException
  {
     // Preparing the output stream writer
     File tempFile = File.createTempFile("VRRequestExample-", ".vrdata");
     tempFile.deleteOnExit();

     FileOutputStream fos = new FileOutputStream(tempFile);
     OutputStreamWriter osw = new OutputStreamWriter(fos);

     // Create the saver and store the request data
     RequestDataXmlSaver xmlSaver = RequestDataXmlSaver.createInOutParameterSaver(osw);
     xmlSaver.append(requestData);

     // Tell the saver that no more request data will be appended
     // (there can be multiple request data instances appended)
     xmlSaver.finish();

     // cleanup of the output writer
     osw.flush();
     osw.close();
```

3.2.16. Convert Request Data into XML fragments

Request data can be converted into XML fragments or populated from such a XML fragment. These XML fragments correlates to the input/outut sections of VRRequests and VRRespones as they are handled by the ACTICO Execution Server (see Execution Server Guide, chapter *Rule Service Request Format*).

The following example code illustrates how a request data object can be converted into a XML fragment. In a second step that XML fragment is used to populate a newly created request data object.

This example is taken from the "Rule Execution API Example" project which is shipped with ACTICO Modeler (see `ExampleVRRequest.java`).

```
  public static void main(String[] args) throws Exception
  {
     // Create and populate a new request data instance
     IRequestData requestData = createPopulatedRequestData();

     // Create a XML snippet from the request data
     String xml = writeToXMLSnippet(requestData);
     System.out.println("Request data (input values) transformed to a XML snippet: " + xml);

     // Create a new request data instance based on the XML snippet
     IRequestData recreated = readFromXMLSnipped(xml);
     Customer customer = (Customer) recreated.getInParameter("inA");
     System.out.println("Customer recreated from the XML snippet: " + customer.getName());
  }

  /**
   * Creates a XML snippet for the given request data instance.
   *
   * @param requestData the request data
   * @return a XML snippet representing the request data values
   */
  private static String writeToXMLSnippet(IRequestData requestData) throws Exception
  {
     ByteArrayOutputStream bos = new ByteArrayOutputStream();
     OutputStreamWriter osw = new OutputStreamWriter(bos);

     VRRequestSaver requestSaver = VRRequestSaver.createRequestSaver(osw);
     requestSaver.appendAsInput(requestData); // It's also possible to write the output data
by using the
                                              // appendAsOutput method

     osw.flush();
     osw.close();

     return bos.toString();
  }

  /**
   * Read the XML snippet and create a populated request data for rule 'RuleA'.
   *
   * @param xml the XML snippet
   * @return the created and populated request data instance
   */
  private static IRequestData readFromXMLSnipped(final String xml) throws Exception
  {
     // Prepare the XML reader
     IRuleARequestData requestData = Example_RulesRuleModel.createRuleARequestData();
     ByteArrayInputStream bis = new ByteArrayInputStream(xml.getBytes());
     XMLStreamReader xmlReader = XMLInputFactory.newInstance().createXMLStreamReader(bis);

     // Create the request loader
     VRRequestLoader requestLoader = VRRequestLoader.createRequestLoader(xmlReader);

     // Populate the request data
     requestLoader.populate(requestData);

     return requestData;
  }

  /**
   * Creates a request data instance for rule 'RuleA' and stores some values in it.
   *
   * @return the request data instance
   */
  private static IRequestData createPopulatedRequestData()
  {
     // Create request data for rule 'RuleA'
     IRuleARequestData requestData = Example_RulesRuleModel.createRuleARequestData();

     // Store some dummy values into the request data
     IContextExample_Rules exampleRuleCtx = requestData.getContextExample_Rules();
     IContextRuleA rulesAAACtx = requestData.getContextExample_RulesAAARuleA();
     Customer customer = new Customer();
     customer.setBirthdate(new Date());
     customer.setId(1744);
     customer.setName("Foo Bar");
     rulesAAACtx.setInA(customer);

     return requestData;
  }
```

Chapter 4. Executing State Flows

State Flows are integrated and called by applications using the State Flow Execution API. The State Flow Execution API is a programming interface provided by the Rules Runtime Library and the generated source code. This API enables you to:

- create session objects that are necessary for executing State Flows
- initialize State Flows
- initialize State Flows in a desired state
- set input data
- create events necessary during the execution
- execute State Flows
- register and deregister listeners for State Flow events
- query information about State Flows

The following sections explain the use of this API to call State Flows including the entire necessary data exchange between application and State Flow.

4.1. Features

A State Flow provides an overview of different states which a system may adopt during runtime and it specifies the events which may cause transitions between individual states. These state transitions can be rule-based. In the following, important details of State Flows are summarized that should be considered using the State Flow Execution API.

4.1.1. State Flows (IStateflow)

The central Java class of State Flows is `IStateflow`. A State Flow doesn't run independently but is integrated into an application. The application uses it, passes events and responds to changes in state. This state transition triggered by an event is reached by calling the method `transition` of the State Flow object `IStateflow`. The continuous integration into the application is a significant property and distinguishes it from a rule call. Therefore, a State Flow doesn't represent a rule but it can use so-called Transition Rules to control state transitions.

Related Concepts.

- Executing State Flows

4.1.2. Events (IEvent)

State transitions are triggered by fired events referring to a certain state. The decision criterion is the name of the event. For this purpose, the name of the events defined for a state has to be unique. Events can also provide input data for the transition rules that are used to determine the target state. In State Flows you have the possibility to specify „Else“ events for states. This concept makes it possible to trigger state transitions if the name of the events doesn't match.

Related Concepts.

- Creating Events

4.1.3. States (IState)

During runtime a State Flow can only be in one active state. When a State Flow is instantiated the start state automatically represents the active state. Hence the execution of state transitions can also be regarded as a change of the active state. Besides normal states a State Flow may also possess one or multiple end states. If the active state is an end state, this is equivalent to the end of a State Flow during runtime. The active state and further information can be asked during runtime.

4.1.4. Transitions and Transition Rules (ITransition)

Transitions between states are represented by transition objects (`ITransition`). They can be divided into two types. On the one hand there are rule-based transitions including the execution of transition rules and having the task to set the target state. On the other hand there are ordinary, non-rule-based transitions. If a transition is rule-based the last entered Transition Target Element of the rule decides which target state will be taken. If no Transition Target Element was entered during runtime the State Flow remains in the current active state. The decision which target state will be reached can be made on the base of input data. They can be defined globally in the State Flow or privately in the transition rules. An input data element defined in the State Flow is visible and usable in all transition rules. It is important to note that an input data element of a transitional rule may only be of type "input".

4.1.5. On-Entry and On-Exit Rules

It is possible to define rules which are executed on entry or exit of a state. These rules are executed with a transition, where the on-exit rule of the originated state is executed first, then the transition is triggered and finally the on-entry rule of the target state is executed. In case an on-entry rule throws an exception, the State Flow will not be in the target state but the transition was triggered. In case an on-exit rule throws an exception, the State Flow will stay in the originating state and no transition is triggered. There are events for the on-entry and on-exit rules in order to observe this behavior at runtime. See Observing the Execution of State Flows and Table 4.1, “Event Overview” for more information.

4.2. Concepts

4.2.1. Rule Session (ISession)

Before you can call a State Flow it is necessary to establish a rule session. The reason is that the containing rule based transitions of a State Flow require the rule session to perform their transition rules. The State Flow is responsible for providing the rule session across the transition rules. If the Stateflow Execution API is used the lifecycle of a rule session is as follows:

1. At first the application creates a rule session (`ISession`) by the `RuleSessionFactory`
2. The application creates an instance of the State Flow (`IStateflow`) by the `IStateflowFactory` and the rule session
3. The application triggers any number of state transitions. This might result in the execution of transition rules with the created rule session
4. Finally the rule session is closed by the application

The main difference to the call of rules is that calling State Flows by the State Flow Execution API encapsulates the creation of the rule requests. The following code excerpt exemplifies the creation of a session object and the initialization of a State flow.

```
 1 // (1) create a rule session
 2 ISession session = RuleSessionFactory.createSession();
 3 try
 4 {
 5     // (2) create a State Flow instance
 6     IStateflow stateflow = Example_StateflowsStateflowFactory.
 7         createExample_StateflowsAStateflowA(session);
 8
 9     // (3) trigger a transition
10     stateflow.transition(stateflow.createEvent("process"));
11 }
12 finally
13 {
14     // (4) close the rule session
15     session.closeSession();
16 }
```

4.2.2. Initializing State Flows

Initializing a State Flow is done by instantiating a State Flow object (`IStateflow`). This is exemplified in step two of the previous source code snippet. During that the start state is automatically set as the current state in the State Flow. The State Flow Execution API also provides the possibility to initialize a State Flow in a specific state. Totally the simple initialization of a State Flow consists of the following two steps:

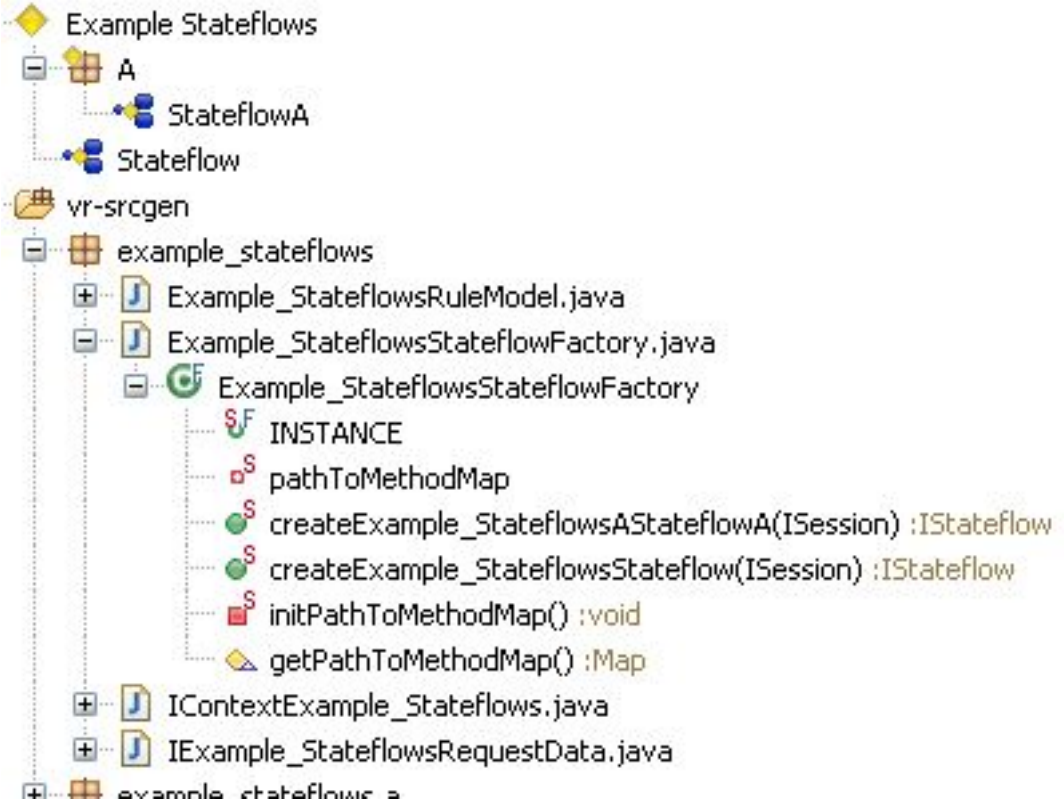
a. Setting the start state as the active state

b. Creating a State Flow Data object (`IStateflowData`) which is required to set global input data `IRequestData`. The request data object can also be used to query the active state in a State Flow.

Similar to creating a rule session the State Flow Execution API also offers the ability to create a State Flow (`IStateflow`) by a factory pattern (`IStateflowFactory`). This factory is part of the generated source code and provides a static method (`createXXX(ISession session)`). The method name is a composition of the prefix "create", the name of the rule model and the name of the State Flow. Currently there are two ways to use such a method. For one thing it can be used as follows:

```
IStateflow stateflow = Example_StateflowsStateflowFactory.
    createExample_StateflowsAStateflowA(session);
```

For another thing it can be called via the rule model. To call the State Flow factory via that way is very useful if the rule model contains more than one State Flows. Such a scenario is illustrated by the following diagram:



- The rule model "Example Stateflows" contains a rule package named "A"
- The rule package "A" contains a State Flow named "StateflowA"
- The rule model "Example Stateflows" contains a State Flow "Stateflow"

Relating to the diagram it is possible to identify all State Flows in the rule model "Example Stateflows" by calling:

```
String[] identifiers = Example_StateflowsRuleModel.INSTANCE.
    getStateflowFactory().getStateflowIdentifiers();
```

The result of that call would be an Array containing the State Flow identifiers respectively the rule paths of the State Flows (`[/Example Stateflows/Stateflow, /ExampleStateflows/A/StateflowA]`). These rule paths then can be used to instantiate a specific State Flow via the rule model and the State Flow Factory.


```
IStateflow stateflow = Example_StateflowsRuleModel.INSTANCE.getStateflowFactory().
    createStateflow(session, "/Example Stateflow/Stateflow");
```

As previously mentioned the rule model `Example_StateflowRuleModel` as well as the State Flow factory are part of the generated source code of a rule project. So there are certain rules in the naming of these classes which should be considered:

a. The class-name of the rule model is composed of the rule model name and the suffix "`RuleModel`"

b. The class-name of the State Flow factory is composed of the rule model name and the suffix "`StateflowFactory`"

c. The name of the `create`-method is composed of the prefix "`create`", the rule model name, may be the package names and finally the name of the State Flow e.g. (`createExample_StateflowsAStateflowA`)

If a State Flow has been initialized, the `IStateflow`-instance is the central entry point to perform state transitions and other operations.

Related Concepts.

- Initializing State Flows in a specific State

4.2.3. Initializing State Flows in a specific State

The initialization process of a State Flow is described in detail in the previous section. In that section the focus is on starting a State Flow and setting the active state to the start state. Apart from that the State Flow Execution API additionally has the ability to initialize a State Flow in s specific state. This is especially useful if a State Flow should be restored after it has been persisted. The following code snippet shows how this re-initialization works:

```
// (1) create a rule session
ISession session = RuleSessionFactory.createSession();
try
{
    // (2) create a State Flow instanz
    IStateflow stateflow = Example_StateflowsStateflowFactory.
        createExample_StateflowsAStateflowA(session);

    // (3) initialise the State Flow in the state "Processed"
    IStateflowData data = stateflow.createStateflowData("Processed");
    stateflow.setCurrentStateflowData(data);

    // (4) trigger a transition
    stateflow.transition(stateflow.createEvent("finish"));
}
finally
{
    // (5) close the rule session
    session.closeSession();
}
```

In this example the State Flow "StateflowA" is instantiated. After that the state "Processed" is set as active state. This concept is based on the class `StateflowData` which maintains the active state in the State Flow. Compared to initialising a State Flow in the start state the initialisation in a specific state only differs in the additional necessary call written in step three.

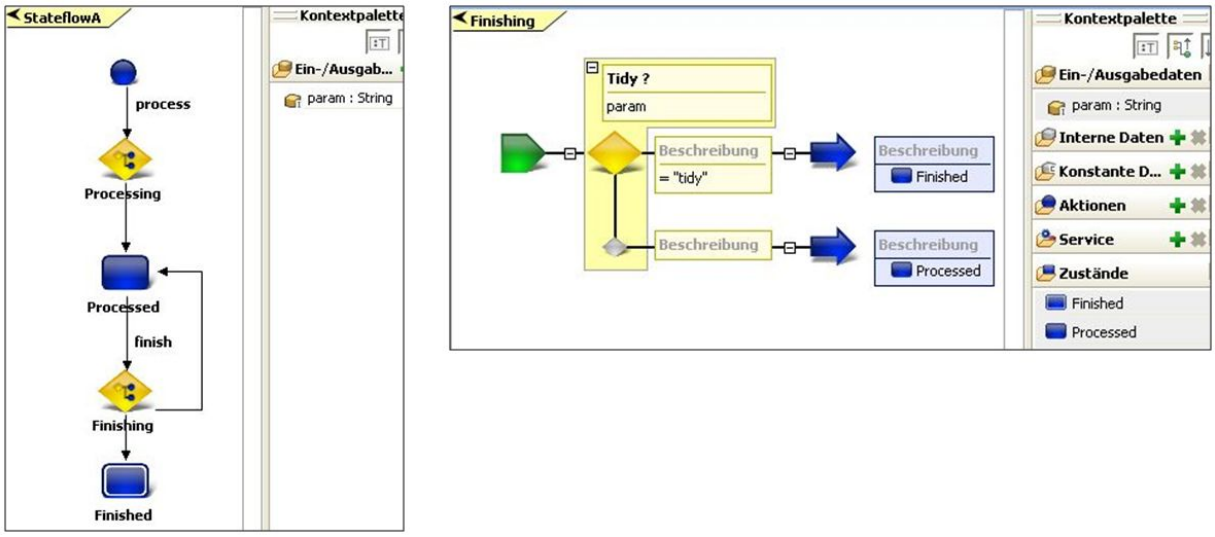
a. Creating a `StateflowData` object by the method `createStateflowData()` via the `IStateflow` instance. This object contains the name of the active state

b. Setting the active state (`StateflowData`) by the method `setCurrentStateflowData` via the `IStateflow` instance

4.2.4. Setting global Request Data

Input data (`IRequestData`) can be defined globally in a State Flow. Global input data are visible and usable for each transition rule. At this point, they are mainly used to determine the target state of a state transition. A

State Flow itself does not use input data. But defining input data globally in the State Flow can be very useful to enable reusability. As the following figure shows, the State Flow "StateflowA" defines an input date element named "param". This is available in the transition rule "Finishing".



At runtime the application has to populate the input data element "param" with specific data. This shows the following source code excerpt:

```
// set the input data element "param"
stateflow.getCurrentStateflowData().getRequestData().setInParameter("param", "tidy");
```

By this statement the value "tidy" is assigned to the input data element "param" in the State Flow. In detail this statement does the following:

1. Queries the State Flow Data object (`StateflowData`) by the `getCurrentStateflowData()` method on the `IStateflow` instance.

2. Queries the request data object (`RequestData`) by the `getRequestData()` method.

3. Populates the request data object (`RequestData`) by the generic `setInParameter()` method.

A State Flow has to be initialized before it can be populated with input data. How this can be achieved is described in the previous sections (Initializing State Flows) and (Initializing State Flows in a specific State). If you would like to initialize the State Flow "StateflowA" into the State "Processed" and then set the input data element "param", the following code has to be implemented:


```
// (1) create a rule session
ISession session = RuleSessionFactory.createSession();
try
{
    // (2) create a State Flow instanz
    IStateflow stateflow = Example_StateflowsStateflowFactory.
        createExample_StateflowsAStateflowA(session);

    // (3) initialise the State Flow in the state "Processed"
    IStateflowData data = stateflow.createStateflowData("Processed");
    stateflow.setCurrentStateflowData(data);

    // (4) set the input data element "param"
    stateflow.getCurrentStateflowData().getRequestData().
        setInParameter("param", "tidy");

    // (5) trigger a transition
    stateflow.transition(stateflow.createEvent("finish"));
}
catch (VRException e)
{
    e.printStackTrace();
}
finally
{
    // (6) close the rule session
    session.closeSession();
}
```

The concept of setting request data is already described in Request Data (IRequestData) and is not discussed in more detail in this section. In the context of State Flows, however, some special rules have to be considered when an application triggers events and if input data are required:

- So it is important to set input data before calling the corresponding `transition`-method
- It is also important to note that when initializing the State Flow in a state (step 3) no input data may be set
- It is necessary to call `getCurrentStateflowData()` to set input data properly

Global input data can be declared as required or as optional input data. A rule-based state transition can only succeed if all required input data have been set. To check this, the method `getNotSetRequiredInParameters()` can be used. It retrieves all required and not set input data as `IParameterInfo` objects.

```
// Query all required and not set input data
IParameterInfo[] requiredInfo =
 stateflow.getCurrentStateflowData().getRequestData().getNotSetRequiredInParameters();
```

All optional and not set input data can be retrieved by the method `getNotSetOptionalInParameters()`.

```
// Query all optional and not set input data
IParameterInfo[] optionalInfo =
 stateflow.getCurrentStateflowData().getRequestData().getNotSetOptionalInParameters();
```

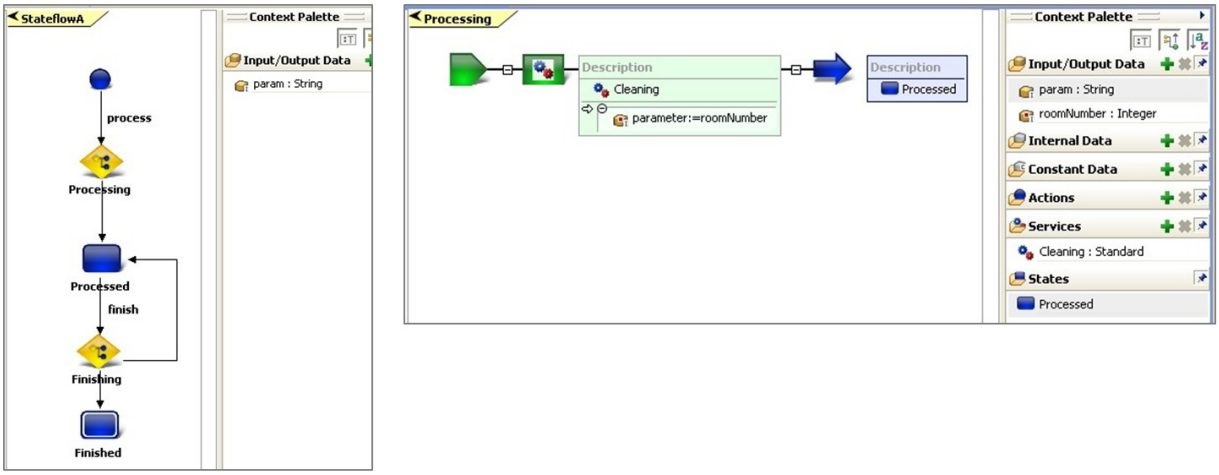
4.2.5. Creating Events

Events are the triggers causing state transitions. Normally a fired event causes the change of the active state. The occurrence of a certain event in the State Flow is detected by matching the name of the event. As described in previous source code examples you can create an event in the following way:

```
// Creation of the event with the name "process"
IEvent event = stateflow.createEvent("process");
```

In this example the event `event`, which is an instance of class `BasicEvent` and implements the interface `IEvent`, is created.

In State Flows and in transition rules as well, input data can be defined. Such a scenario is described by the following figure:



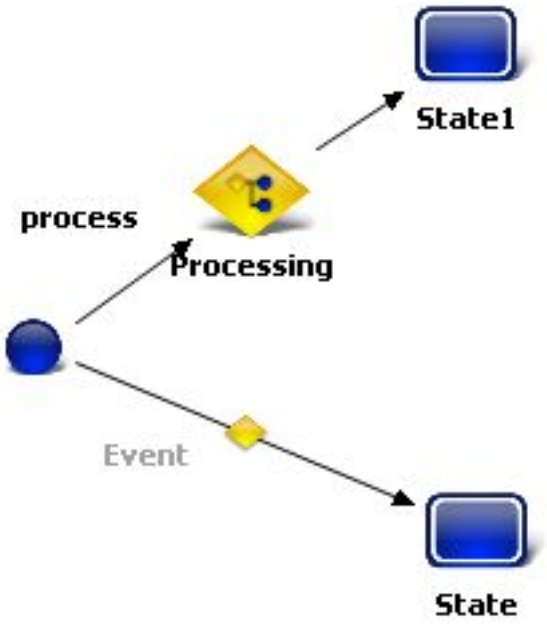
In the left screenshot the State Flow "A" is shown, which contains an input data element named "param". In the right screenshot the transition rule named "Processing" is shown. The rule has access to the global input data element "param" and defines a private input data element "roomNumber". Both input data elements are visible and usable within the rule. Due to the fact that the triggering event leads to a transition rule call, the event is responsible for providing the input data value at runtime. Referring to the example, this means that the event "process" must deliver the value of the input date "roomNumber". With the State Flow Execution API this can be done in the following way:

```
// Create the event processEvent with the name "process" and set the input data element
 "roomNumber"
IEvent processEvent = stateflow.createEvent("process");
processEvent.getRequestData().setInParameter("roomNumber", 11);
```

Optional and required Input data of transition rules can be checked by the methods `getNotSetRequiredInParameters()` and `getNotSetOptionalInParameters()` whether they already have been set.

```
// Query all required and not set input data
IParameterInfo[] requiredInfo = processEvent.getRequestData().getNotSetRequiredInParameters();
// Query all optional and not set input data
IParameterInfo[] optionalInfo = processEvent.getRequestData().getNotSetOptionalInParameters();
```

A special feature in State Flows is the default event that is explained in the following.



Here a State Flow with the following properties is displayed:

- The State Flow contains the states: "Start", "State1" and "State".
- The transition from state "Start" to state "State1" is triggered by the event "process", is rule-based and is controlled by the Transition Rule "Processing".
- The State Flow contains a non-rule-based transition from state "Start" to state "State" which is caused by a default event.

A default event (also referred to as else-event or then-event) is indicated within the editor by the missing name. In this case the name of the event has the value `null`. For example, if the start state is the active state then this has the following meaning:

- If the event with the name "process" is fired then a transition from state "Start" to state "State1" takes place.
- If an event occurs whose name doesn't equal the name "process" (e.g. "start", "test" or "proces") a default transition from state "Start" to state "State" takes place.

The default event offers the possibility to trigger a transition by an event with an arbitrary name, starting from a certain state. If a state doesn't have a default event the firing of an unknown event would cause an exception and the state transition would fail. Using the State Flow Execution API a default event can be created in the following way:

```
// Creation of the default event defaultEvent
IEvent defaultEvent = stateflow.createEvent(IEvent.DEFAULT);
```

4.2.6. Executing State Flows

The execution of State Flows corresponds to the triggering of one or multiple state transitions depending on events occurring. A State Flow can be considered to be finished if the active state is an end state. A state transition can be executed via the State Flow object `IStateflow` with the method `transition(IEvent event)`. The use of this method is described within source code segments of previous sections and it is the most important function of the State Flow Execution API. In a State Flow it should be used in the following way:

```
// Creation of the event with the name "process"
IEvent event = stateflow.createEvent("process");

// Execution of a state transition
IStateflowData data = stateflow.transition(event);
```

The method `transition` requires the triggering event - an instance of `IEvent` - as parameter and returns a State Flow data object `IStateflowData`. The State Flow data object is the central object used to initialize State Flows with a certain state (Initializing State Flows in a specific State) or to set input data (Setting global Request Data). After the execution of the method `transition` it's possible to access the data object to find out the active state.

4.2.7. Observing the Execution of State Flows

The execution of a State Flow consists of a sequence of state transitions which depend on events occurring. The state transitions may be rule-based and contain the execution of Transition Rules. The State Flow Execution API provides an event-listener mechanism allowing the observation of the execution of State Flows. This informs the application about different events occurring during the execution of a State Flow. For example, this concept may be used for a user-specific observation of the State Flow or for any other user-specific operations.

The event-listener concept of the State Flow Execution API provides for two roles. On the one hand there are event producers sending special event objects. On the other hand there are observers that are registered/deregistered at the producers - they are called listeners in this context.

`Event Producers:`
: The event producer is the State Flow itself. It sends event objects occurring during the execution of state transitions. Events extend the class `AbstractStateflowEvent` and `IStateflowEvent` respectively.

`Observers:`

The observers or State Flow listeners implement the Java interface `IStateflowEventListener`. If a State Flow event occurs they are informed about it via the method `handleStateflowEvent(IStateflowEvent stateflowEvent)`. For this purpose the State Flow class offers the method `addStateflowEventListener(IStateflowEventListener stateflowEventListener)` to register observers. Whenever an event takes place the State Flow creates an event object and informs each registered listener by calling the method `handleStateflowEvent(IStateflowEvent stateflowEvent)`.

The `AbstractStateflowEventListener` class makes the implementation of a `IStateflowEventListener` easier.

Table 4.1. Event Overview

Event	Description
`TransitionFiredEvent`	Fired during the execution of a State Flow if a state transition has happened. It provides the source state (`sourceState`), the target state (`targetState`) and the triggering event (`triggeringEvent`) as attributes
`TransitionTargetTriggeredEvent`	Fired if a state transition is rule-based and, during the execution of the Transition Rule, a Transition Target Element is reached. This event provides the target state (`targetState`), the rule path of the Transition Rule (`rulePath`) and the ID of the Transition Target Element (`id`) as attributes.
`RuleExecutedOnEnterEvent`	Fired after an on-entry rule has been executed. The event contains information about the state that is going to be entered and the rule path of the executed rule. This event is fired even in case the rule execution throws an exception.
`RuleExecutedOnExitEvent`	Fired after an on-exit rule has been executed. The event contains information about the state that has been exited and the rule path of the executed rule. This event is fired even in case the rule execution throws an exception.

Chapter 5. Debugging Rules

ACTICO Modeler provides rule debugging capabilities based on the Eclipse debugging framework. This can be used to define breakpoints within rules, single-step through rule execution and inspect the data currently being processed by the rules.

There are two different kinds of debug scenarios. The first scenario is debugging rule tests. This can simply be done with the rule test editor and is explained in detail in the "Rule Modeling Guide".

The second scenario is debugging rules integrated into any application, running in some JVM. This is achieved by using the remote debugging capabilities of ACTICO Modeler and is explained in this chapter.

5.1. Concepts

5.1.1. Rule Test Debugging

ACTICO Modeler has built-in capabilities for rule test debugging. This lets a user step through a rule test and see what is happening in the rules being executed. Rule test debugging can simply be initiated by clicking the Debug button in the rule test editor or by launching an ACTICO Rules Test launch configuration in Debug mode.

Rule test debugging is explained in full detail in the "Rule Modeling Guide".

5.1.2. Remote Debugging

Remote debugging allows the rule debugger to debug rules being executed in a separate JVM. This JVM can run on the same machine as the debugger or on any other machine connected to the network. This is useful when rules have been integrated into Java applications, e.g. web applications running on an application server or else.

In order for remote debugging to work, the JVM running the application containing the rules needs to be configured for remote debugging. ACTICO Modeler uses the standard debugging interface of a Java VM (JPDA - Java Platform Debugger Architecture) for this purpose.

Once the remote JVM is configured correctly and running, the rule debugger can connect to this JVM and do rule debugging.

See the related sections containing instructions for both these steps.

Related Tasks.

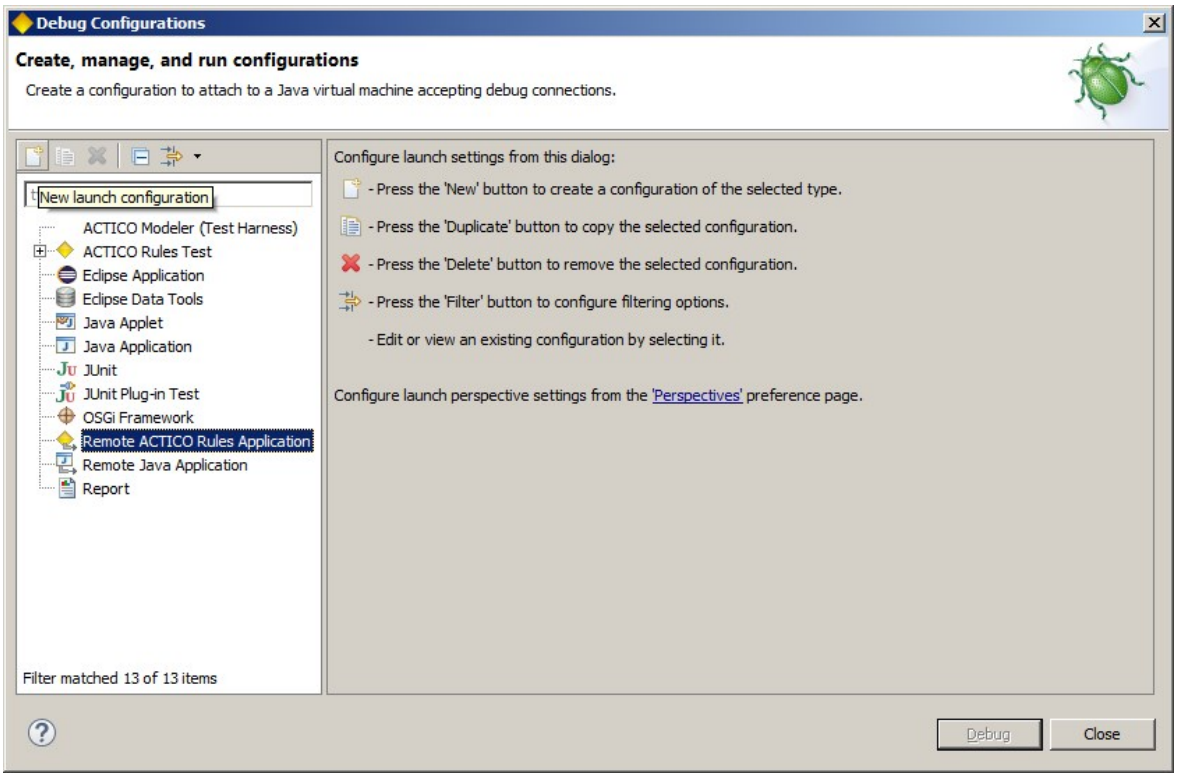
- Debugging Rules Remotely
- Preparing a Java VM for remote Debugging

5.2. Tasks

5.2.1. Debugging Rules Remotely

In order to debug a rule execution in another JVM, do the following:

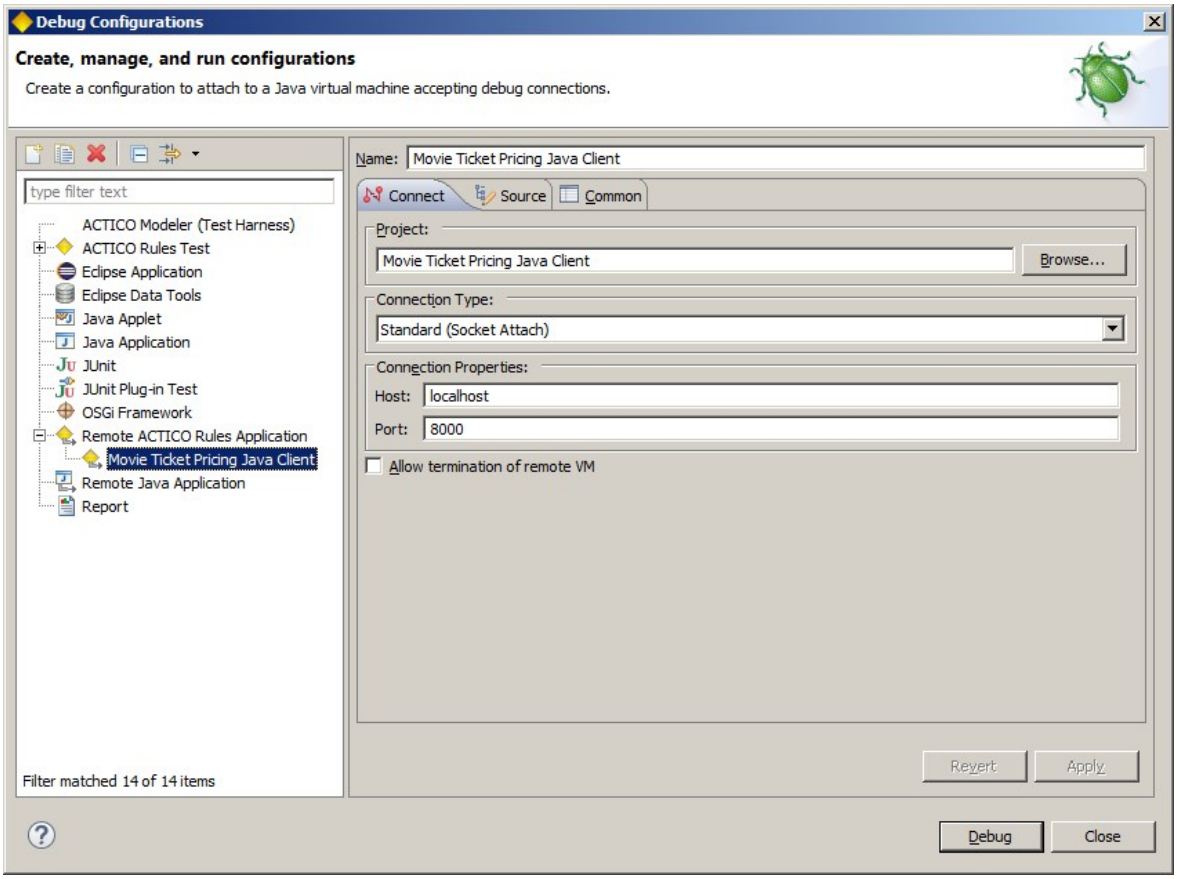
1. Setup the remote JVM executing the rules to listen for incoming debug connections. See Preparing a Java VM for remote Debugging on how to do that.
2. Start the remote JVM containing the application and rules to be debugged.
3. Select Run > Open Debug Dialog... to open the Debug dialog.
4. Select Remote ACTICO Rules Application from the list of launch types. Click the New launch configuration button above the list.



5. Enter a name for the new configuration.

6. Specify the Project that corresponds to the remote application/rules you want to debug. This can be either a rule project or any other project in the workspace using the rules to be debugged.

7. Enter the host and port number where the remote JVM is listening for incoming debug connections.



8. Click on Debug to connect to the remote JVM.

9. Once the remotely executed rules hit a breakpoint, the Debug perspective will be activated and you can start debugging. See the "Rule Modeling Guide" for more information about how to use the debugger.

If the remote JVM was configured to wait for the debugger (`suspend=y`), it will start execution as soon as you click on the Debug button and the debugger connects. So you may want to define breakpoints *before* you start the debug session.

Related Tasks.

- Preparing a Java VM for remote Debugging

5.2.2. Preparing a Java VM for remote Debugging

ACTICO Modeler uses the standard debugging interface of the Java VM (JPDA - Java Platform Debugger Architecture) to debug rule execution remotely.

You have to setup remote debugging only if you want to debug rules integrated into a Java application. If you want to debug a rule test, you can simply use the debug button within the rule test editor. See the "Rule Modeling Guide" for details.

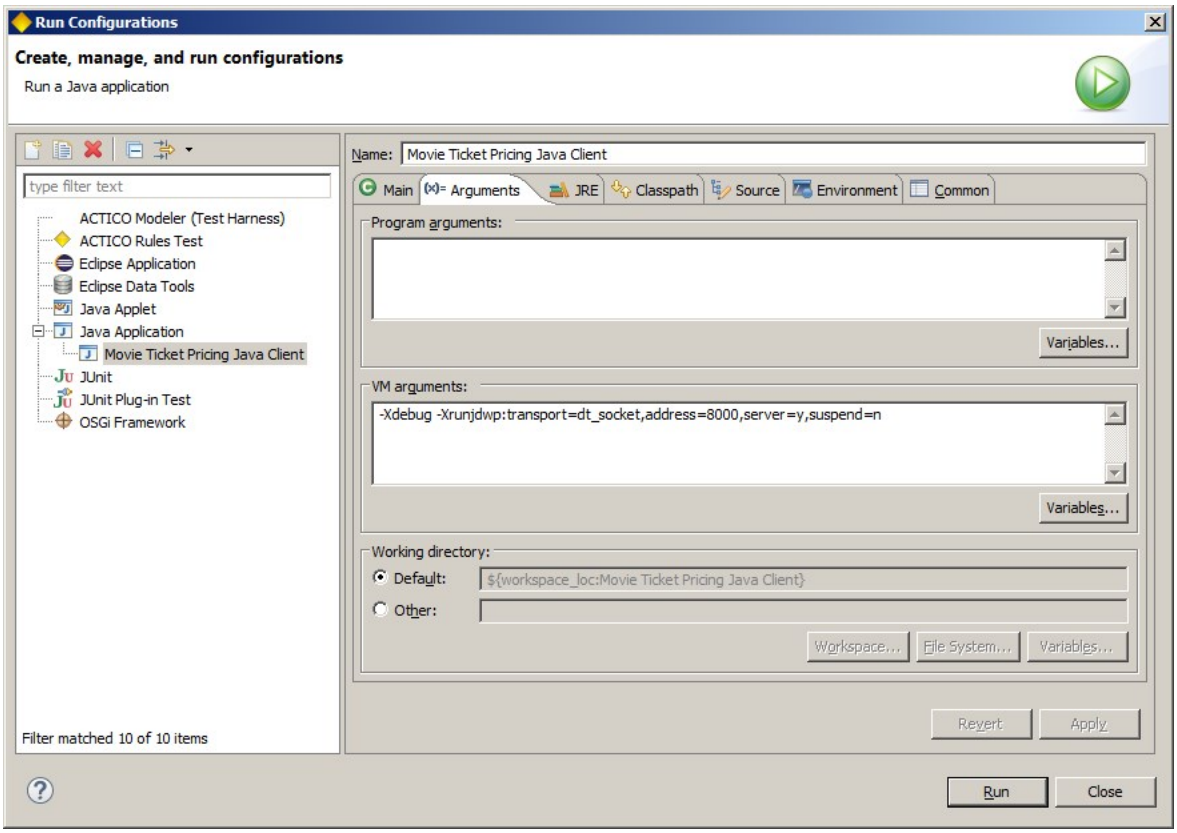
1. Add the VM arguments necessary to enable remote debugging to the command line starting the JVM. For a Sun JVM these options look like this:

```
-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```

   The number 8000 specifies the port number that the remote debugger later can connect to. You can choose any other port if you like.

   If you want the JVM to suspend execution until the remote debugger connects, you should say `suspend=y` at the end of the example line.

2. If you are starting your application from within Eclipse, you can enter the above VM arguments on the Arguments tab of the launch configuration. For example, this is how you can start the "Movie Ticket Pricing Java Client" example with debugging enabled.



3. Once the remote JVM is up and running, you can connect the rule debugger to do rule debugging. See Debugging Rules Remotely for instructions on how to do that.

Related Tasks.

- Debugging Rules Remotely

Chapter 6. Importing Java Object Models

ACTICO Modeler can import existing Java object models so they can be used as data types for rules. These Java object models can consist of any number of Java classes with getter and setter methods for their attributes. This chapter explains the concepts and tasks involved for importing Java object models.

6.1. Concepts

6.1.1. Java Type Library

ACTICO Modeler can import existing Java object models. In ACTICO Modeler these object models are called Java type libraries. Rule models (or rule packages) can define a Java type library and then import the Java classes into that library.

Java type libraries are configured on the Java Type Library tab, available when a rule model or a rule package is selected. A Java project needs to be specified where the java classes can be found. Additionally include and exclude patterns that specify exactly which classes to import have to be specified.

During the import into the Modeler every Java class imported is turned into a structure, enumeration or exception. Normal classes become structures, Java 5 enums become enumerations and classes that extend `java.lang.Exception` become exceptions.

The Modeler also analyzes the getters and setters and creates corresponding attributes in the data types. These attributes may be read-only or write-only depending on whether only a getter, only a setter or both are found.

Constants defined within the classes (`public static final`) are imported and corresponding constant elements are created within the data types or in the package (as selected by the user).

ACTICO Modeler imports all the built-in Java primitive types (`int`, `long`, `double`, `char` etc.) and wrapper types (`java.lang.Integer`, `java.lang.Long`, `java.lang.Double` etc.), `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.Date`, `java.util.Calendar`, `java.sql.Time`, `java.sql.Date`, `java.sql.Timestamp`. Collections (`java.util.Collection`, `java.util.Set`, `java.util.List`) and maps (`java.util.Map`) are also imported.

The classes imported have to implement `java.io.Serializable` or implement a `clone()` method, if and only if the Java code generator is configured to use Copy by value for assignments (see Java Code Generator Tab) or when the rules use the `copy()` function to create a copy of a value of that type. In both cases, it is necessary for the rules to be able to create copies of an object.

It either does that by calling the `clone()` method or, if that method is not available, through serialization. You will see a `NotSerializableException` when neither the `clone()` method is available, nor does the class implement `java.io.Serializable`.

If your Java code generator is set to Copy by reference and your rules don't use the `copy()` function, the imported classes neither have to implement `java.io.Serializable`, nor do they have to implement the `clone()` method.

It is essential to ensure that imported classes implement `equals()` and `hashCode()` methods correctly, i.e. considering technical and business aspects, because those methods are used for comparison. Details on how the comparison operators work in rule models can be found in the **Modeling Guide**.

If imported classes come with a parameterless default constructor then they can be constructed by the rule runtime. If such a class is used as the data type for a data element then an instance is automatically created when the data element is accessed for the first time.

If a default value is defined for the data element, this value will be used. That way it is possible to suppress the automatic creation of a structured data type by setting `null` as default value. If a data type cannot be constructed, an instance will not be automatically created. In this case the default values is `null`.

The Java Type Library tab is not available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective first.

Related Tasks.

- Importing Java Object Models

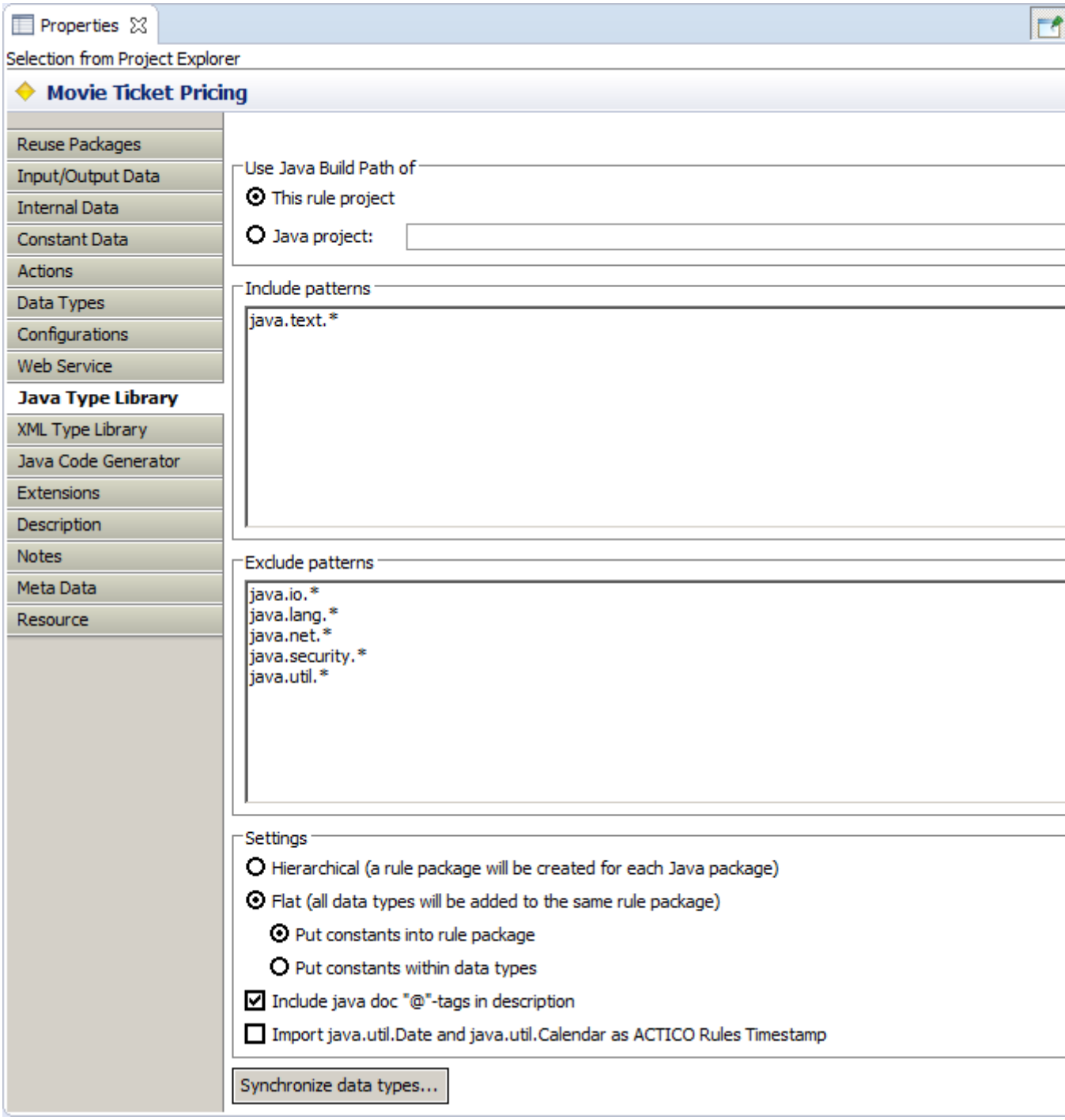
Related References.

- Java Type Library Tab

6.2. Tasks

6.2.1. Importing Java Object Models

1. Select a rule package where you want the imported data types to be located.
2. Use the Java Type Library properties tab to refer to a Java project in your workspace. You can import any Java classes that are visible on the build path of that project. You can either select the This Rule Project option or refer to any other Java project in the workspace.



The Java Type Library tab is not available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective first.

3. Click on  or  to define which packages or classes you want to import. All the classes and interfaces that match one of the include patterns will be imported, along with all related classes and interfaces.

You can sort elements alphabetically by clicking on  .

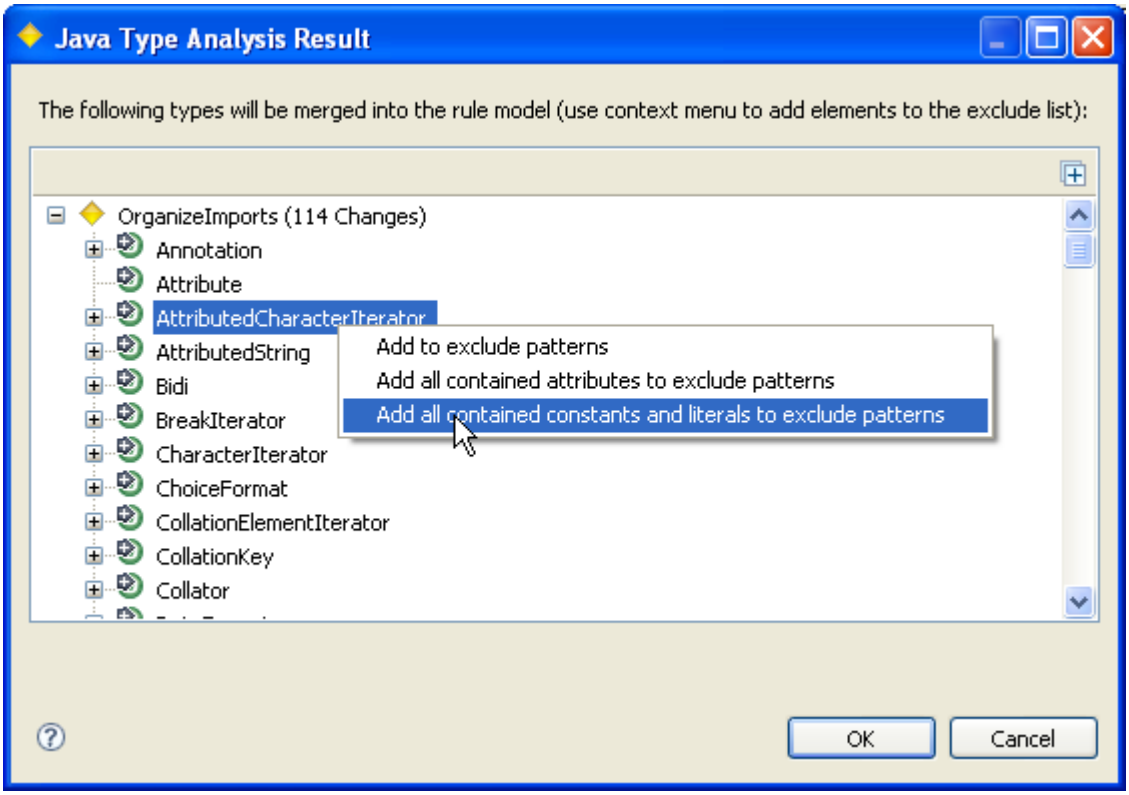
4. Optionally you can also define one or multiple exclude patterns to exclude specific packages or classes.

5. Select either the Hierarchical or Flat option to specify whether the import should create a rule Jpackage for each imported Java package (resulting in a rule package hierarchy looking exactly like the Java package hierarchy), or if all the imported classes will directly go into the rule model (resulting in a flat list of data types). In case you chose Flat, you may additionally define whether the constants should be put into the rule package or within the data types.

6. Select whether you want to include java doc "@" tags (for example `@author`, `@since` etc.) in the descriptions of the imported elements or not.

7. Press the Synchronize data types... button to start the import.

8. A preview dialog will appear showing you the changes that will result from the import.



9. Here you can right-click on a package, class, constant or even attribute and select Add to exclude patterns to exclude it from the import. When right clicking on a class you can also select All all contained attributes to exclude patterns to add a wildcard (*) exclude pattern for all the attributes in the class or Add all contained constants and literals to exclude patterns to add a wildcard (*) exclude pattern for all the constants and literals in the class.

10. After pressing OK the types shown in the preview will be merged into the rule model.

After setting up the Java type library as described above the import can be repeated to adopt the changes in the java classes by right clicking the rule model or rule package that has the type library configured and selecting Synchronize Java Type Library in the context menu.

All rule packages that use the imported data types will have to reuse the rule packages containing the imported data types. This is configured on the Reuse Packages tab in the Properties view.

Related Concepts.

- Java Type Library

Related References.

- Java Type Library Tab

Chapter 7. Mapping Data Types between Rules and XML

ACTICO Modeler can import data types from XML Schema (XSD). The import of XML Types from XML schema enables processing of XML data using rules and builds the cornerstone for more advanced use cases such as the integration of web services. The business user simply works with standard rule data structures (using rule expressions and functions he already is familiar with). This chapter explains the concepts and tasks involved for importing XML Types from XML Schema documents. Also rule data types can be exported as XML data types.

Since the XSD language is substantially more powerful and complex for defining data types, not all XSD defined types can directly be mapped to data structures.

The following list enumerates some of the XSD features constrained by ACTICO Modeler:

- top level element and attribute declarations are only imported if they are referenced from complex types
- model group particles of imported types must not define other particle attributes but 'minOccurs' and 'maxOccurs' to '1'
- types which utilize wildcards are imported but cannot be refined
- 'targetNamespace' is required
- types which extend other custom complex types can only be imported if the extending type does not redefine inherited components
- complex types with mixed content cannot be imported

Before importing a schema, in which some of the types utilize unsupported XSD features, you can selectively exclude these types using the XML type library.

Related Tasks.

- Importing XML Data Types
- Exporting Data Types

7.1. Concepts

7.1.1. XML Type Library

XML Type libraries can be defined on rule model as well as on rule package level. A single XML Type library defines from which XSD source files the types are imported and exposes a control which can be used to trigger an import. Additionally it provides advanced settings for customizing rule package names and defining which types are excluded from the import.

Related References.

- XML Type Library Tab

7.1.2. Representation of XML Schema Types

During the import of XSD types all XML schema built-in types are mapped to primitive types in the rules type system according to the following tables:

7.1.2.1. Built-in Types

Built-in Primitive Types

Table 7.1. Built-in Primitive Types

XML Schema Type	Rule Model Type	Java Type
xsd:time	Time	
xsd:string	String	java.lang.String
xsd:QName	String	java.lang.String
xsd:NOTATION	String (Type Alias)	java.lang.String
xsd:hexBinary	String	java.lang.String
xsd:gYearMonth	String	java.lang.String
xsd:gYear	String	java.lang.String
xsd:gMonthDay	String	java.lang.String
xsd:gMonth	String	java.lang.String
xsd:gDay	String	java.lang.String
xsd:float	Float	float
xsd:duration	String	java.lang.String
xsd:double	Float	double
xsd:decimal	Float	java.math.BigDecimal
xsd:dateTime	Timestamp	java.sql.Timestamp
xsd:date	Date	java.util.Date
xsd:boolean	Boolean	boolean
xsd:base64Binary	String	java.lang.String
xsd:anyURI	String	java.lang.String
xsd:anyType	Any	
xsd:anySimpleType	Any	

Built-in Derived Types

Table 7.2. Built-in Derived Types

XML Schema Type	Rule Model Type	Java Type
xsd:integer	Integer	java.math.BigInteger
xsd:nonPositiveInteger	Integer	java.math.BigInteger
xsd:nonNegativeInteger	Integer	java.math.BigInteger
xsd:negativeInteger	Integer	java.math.BigInteger
xsd:positiveInteger	Integer	java.math.BigInteger
xsd:unsignedLong	Integer	java.math.BigInteger
xsd:long	Integer	long
xsd:unsignedInt	Integer	long
xsd:int	Integer	int
xsd:unsignedShort	Integer	int
xsd:short	Integer	short
xsd:unsignedByte	Integer	short
xsd:byte	Integer	byte
xsd:normalizedString	String	java.lang.String
xsd:token	String	java.lang.String
xsd:language	String	java.lang.String
xsd:NMTOKEN	String	java.lang.String
xsd:Name	String	java.lang.String
xsd:NCName	String	java.lang.String
xsd:ID	String	java.lang.String
xsd:IDREF	String	java.lang.String
xsd:ENTITY	String	java.lang.String
xsd:NMTOKENS	String List	java.util.List<java.lang.String>
xsd:IDREFS	String List	java.util.List<java.lang.String>
xsd:ENTITIES	String List	java.util.List<java.lang.String>

7.1.2.2. Custom Types

The term "Custom Types" refers to the type definitions contained in the XSD source files, which are specified in the XML type library. During the import of these types target representations for the schema-defined components are generated. Here for each namespace a rule package is created which contains the respective types.

Custom Simple Types

ACTICO Modeler provides limited support for the import of custom defined simple types which are directly or indirectly derived from XSD built-in types. Such derivations are themselves typically defined by custom

simple types which restrict the value space of the XSD built-in types. During the import each custom simple type is mapped to a Type Alias. Here the documentation annotations of a XSD simple type are mapped to the description of the Type Alias.

Please note that although the value space could be restricted in XML schema, no such constraints are technically enforced in rule models. Therefore it is up to the user to ensure that only valid values are assigned to attributes.

```
<xsd:simpleType name="age">
    <xsd:restriction base="xsd:int">
        <xsd:minInclusive value="0"></xsd:minInclusive>
        <xsd:maxExclusive value="100"></xsd:maxExclusive>
    </xsd:restriction>
</xsd:simpleType>
```

When the schema above is imported into ACTICO Modeler a Type Alias named 'Age' is created which uses an 'Integer' as basic data type and its Java Type is set to the primitive Java Type 'int'. The constraint that the type 'Age' only includes values between 0 and 100, is not technically enforced by the rule runtime.

Example 7.1. Example

An exception to the general handling of simple types are simple types which are derived from the XSD built-in type 'token' and have enumeration facets. Such a type is mapped to an Enumeration with one literal for each enumeration facet. This special mapping is applied to ensure consistency with XML schema documents generated by ACTICO Modeler during publishing of a rule model to the ACTICO Execution Server.

Custom Complex Types

Custom XSD complex types are mapped to structures. XSD elements and attributes are mapped to attributes of these structures.

In this context ACTICO Modeler determines the valence of the target attribute. For XSD attributes the valence is determined by evaluation of the valence of the XSD type of the attribute. If the XSD type is derived from a list the mapped attribute is defined to be multi-valued. In contrast for XSD elements the type as well as the particle of the element is inspected. If the type of the element is a multi-valued simple type or the particle states that the element may appear more than one time, then the target attribute will be defined to be multi-valued (a type that has elements fulfilling both conditions cannot be imported). If the type of the element is an anonymous complex type with a sequence of exactly one element, which according to its particle may appear more than one time, then the mapped attribute also will be defined as multi-valued. Two common formats used to represent data from lists in XML (wrapped/unwrapped) are supported.


```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.org/valenceExample"
            xmlns:tns="http://www.example.org/valenceExample" elementFormDefault="qualified">

    <xsd:complexType name="ValenceExample">
        <xsd:sequence>
            <xsd:element name="singleValued" type="xsd:int"></xsd:element>
            <xsd:element name="multiValued_unwrapped" type="xsd:int" maxOccurs="unbounded">
            </xsd:element>
            <xsd:element name="multiValued_wrapped">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="listElement" type="xsd:int" maxOccurs="unbounded">
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

For the type 'ValenceExample' a structure named 'ValenceExample' is created during the import which has three attributes. The first attribute is named 'singleValued' and is not multi-valued since it has the single-valued type 'int' and does not have a multi-valued particle (since the particle is not explicitly defined 'minOccurs' and 'maxOccurs' both default to 1). The other attributes both are defined multi-valued since the first has a multi-valued particle definition and the second matches the special wrapped pattern described above.

Example 7.2. Example

Another specialized mapping is applied to support a common XML schema representation of Map-style data structures. Therefore ACTICO Modeler recognizes certain element declarations in complex types as Map-attributes. Such element declarations must have an anonymous complex type that declares exactly one element in its content, which according to its particle may appear more than one time. Additionally the type of this element must be an anonymous complex type that declares exactly two elements. The first element must be named 'key' and according to its particle may not occur more than one time. The second element must be named 'value' and again may not occur more than one time.

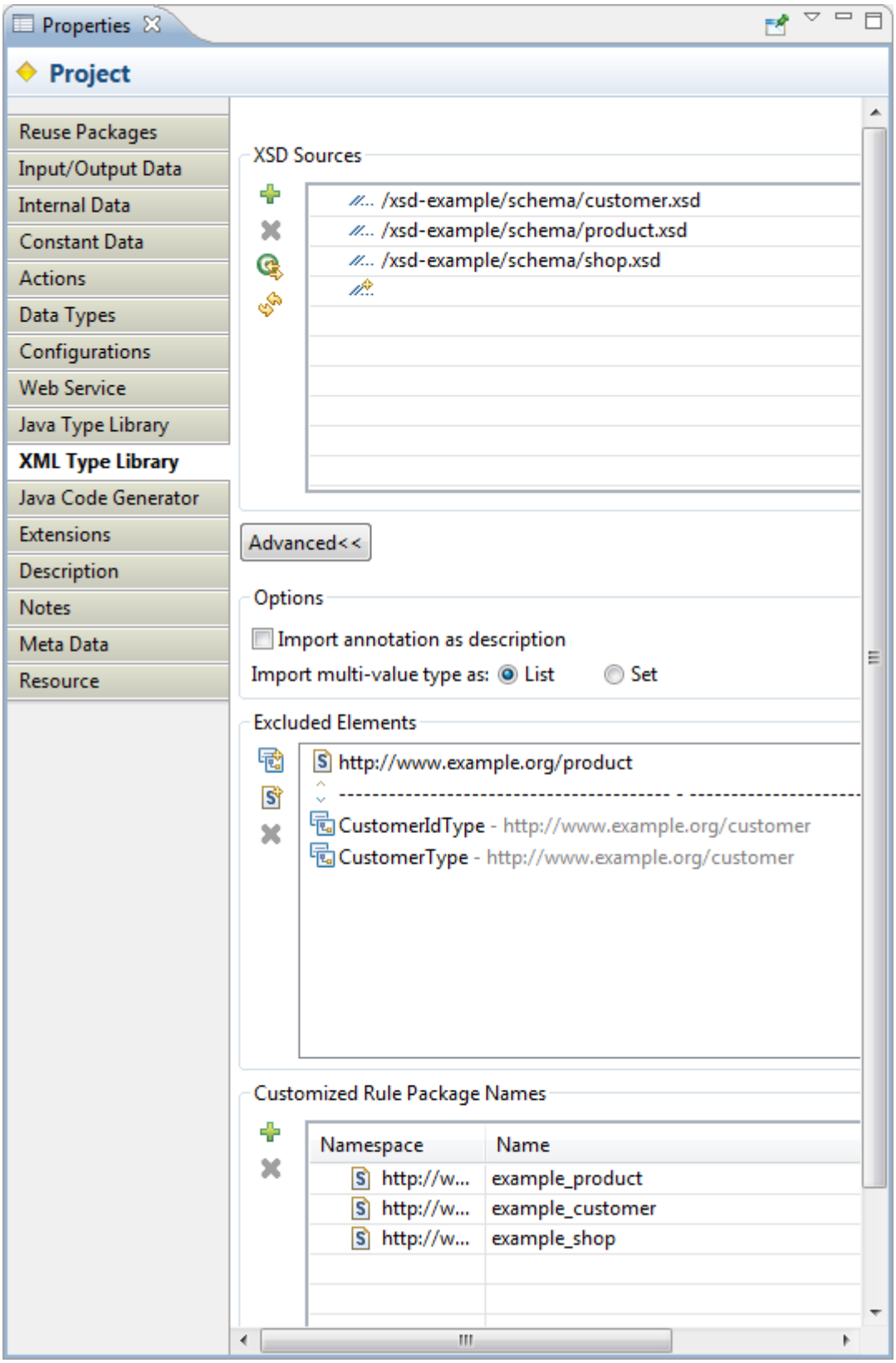
```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/mapExample"
        xmlns:tns="http://www.example.org/mapExample" elementFormDefault="qualified">

    <complexType name="MapExample">
        <sequence>
            <element name="mapAttribute">
                <complexType>
                    <sequence>
                        <element name="entry" maxOccurs="unbounded">
                            <complexType>
                                <sequence>
                                    <element name="key" type="string"></element>
                                    <element name="value" type="integer"></element>
                                </sequence>
                            </complexType>
                        </element>
                    </sequence>
                </complexType>
            </element>
        </sequence>
    </complexType>
</schema>
```

Example 7.3. Example

7.2. Tasks

7.2.1. Importing XML Data Types

1. Select a rule package where you want the imported data types to be located.
2. Switch to the XML Type Library properties tab.



The XML Type Library tab is not available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective first.

3. Add the relevant XSD sources by clicking on  .

4. Click on  to start the import. For each individual namespace a rule package will be created.

Once the data types have been imported, you can click on  to synchronize them.

When clicking on Advanced >>, you also have the following options:

- Selecting whether documentation annotations in the XSD sources will be imported as descriptions
- Selecting whether a multi-value type will be imported as a list or as a set
- Specifying namespaces and types, which will be excluded from the import (Types which depend on an excluded type are excluded automatically)
- Change the default names for rule packages resulting from the import

Related Concepts.

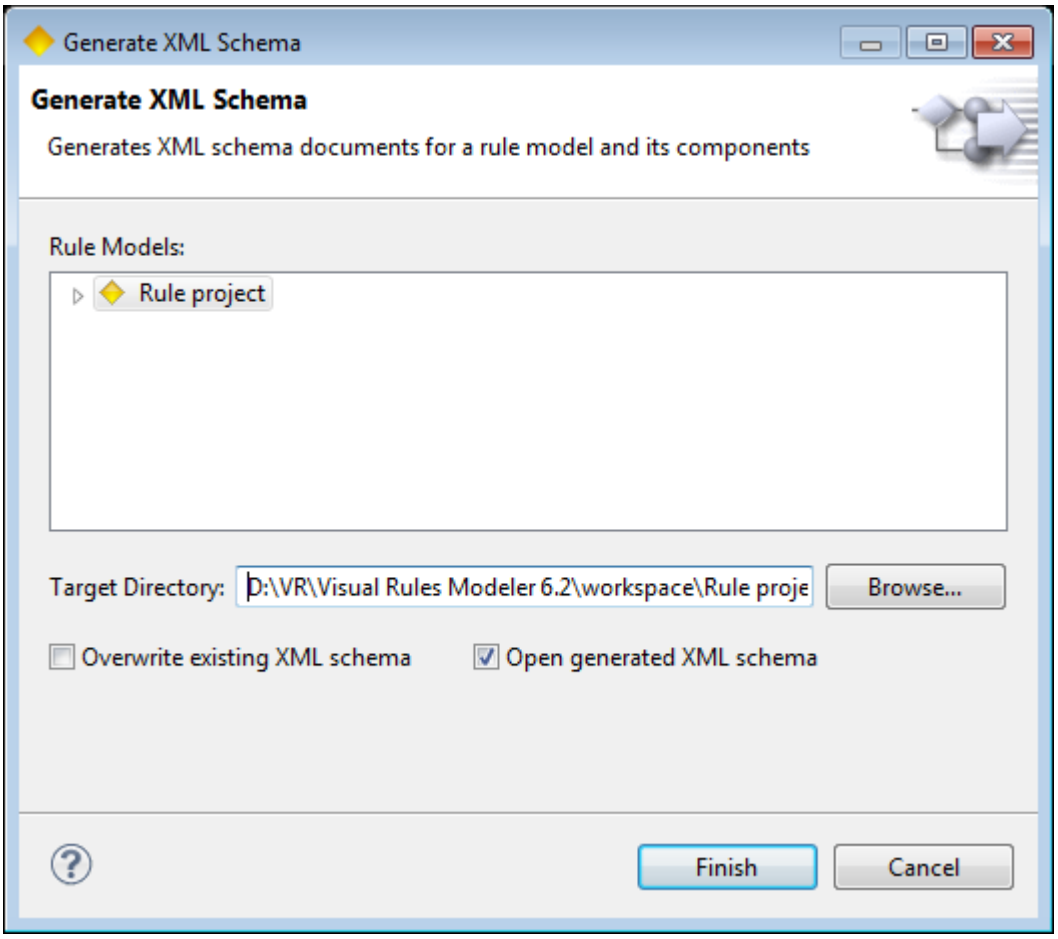
- XML Type Library

Related References.

- XML Type Library Tab

7.2.2. Exporting Data Types

1. Right-click on the rule model, from which you want to export data types.
2. In the context menu, select Export > XML Schema.



This dialog is not available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective first.

3. In the field Target Directory, specify a location, where you want the generated XML schema to be placed.

You can browse your workspace directory by clicking on Browse.

4. Select, whether you want an existing XML schema in the target directory to be overwritten, if so.
5. Select, whether you want an editor to be automatically opened for the generated XML schema.
6. Click on Finish to start the export.

Chapter 8. Extending ACTICO Modeler

ACTICO Modeler has multiple customization options that allow to extend its functionality. With ACTICO Modeler you can provide and register your own Java implementations of functions, actions and services. These are then fully integrated into the graphical modeling environment, so a rule author can make use of these custom extensions just in the same way as of the built-in ones.

These are the customization options which are described in the following sections.

- Custom functions
- Custom action types
- Custom user interfaces for these action types
- Custom service types
- Custom user interfaces for meta data
- Integration into the Java service landscape

8.1. Concepts

8.1.1. Custom Function

A custom function is a user provided function that can be used in rule expressions just like any other of the built-in functions. Custom functions must be registered in a rule model in order to become available to the rule authors. The custom functions will appear in the content assist, and also provide tooltips and will be syntax highlighted just like any other function.

Custom functions can be used for many different purposes. For example they may encapsulate complex computations or even provide access to some external data, e.g. a data warehouse.

Custom functions are implemented in Java. They can take any number of parameters of any Java type, including object types and collections. The result of a custom function can also be of any Java type.

Any static method of a Java class can become a custom function in ACTICO Modeler. The only restriction is that it may not throw a checked exception. The Java class containing the function implementation must be on the classpath of the project in order for the generated rule code to compile without error. And of course it must also be available at runtime when the rules are executed. It is the developer's responsibility to configure the classpath accordingly. Often the source code for the custom function is simply put into an additional source folder within the project that contains the rule model where the function is registered.

Related Tasks.

- Defining a custom Function

8.1.1.1. Context aware Functions

Usually functions are stateless and don't have to know anything apart from the parameters they receive. But sometimes functions need to store state information or know about the context in which they are called. ACTICO Modeler supports the concept of such "context aware" functions. When a function is registered in a rule model it can be defined to be "context aware".

A context aware function will receive an additional parameter of type `IContextAwareRequestData` (this is always the last parameter in the function's signature). This object gives the function access to the complete request data that the current rule is processing. This includes - among other things - all input/output data elements (parameters), internal data elements (variables) and actions.

It also provides the model path (`getModelPath()`) of the current rule being executed. And it even provides the ID (`getNodeId()`) of the currently executed node (which corresponds to a specific rule element in the flow rule).

Most importantly, it gives access to the `IRequest` via `getRequest()`, and from there it is possible to access the `ISession` (via `getSession()`). And the session can be used to associate any state information the function may want to store or have access to. See methods `putUserObject()` and `getUserObject()` for this purpose. The Group Id, Artifact Id and Version of the current Rule Service being executed are stored in the statistics and can be retrieved via `getRequest().getSession().getStatistics().getMetaData()`. The values of the returned `Map` can be accessed using the keys stored in `IStatistics.METADATA_ARTIFACT_GROUP`, `-.METADATA_ARTIFACT_NAME` and `-.METADATA_ARTIFACT_VERSION`.

By the way, the same `IContextAwareRequestData` object is also provided to custom action types when they are executed. So actions can also store state in a session, which could then later be retrieved by some context aware functions. This may prove useful in situations where specific actions and functions need to know about each other.

8.1.2. Custom Action Type

A custom action type is a user provided action type that allows to execute custom Java code when an action is fired. Custom action types must be registered in a rule model to be available to rule authors. Once they are registered, a rule author can create actions of this type, configure and fire them from within rules.

An action type defines the exec parameters that have to be specified in the Fire Action element. It also defines the functional init parameters that have to be specified for every action of this action type. An action type has one or more Java implementations (Java classes implementing the `IAction` interface). Each of these implementations can define technical init parameters. An action selects one of the implementations (for each of possibly multiple configurations) and then provides values for the corresponding technical init parameters. These relations are illustrated in the following figure.

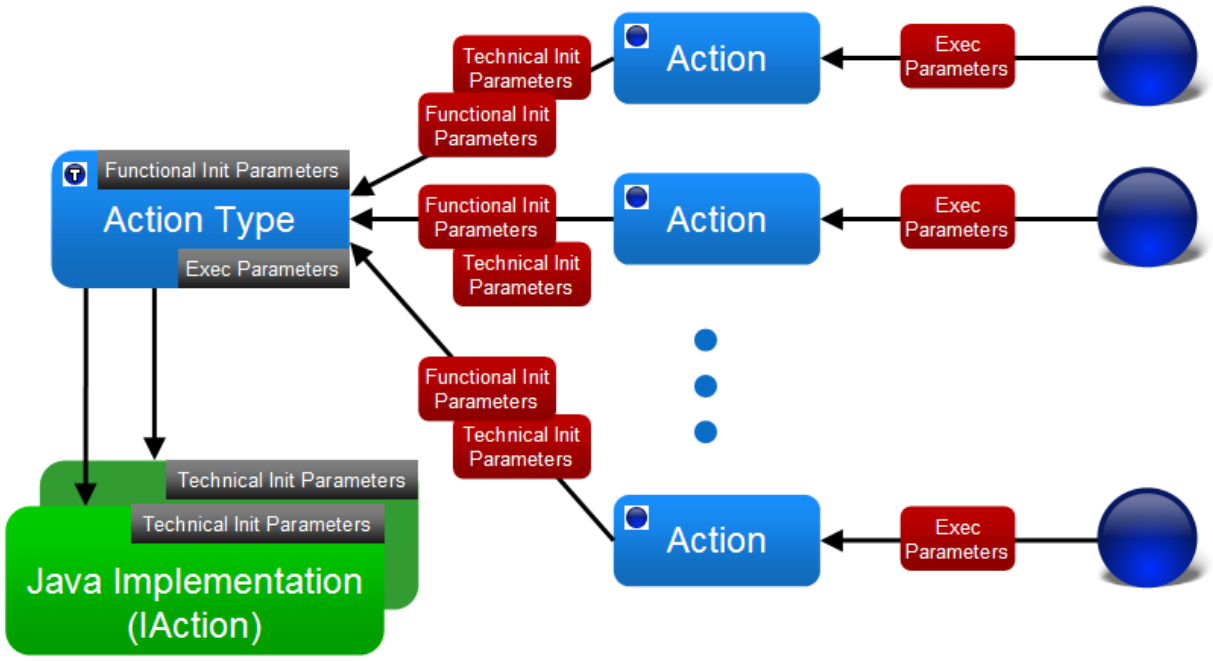
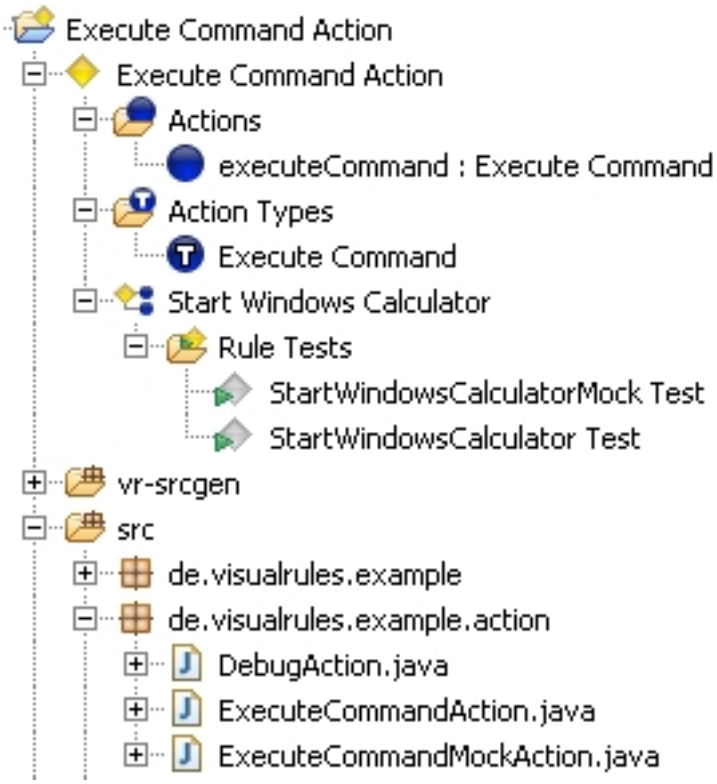


Figure 8.1. Relationships between Action, Action Type and Java implementation

The following screenshot shows a rule model with a custom action type "Execute Command". The action type () is defined on the rule model. An action named "executeCommand" of that custom action type is also defined and fired in the "Start Windows Calculator" rule.

The Java code for the custom action can be found in `ExecuteCommandAction.java`.

The screenshots here are taken from the "Execute Command Action" example that is shipped with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace.



Related Tutorials.

- Custom Action Type Tutorial

Related Tasks.

- Defining a custom Action Type

8.1.3. Action Life Cycle (`IAction`)

A developer of a custom action type needs to understand the action life cycle that actions go through during rule execution. The life cycle is reflected by the methods of the `IAction` interface, like `initFunctional()`, `initTechnical()`, `enter()`, `leave()`, `sessionClosed()` and `execute()`. These methods will be called by the runtime at specific points in time.

An important factor in an action's life cycle is the location within the rule model where it is defined. It can be defined either globally on a rule model, or on some rule package or on some specific rule. The action will be initialized whenever rule execution enters the context where the action is defined, i.e. whenever the action comes into scope.

A globally defined action therefore will be initialized once at the beginning of each request. An action defined on a rule will be initialized every time that rule is called. And an action defined on a rule package is initialized every time rule execution enters the package, which happens when a rule within that package (or in any sub package) is called from outside the package.

Each time an action is initialized, the methods `enter()`, `initFunctional()` and `initTechnical()` will all be called, in exactly this order.

Then, whenever the action is fired, the method `execute()` is called. This may happen any number of times or not at all, depending on how often the rules fire the action while it is in scope.

When rule execution leaves the scope of the action, the method `leave()` will be called. Again, for a globally defined action (on the rule model) this happens once right before control is given back to the client (end of request). For an action defined on a rule package or a rule, this happens when the rule package or rule is left, respectively.

Finally, the `sessionClosed()` method is called when the client closes the session. This usually is a good time to clean up resources, close files or flush cashes.

Table 8.1. `IAction` life-cycle methods

Method	Description
`initFunctional(Map functionalInitParameters)`	Called when the action comes into scope.
`initTechnical(Map technicalInitParameters)`	Called when the action comes into scope, right after `initFunctional()` was called.
`enter(IRequest request)`	Called when the action comes into scope, right after `initTechnical()` was called.
`execute(IContextAwareRequestData requestData, Map execParameters)`	Called when the action is fired.
`leave(IRequest request)`	Called when the action gets out of scope. Called even when the action was not fired (`execute()` was not called).
`sessionClosed()`	Called when the rule session is closed with `closeSession()`.

Related Tasks.

- Defining a custom Action Type

8.1.4. Custom Service Type

Custom service types are user provided service implementations that are called when a rule calls a service of that type. The implementation is provided as a Java class implementing the `IService2` interface or its predecessor `IService`. Implementations can be exchanged at runtime as well, which is described in Exchanging Service Implementations. Custom service types must be registered in a rule model to be available to rule authors. Once they are registered, a rule author can create services of this type, configure and call them from within rules. This is illustrated in the following diagram.

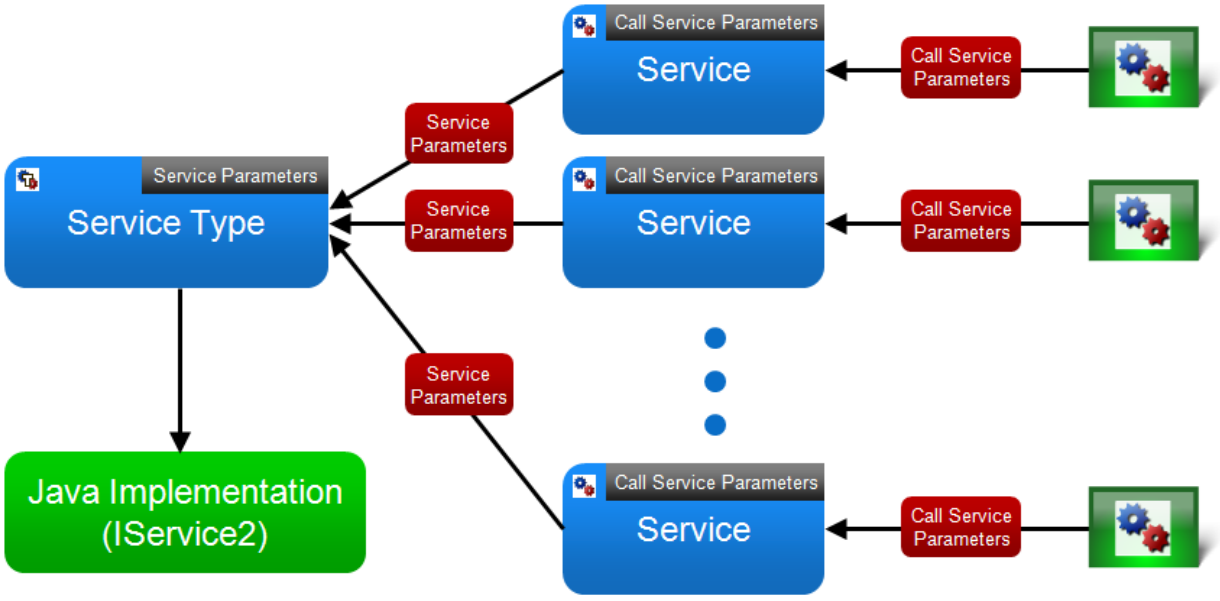


Figure 8.2. Relationships between Service, Service Type and Java implementation

Every service defines its input and output parameters. Therefore, each service can have different inputs and outputs even when they are of the same type. A rule calling a service will have to specify values for each of the input parameters.

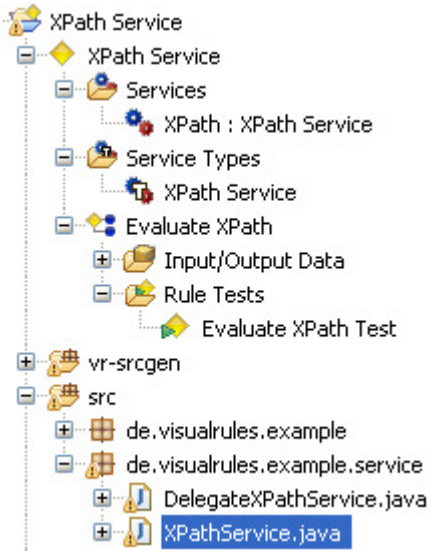
The service type can optionally define several service parameters. These are technical parameters that the implementation requires to distinguish between the different services of the same type. Each service defines values for these service parameters.

Both the parameters specified by the Call Rule element and the parameters specified by the service are passed on to the Java implementation. The Java Implementation reads the inputs, performs the actual service call and returns the results.

The following screenshot shows a rule model with a custom service type "XPath Service". The service type is defined on the rule model ( ). A service named "XPath" of that custom service type is also defined and called in the "Evaluate XPath" rule.

The Java code for the custom service type can be found in `XPathService.java`.

The screenshots here are taken from the "XPath Service" example that is shipped with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace.



Often, a custom Service Type is used to integrate existing functionality. In this case a Service can be imported when it is based on a Java interface. See Integration into the Java Service Landscape for more information.

Related Tasks.

- Defining a custom Service Type

8.1.5. Custom Meta Data

Rule models, rule packages, rules, input/output data elements, constant data element, enumerations, exceptions, structures and attributes can have custom meta data associated with them. To edit meta data inside ACTICO Modeler, the Meta Data property tab can be used.

8.1.6. Integration into the Java Service Landscape

There are two different scenarios for integrating rules into the Java service landscape:

- The integration of rules as service into existing Java applications.
- The use of provided Java services in rule models.

These are typical tasks of a service oriented architecture (SOA). Functionalities are offered as services which can be used in other applications. The communication between service user and service provider is managed by an interface. This interface only describes the service and contains no details about the implementation.

In Java, such a service interface is implemented as a Java interface. A service offered by the service provider must implement this interface. The service user can call interface methods, which will be executed by the service implementation.

Rules are able to implement an interface and thus can be called as normal Java services. On the other hand it is possible to call Java services inside a rule using services and service types with a special service implementation. Both scenarios are described below.

8.1.6.1. Use of Services based on a Java Interface

This section describes the use of Java services in rule models. Services in rule models have a different concept than Java services. Services are executable units with input and output parameters and correspond to an interface method in Java. Therefore the service implementation has to execute the Java service itself. Due to the fact that one Java interface contains several methods, there can exist more than one service for one Java interface.

Concept

IACTICO Modeler has a wizard for creating services based on a Java interface. You can open this wizard via Import... > ACTICO Modeler > Java interface as services. There you can choose the Java interface as well as all Java methods to be used. Based on all the information, the assistant creates and configures services in a rule model. The necessary input and output service parameters and required structures are created automatically by importing the Java object model of the Java service. Finally the service settings will be added to ensure that the service can be executed.

The service implementation requires various information to find and execute suitable Java services. All this information is located in the service settings including the fully qualified name of the Java interface (the service interface), the used methods as well as the service properties. The service properties are intended to be user defined and passed to the service implementation. They contain additional information like the address and version of the service and can be used for its discovery.

Often the service implementation is only available at runtime and not at the time of rule modeling. Created services will automatically be assigned to the service type "Java Integration" so that testing of rule models is still possible. This service type does not execute any Java service, but returns default values for the output parameters and therefore is only suitable for testing purposes.

Implementation

A service implementation must be provided to call a Java service from rules. The abstract base class "AbstractJavaIntegrationService2" already provides all basic functionalities for calling Java services. Only the discovery of the service has to be implemented. The following code describes such an implementation.

The example here is taken from the "Rule Execution API Example" that is shipped with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace. See `CustomerJavaIntegrationService.java` to find the following code.

```
 1 public class CustomerJavaIntegrationService extends AbstractJavaIntegrationService2
 2 {
 3    /**
 4     * The abstract method findService must be overridden and return an instance of the
 5     * service.
 6     */
 7    protected Object findService(IContextAwareRequestData requestData, String
fullyQualifiedName, Map serviceProperties)
 8       throws Exception
 9    {
10       System.out.println("Entered CustomerJavaIntegrationService.findService.");
11       System.out.println();
12       return new CustomerServiceImpl();
13    }
14 }
```

The service implementation must extend the class "AbstractJavaIntegrationService2" and also implement the method `findService()`.
The method `findService()` is used to find the desired Java service. As return value this method expects an instantiated Java object which implements the given Java interface.

In this case the Java class is directly instantiated. However this is a simple scenario. In Java service landscapes this can be far more complex, for example the service may be discovered dynamically using a service registry.

If the service implementation returns no service instance, a "ServiceNotFoundException" is thrown at runtime. This exception extends the "GeneralException" and so can be handled in the calling rule.

The service implementation now can be exchanged and used as described in Exchanging Service Implementations.

There is an exemplary rule call with an exchanged service implementation in the "Rule Execution API Example" that comes with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace. The class `ExampleJavaServiceIntegration.java` shows the corresponding rule call.

Related Concepts.

- Custom Service Type

Related Tasks.

- Exchanging Service Implementations

8.1.6.2. Using Rules based on Java Interfaces

This section describes how to offer a rule model as Java service. There exist two different approaches for integrating services:

- Contract-Last: The service interface will be generated from the service implemenation.
- Contract-First: A service will be created on the basis of an well defined interface.

The approach "Contract-First" is used for the integration of rules in the Java service landscape. For this the rule model has to be created based on a Java interface. Rules and methods of the Java interface must have the same input and output parameters.

Concept

ACTICO Modelers has a wizard for creating rules based on a Java interface. Use the Import... > ACTICO Modeler > Java interface as rules to open this wizard. It creates rules including the input and output parameters and additionally saves information about the used interface.

You can see all information about the used interface in the Service Integration tab of the Properties view.

The Service Integration tab is not available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective.

It's not necessary to implement a rule for every service method. If a method is called which is not available or implemented multiple times inside a rule model, an exception is thrown at runtime. Rule models can also contain implemented methods of different Java interfaces.

Because rules are created based on the signature of a Java method and implement this method, these rules are frozen. This means that input and output parameters of these rules cannot be changed.

Service Call

ACTICO Modeler offers a general service implementation for rules which are implemented by the wizard described before. This implementation is implemented with a dynamic proxy and forwards method calls to

corresponding rules. So the dynamic proxy implements all interfaces implemented by rules. You can create a dynamic proxy instance using the interface `IJavaServiceInterface`. The following example shows such a Java service call.

The example here is taken from the "Rule Execution API Example" that is shipped with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace. See `CustomerJavaIntegrationService.java` to find the following code.

This example describes the use of the service interface `bom.service.ICustomerService` and the service method `welcomeCustomer()`. The service method was imported as the rule "welcomeCustomer" into the rule model "Example Rules" and can be called via the service implementation. See `ExampleCallRuleAsJavaService.java` to find the following code.

```
1          // Creates a new instance of IJavaServiceIntegration.
2          IJavaServiceIntegration javaServiceIntegration = JavaServiceIntegrationFactory
3             .createJavaServiceIntegration(Example_RulesRuleModel.INSTANCE);
4
5          // Checks whether the service interface "ICustomerService" can be instantiated or
6          // not.
7          if (javaServiceIntegration.canInstantiateService(ICustomerService.class))
8          {
9             // Creates an instance implementing all interfaces referring to a rule.
10            // Casts this instance to the interface "ICustomerService".
11            ICustomerService service = (ICustomerService) javaServiceIntegration
12               .newServiceInstance();
13
14            Customer customer = new Customer();
15            customer.setName("John");
16
17            // Calls the service method which internally executes the rule
18            // "welcomeCustomer".
19            service.welcomeCustomer(customer);
20         }
```

The class `JavaServiceIntegrationFactory` creates an instance of `IJavaServiceIntegration`. The parameterized constructor needs an instantiated rule model. This rule model contains rules implementing methods of a Java interface.

In this example, an instance of the rule model "Example Rules" is passed. This rule model contains the rule "welcomeCustomer", which implements the method `welcomeCustomer()` of the interface `bom.service.ICustomerService`.

It's possible to use a constructor with a specific `ClassLoader`, which is used to load the implemented Java interfaces.
The method `canInstantiateService(Class)` is used to check whether a service interface can be called. So it can be tested if the wished interface is implemented by a rule out of the passed rule model as well as can be loaded with the used `ClassLoader`.
The method `newServiceInstance()` creates a service instance. This instance can be cast to every implemented interface.

In this example the service instance is cast to the interface `bom.service.ICustomerService`.

Optionally an object of the type `AbstractJavaServiceAdvisor` can be passed to this method. This class offers the possibility to customize session and request creation and for example can be used to attach listeners.
Now you can call the methods of the interfaces. The proxy object internally searches for a suitable rule and executes this rule. If the method has a return parameter, the return parameter of the rule will be passed through the method to the user.

In this example, the method `welcomeCustomer()` is called. To this method an instantiated `Customer` object is passed. The proxy object executes the rule "welcomeCustomer" and prints a welcome message on the console.

Related Tasks.

- Calling Rules with the specific Rule Execution API

- Calling Rules with the generic Rule Execution API

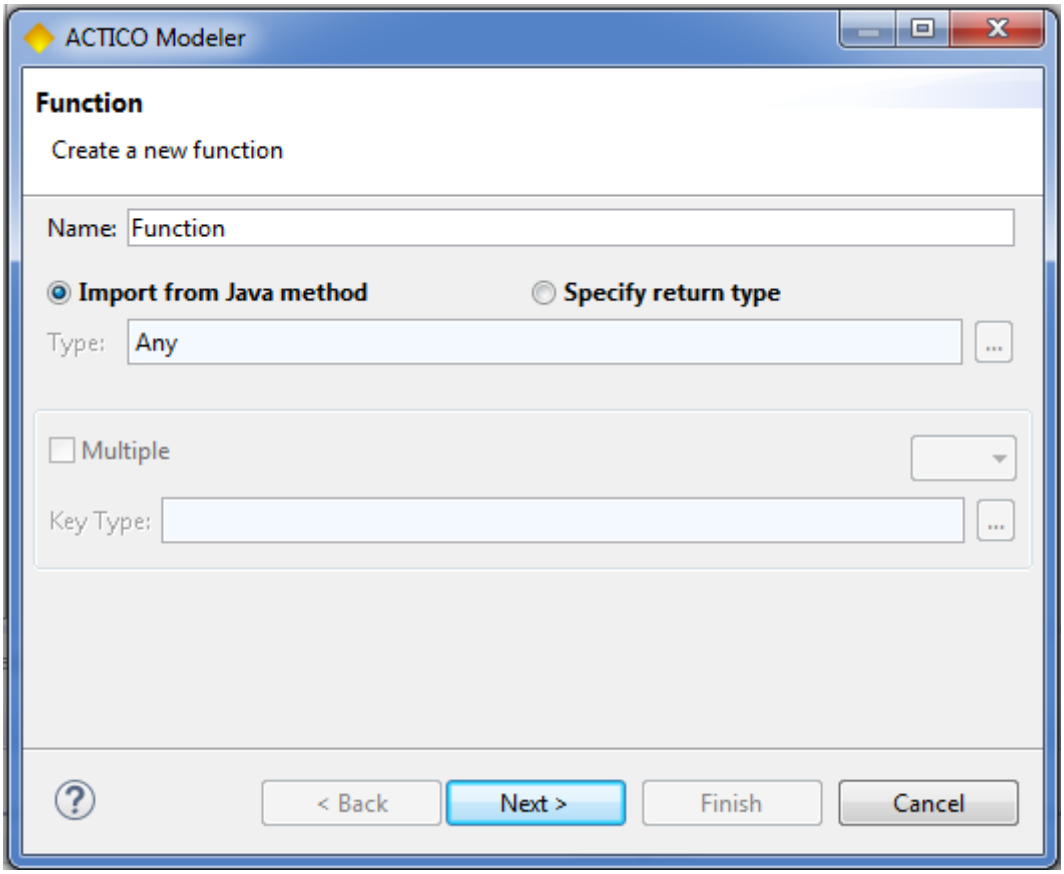
8.2. Tasks

8.2.1. Defining a custom Function

Any static method of a Java class can be registered within ACTICO Modeler to become a custom function.

1. Right-click on a rule package or rule model and select New Element > Function.

   The following dialog appears:



2. Provide a name for the function. This does not necessarily have to match the Java method name.
3. Select Specify return type.
4. Specify the return type of the function.
5. Click on Finish.
6. Use the Function tab in the Properties view to specify the function parameters. The number, order and data types of these parameters must match the Java method's parameters.
7. Use the Java Implementation tab to add at least one implementation. Specify the fully qualified class name of the implementation class. You can use Browse... to open a class browser.
8. Specify the name of the method in that class. You can use Browse... to open a method browser.
9. Specify the exact method signature as a semicolon separated list of the Java types of the parameters. Use fully-qualified Java names for class/interface type parameters. For example: `int; double; java.lang.BigDecimal; com.acme.Customer; java.util.List; java.lang.Object`
10. Check the Context Aware checkbox if your function must know the context from where it is called. It will then have to have an additional parameter of type `IContextAwareRequestData`. Through this interface

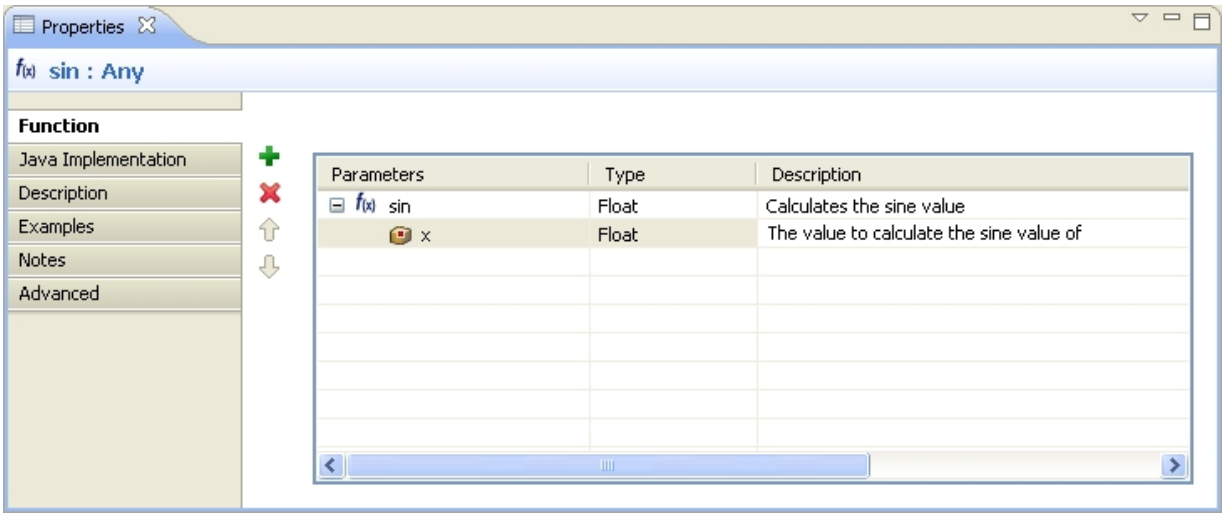
the function will have access to the current rule being executed, the node id, and all data elements, the request and the session (see Context aware Functions).

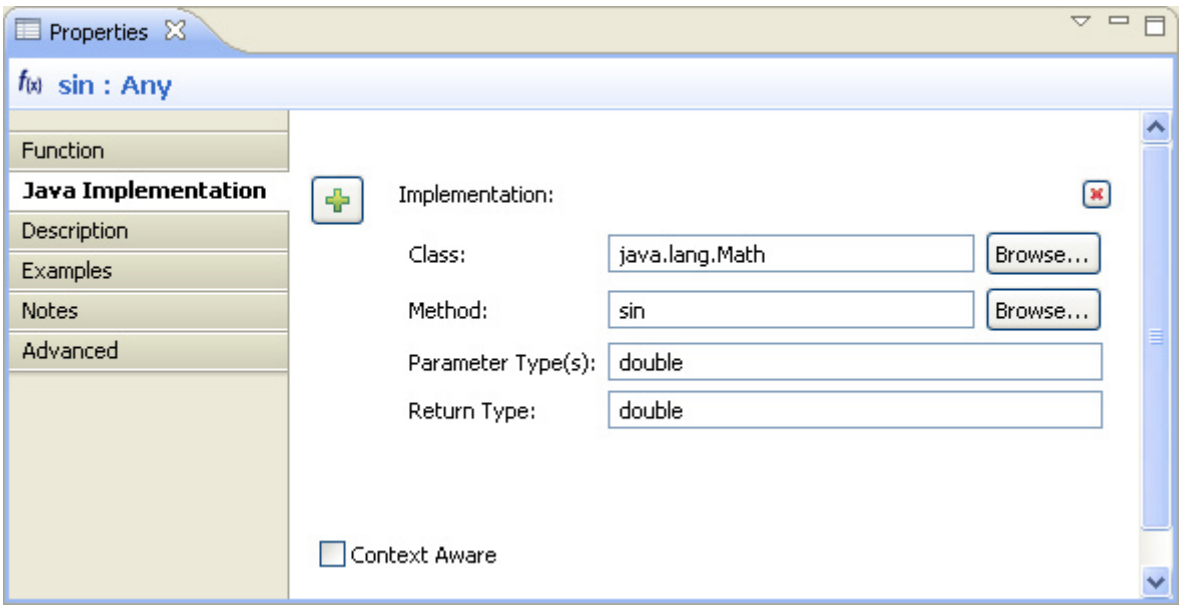
11. Specify the Java return type of the method. Again, you must use a fully-qualified Java name for a class/interface return type.

12. You should also enter a description for the function and possibly provide one or several examples for its usage. They are available to the users from within the content assist. This can be done on the Description and Examples tabs, respectively.

It is possible to provide multiple Java implementations for one function. Typically if there are multiple methods which take similar types of parameters, e.g. either a `double` or a `java.math.BigDecimal`.

The code generator will always use the implementation which matches best in each case. It will select the implementation that does not require it to do any type conversion and if so, it tries to do it with the lowest possible loss of precision.





Example 8.1. Definition of a custom function "sin"

Related Concepts.

- Custom Function
- Context aware Functions

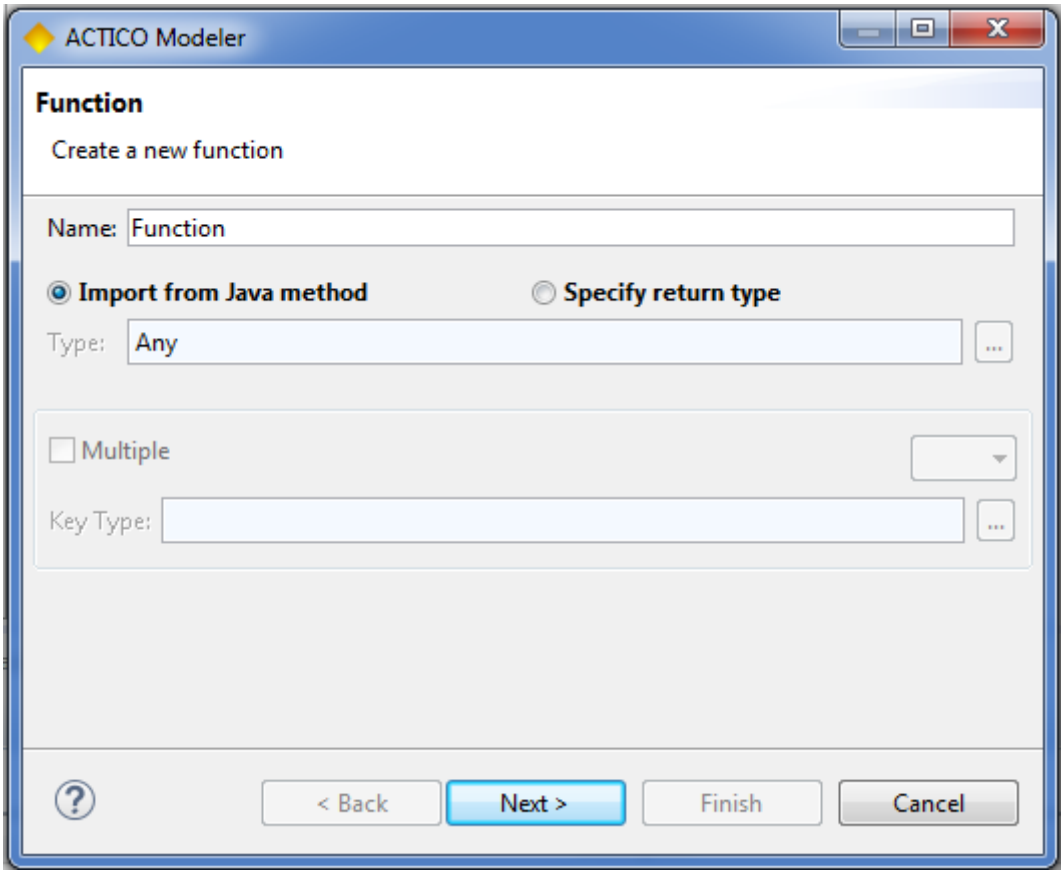
8.2.2. Importing a custom Function

If there is a JavaDoc provided for the relevant custom function, then by importing the function this is automatically imported as a description (in the Properties view).

To import a custom function, do the following:

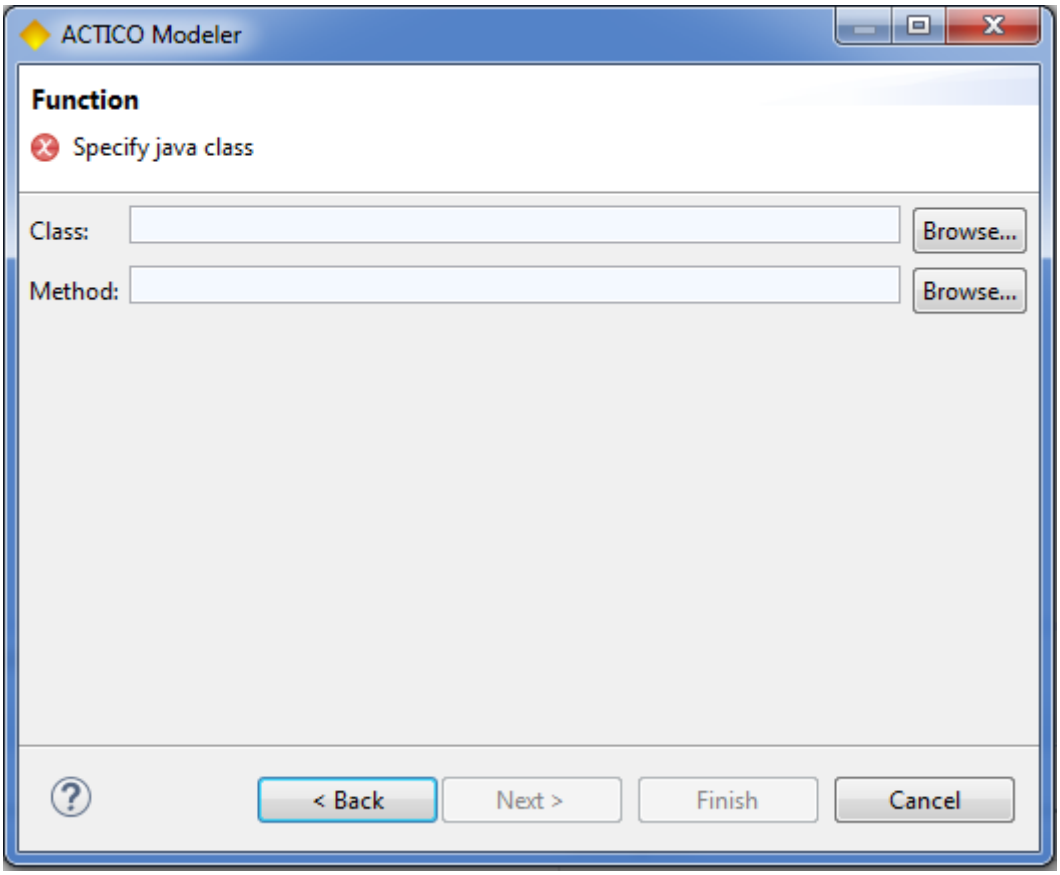
1. Right-click on a rule package or rule model and select New Element > Function.

   The following dialog appears:



2. Select Import from Java method.
3. Click on Next.

   The following dialog appears:



4. Specify the Java class and method.
5. Click on Finish.

Related Concepts.

- Custom Function
- Context aware Functions

8.2.3. Defining a custom Action Type

To define a custom action type you need to provide a Java class implementing the desired functionality. This class is then registered by defining the action type in a rule model or package. An action type definition also defines all the parameters that need to be specified for an action (functional and/or technical init parameters), and when firing an action of that type (exec parameters).

Related Tutorials.

- Custom Action Type Tutorial

8.2.3.1. Define the custom Action Type in a Rule Model

1. Right-click on a rule package or rule model and select New Element > Action Type.
2. Provide a name for the new action type.
3. Use the Action Parameters tab in the Properties view to specify the parameters that an action of your action type will need to initialize. A user must specify values for these parameters when he or she defines an action of the custom action type. These values will later be delivered to the action instance when `initFunctional()` is called.

4. Use the Fire Action Parameters tab to specify the parameters that your action will need to execute. A user must specify values for these parameters within the Fire Action element. These values will be delivered to the action instance when `execute()` is called.

Related Concepts.

- Custom Action Type

8.2.3.2. Write the custom Action Type Implementation

1. Create a Java class that implements the `IAction` interface. The fully-qualified name is `de.visualrules.runtime.IAction`. During rule execution, an instance of this class is instantiated when an action of your custom action type comes into scope. An action comes into scope when execution enters the package or rule where the action is defined.

2. Implement the `initFunctional()` method. This method is called when the action comes into scope. It will receive the values for the action parameters (functional init parameters). If the action does not require such initialization, you can leave the method empty.

3. Implement the `initTechnical()` method. This method is called when the action comes into scope. It will receive the values for the technical action parameters (technical init parameters) specific for the currently selected configuration. If the action does not require such initialization, you can leave the method empty.

4. Implement the `enter()` and `leave()` methods. These methods are called when the action comes into scope or leaves scope, respectively. If the action does not have to prepare or and unprepare at these moments, you can leave these methods empty.

5. Implement the `execute()` method. This method is called every time the action is fired. It will receive the current `IContextAwareRequestData` object and the values of the parameters specified in the Fire Action element (exec parameters).

6. Implement the `sessionClosed()` method. This method is called once the rule session (`ISession`) is closed. If the action does not have things to clean up, you can leave the method empty.

Action types can have multiple different Java implementations. Every implementation will have to be a separate Java class implementing `IAction`.

Related Concepts.

- Action Life Cycle (`IAction`)

8.2.3.3. Register the Java Implementations with the Action Type

1. Select the action type and switch to the Java Implementation tab in the Properties view.

2. Define a symbolic name for the implementation. Each implementation must have a different name. The user will select which implementation to use by specifying this symbolic name.

3. Enter the fully qualified class name of the Java class with the action type implementation (`IAction`)

4. Define the technical initialization parameters specific to this implementation. The user will have to specify values for these parameters when he or she selects the implementation to be used for a specific action instance. These values will later be delivered to the action instance when `initTechnical()` is called.

5. Add additional implementations and configure them accordingly.

8.2.4. Defining a custom Service Type

To define a custom service type you need to provide a Java class implementing the `IService2` interface. This class is then registered with by defining the service type in a rule model or package. The service type optionally defines service parameters that need to be specified by each service of this service type.

Look at the "XPath Service" example that comes with ACTICO Modeler. Use the New > Example... wizard to import the example into the workspace.

Related Concepts.

- Custom Service Type

8.2.4.1. Define the custom Service Type in a Rule Model

1. Right-click on a rule package or rule model and select New Element > Service Type.
2. Provide a name for the new service type.
3. Use the Service Parameters tab in the Properties view to specify the parameters that a service of your service type will need to provide. A user must specify values for these parameters when he or she defines a service of the custom service type. These values will later be delivered to the service instance when `execute()` is called.

8.2.4.2. Write the custom Service Type Implementation

1. Create a Java class that implements the `IService2` interface. The fully-qualified name is `de.visualrules.runtime.IService2`. During rule execution, an instance for each service of your custom service type is instantiated.
2. Implement the `execute()` method. This method is called every time a service of this type is called. It will receive the current `IContextAwareRequestData` object, and both the values of the service parameters specified on the service, and the values of the parameters specified in the Call Service element. It will also receive a `IServiceCallMetaData` object, specifying the names and types of all the expected input and output parameters.

8.2.4.3. Register the Java Implementations with the Service Type

1. Select the service type and switch to the Java Implementation tab in the Properties view.
2. Enter the fully qualified class name of the Java class with the service type implementation (`IService2`).

8.2.5. Defining a Service

1. Right-click on a rule package or rule model and select New Element > Service.
2. In the New Service wizard, enter a name for the new service and select the desired service type. Click Finish to create the new service.
3. Go to the Service Settings tab in the Properties view and configure values for the service parameters required by the service type.
4. Go to the Service Call Parameters tab to specify the input and output parameters for this service. Values for the input parameters will have to be specified in Call Service elements calling this service. The output parameters will be received by the Call Service element as result.

Now this service can be called by a flow rule via the Call Service element.

Related Concepts.

- Custom Service Type

8.3. Tutorials

8.3.1. Custom Action Type Tutorial

This tutorial shows step by step how to

- create a custom action type and multiple Java implementations to be used for different configurations ("Production", "Interactive Test" and "Test")
- write the Java code for the action type implementation
- prepare your workspace for contributing an Eclipse plug-in
- create a custom user interface for editing execution parameters of the custom action type
- create a custom user interface for editing functional initialization parameters of the custom action type
- create a custom user interface for editing technical initialization parameters of an implementation of the action type
- use custom icons for the custom action type
- export your custom UI plug-in for your end users

Beginning with section Creating the "Text Output" Action Type, we are creating the following example:

An action type "Text Output" is defined which has three different implementations: The "FileOutput" implementation writes to a file, the "ConsoleOutput" implementation writes either to stdout or stderr and the "NullOutput" implementation simply does nothing. Here is an overview of the parameters required for this action:

- functional init parameters (common to all implementations, specified for the action instance)
  - `tabWidth` (Integer) defines by how many space characters a tab character is replaced
- execution parameters (common to all implementations, specified for every Fire Action element)
  - `text` (String) is the output text
- technical init parameters (specific to a certain implementation)

  FileOutput
  - `filename` : String
  - `append` : Boolean

  ConsoleOutput
  - `stderr` : Boolean (true for stderr output, false for stdout output)

  NullOutput
  - no additional parameters

8.3.1.1. Creating the "Text Output" Action Type

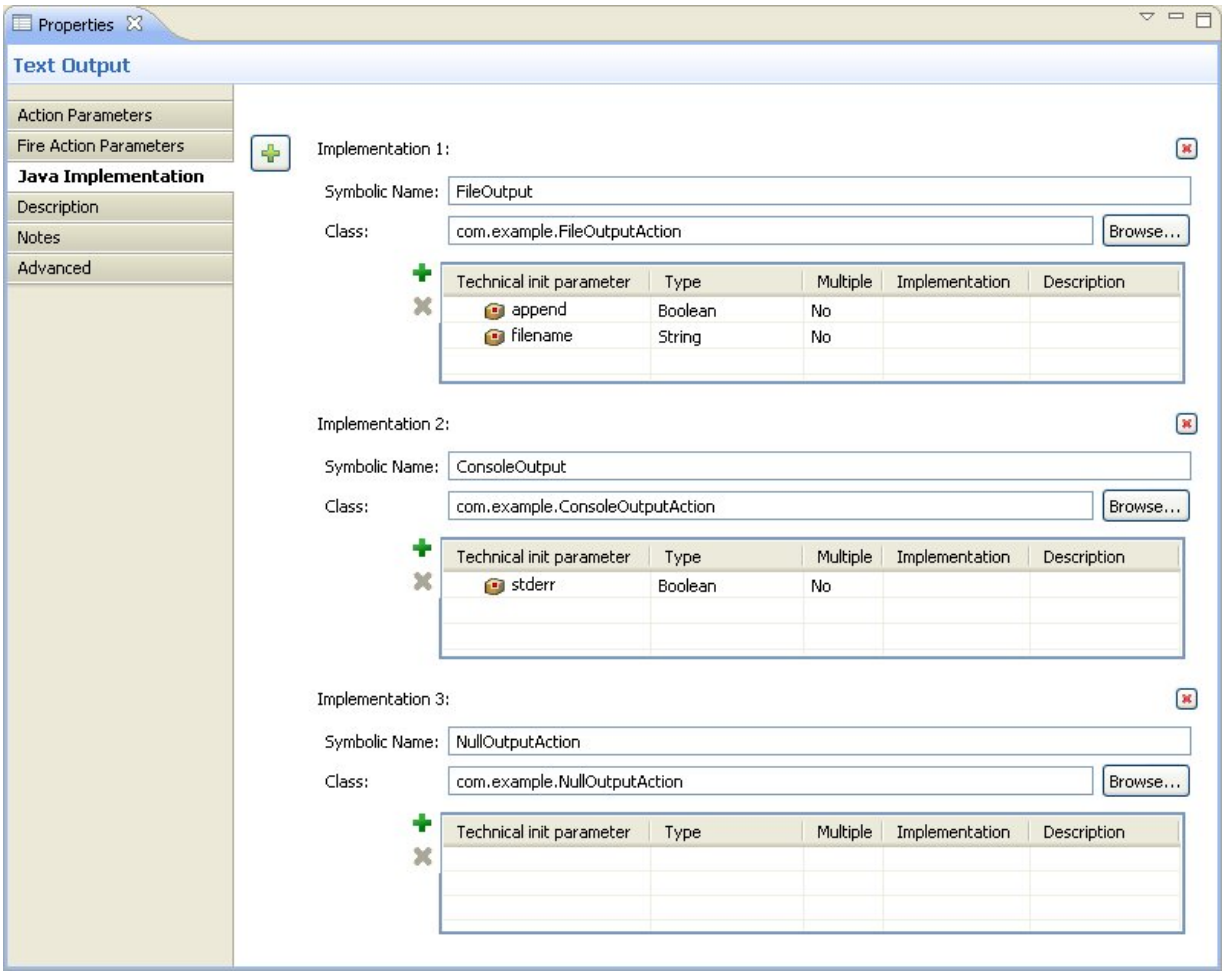
In this section we are creating the example sketched in the overview section. Perform the following steps:

1. Create a rule project "Custom Action Tutorial"
2. Create an action type "Text Output" in the rule model
3. In the Action Parameters tab for the new action type, create a parameter `tabWidth` of type Integer
4. In the Fire Action Parameters tab for the new action type, create a parameter `text` of type String
5. Open the Java Implementation tab

6. Create a new entry with the symbolic name `FileOutput` and the class `com.example.FileOutputAction`. Create the parameters `filename` of type String and `append` of type Boolean.

7. Create a new entry with the symbolic name `ConsoleOutput` and the class `com.example.ConsoleOutputAction`.Create the parameter `stderr` of type Boolean.

8. Create a new entry with the symbolic name `NullOutputAction` and the class `com.example.NullOutputAction`. Don't create any parameters here.



Now we create a rule using an action of the new type:

1. Create a flow rule `Text Output` in the rule model

2. Insert a Fire Action element into the rule

3. Edit the element to fire an action `output`. When asked, choose to create the action. It appears under the rule. You can add it manually as well.

4. Change the type of the action to `Text Output`

5. Specify the value `4` for the `tabWidth` parameter

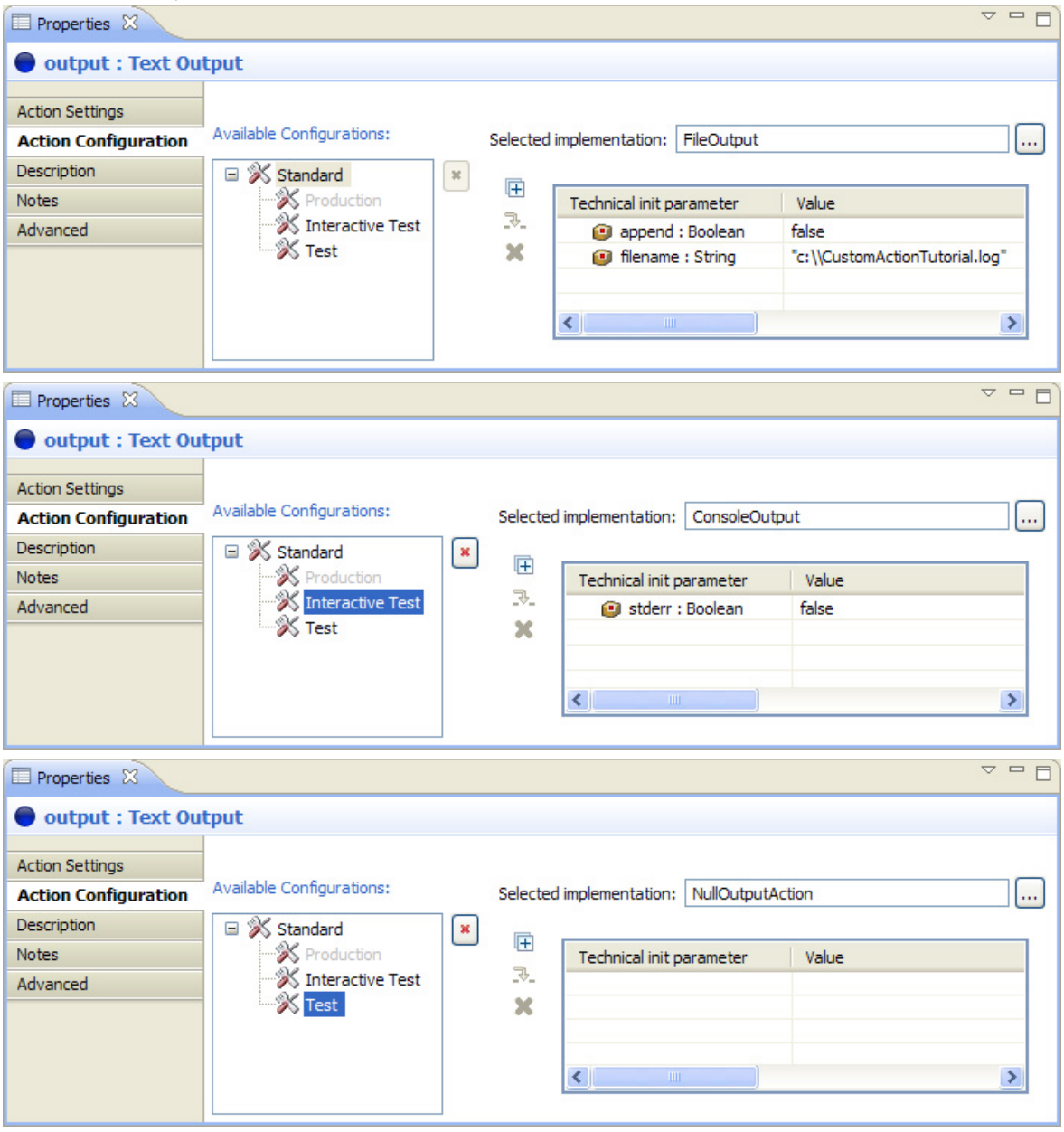
6. Open the Configurations tab of the rule model and create configurations with the names `Production`, `Interactive Test` and `Test`. Don't change their fallback configurations

7. Open the Action Configuration tab (Rule Integration perspective) for the `output` action.

   The tree on the left shows the three configurations created in the last step. We want the predefined configuration `Standard` to represent the `Production` configuration as well. Therefor, don't select an implementation for `Production`, but instead select the `FileOutput` implementation for `Standard`. Do this by pressing the ... button.

If during rule execution a request is started for the `Production` configuration, then the runtime recognizes that there is no special action implementation configured for `Production` and instead uses the implementation configured for its fallback.

For the `append` parameter, enter the value `false`. For the `filename` parameter, enter the value `"c:\\CustomAction.log"`. Now select the `Interactive Test` configuration and select the implementation `ConsoleOutput`. For the `stderr` parameter, enter the value false. Finally, select the `Test` configuration and select the implementation `NullOutput`.



8. Select the Fire Action element in the rule editor and specify the value `"Today is " & formatDate(currentDate())` for the `text` parameter

8.3.1.2. Implementing the "Text Output" Action

Depends on section Creating the "Text Output" Action Type.

We have to implement the action as Java class `com.example.FileOutputAction` and make it available for ACTICO Modeler. Perform the following steps:

1. Create a project "Custom Action Implementation". This may be a simple Java project, a plug-in project or a rule project.

2. Configure the build path of the project to contain the ACTICO Rules Runtime Library. If you created a rule project, this step has already been performed by the wizard.

3. Create the class `com.example.FileOutputAction` in the new project and insert the following code:

```
package com.example;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.util.Map;

import de.visualrules.runtime.IAction;
import de.visualrules.runtime.IContextAwareRequestData;
import de.visualrules.runtime.IRequest;
import de.visualrules.runtime.ISession;

public class FileOutputAction implements IAction {

   private PrintStream stream = null;
   private String tabReplacement = null;

   public void initFunctional(Map functionalInitParameter) {
      if (functionalInitParameter.get("tabWidth") instanceof Integer) {
         int tabWidth = ((Integer) functionalInitParameter.get("tabWidth")).intValue();
         char[] chars = new char[tabWidth];
         for (int i = 0; i < tabWidth; i++) {
            chars[i] = ' ';
         }
         tabReplacement = new String(chars);
      }
   }

   public void initTechnical(Map technicalInitParameter) {
      String filename = (String) technicalInitParameter.
         get("filename");
      boolean append = ((Boolean) technicalInitParameter.
         get("append")).booleanValue();
      try {
         stream = new PrintStream(new FileOutputStream(filename, append));
      }
      catch (FileNotFoundException e) {
         throw new RuntimeException("Unable to open file " + filename);
      }
   }

   public void execute(IContextAwareRequestData requestData, Map execParameterMap) {
      String text = (String) execParameterMap.get("text");
      if (tabReplacement != null)
         text = text.replace("\t", tabReplacement);
      stream.println(text);
   }

   public void leave(IRequest request) {
      if (stream != null)
         stream.close();
   }

   public void enter(IRequest request) {
      // noop
   }

   public void sessionClosed(ISession session) {
      // noop
   }
}
```

Note that the parameter values supplied to the action methods are read from maps, with the map key being the parameter name. It's certainly a good idea to use constants for them in the Java code.

4. Create classes for `ConsoleOutputAction` and `NullOutputAction` in a similar fashion. You might want to use inheritance or delegation for processing the common parameters.

If you now make a test run of the Text Output rule, then you will get an exception message in the console:

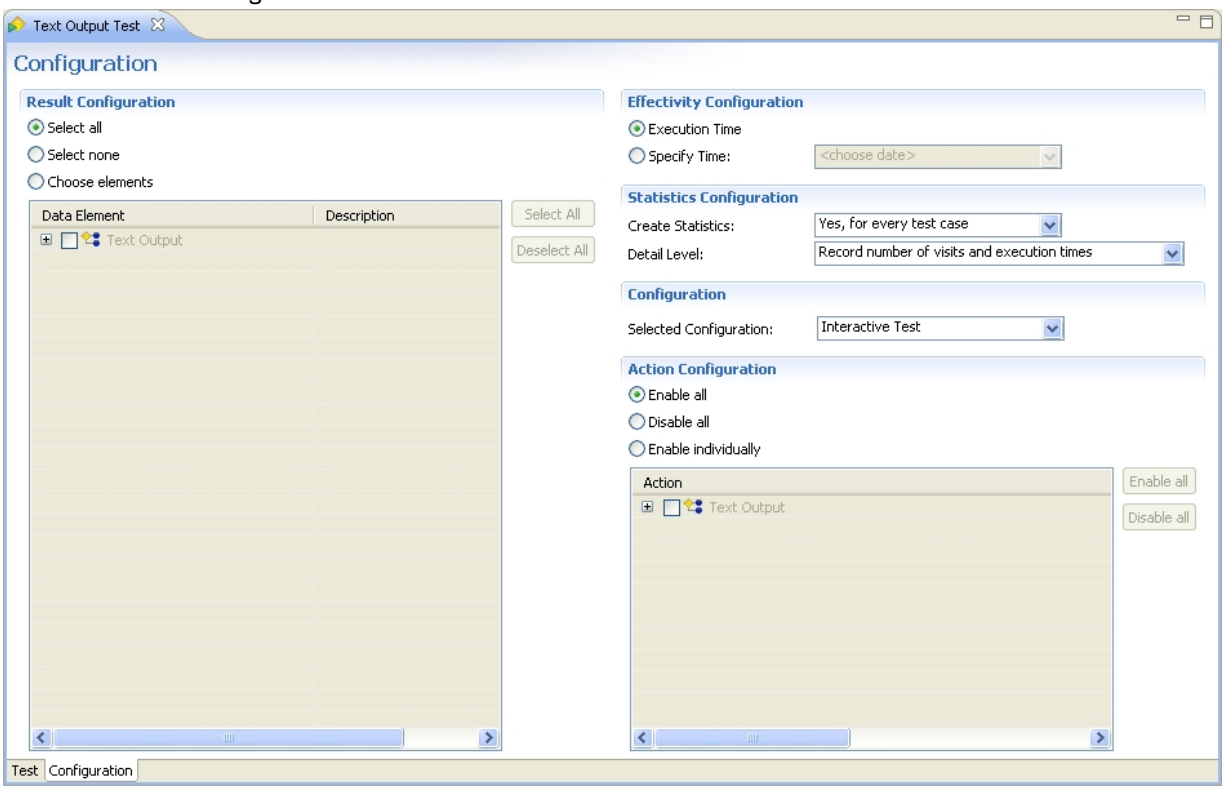
```
de.visualrules.runtime.lang.VRRuntimeException: Can't create action class
'com.example.FileOutputAction'
```

In order to make this class available to ACTICO Modeler, add the "Custom Action Implementation" project to the build path of the "Custom Action Tutorial" project. In your own projects, you may of course choose other methods to achieve this.

Rerunning the test, now the console shows:

```
2008-01-15 15:16:26,437 [main] INFO RuleRunner - Rule execution took 265ms.
2008-01-15 15:16:26,484 [main] INFO TestReport - Test report generation took 16ms.
```

Check the written file. It should contain the current date. The reason the output was not written to the console, is that the test runner uses the "Standard" configuration by default. We configured the `FileOutput` implementation to be used for the "Standard" configuration. In order to choose a different configuration during testing, open the Configuration tab of the test editor and select the desired value in the combo box labeled Selected Configuration.



Related Concepts.

- Action Life Cycle (`IAction`)

8.3.1.3. Preparing for Plug-In Contribution

Depends on section Creating the "Text Output" Action Type and Implementing the "Text Output" Action.

By default, a generic tabular UI component is shown for the user to provide parameter values required for initialization and execution of the `output` action. However, you might have a situation where you need a customized UI that is easier to use.

In contrast to the action implementation, a custom UI cannot be defined in the same Eclipse instance as the rule project that uses it. There are two possible solutions to this problem. For both solutions, an Eclipse Plug-in Project has to be created for the custom UI and will be exported as a jar file for the end users later on.

Solution 1:

The project is created in an arbitrary workspace, e.g. the workspace that holds the other projects as well. In order to test the developed functionality, first the plug-in is exported using the Eclipse export wizard. Then the Eclipse instance with the other two projects (may be the same instance) is closed and restarted after installation of the exported plug-in.

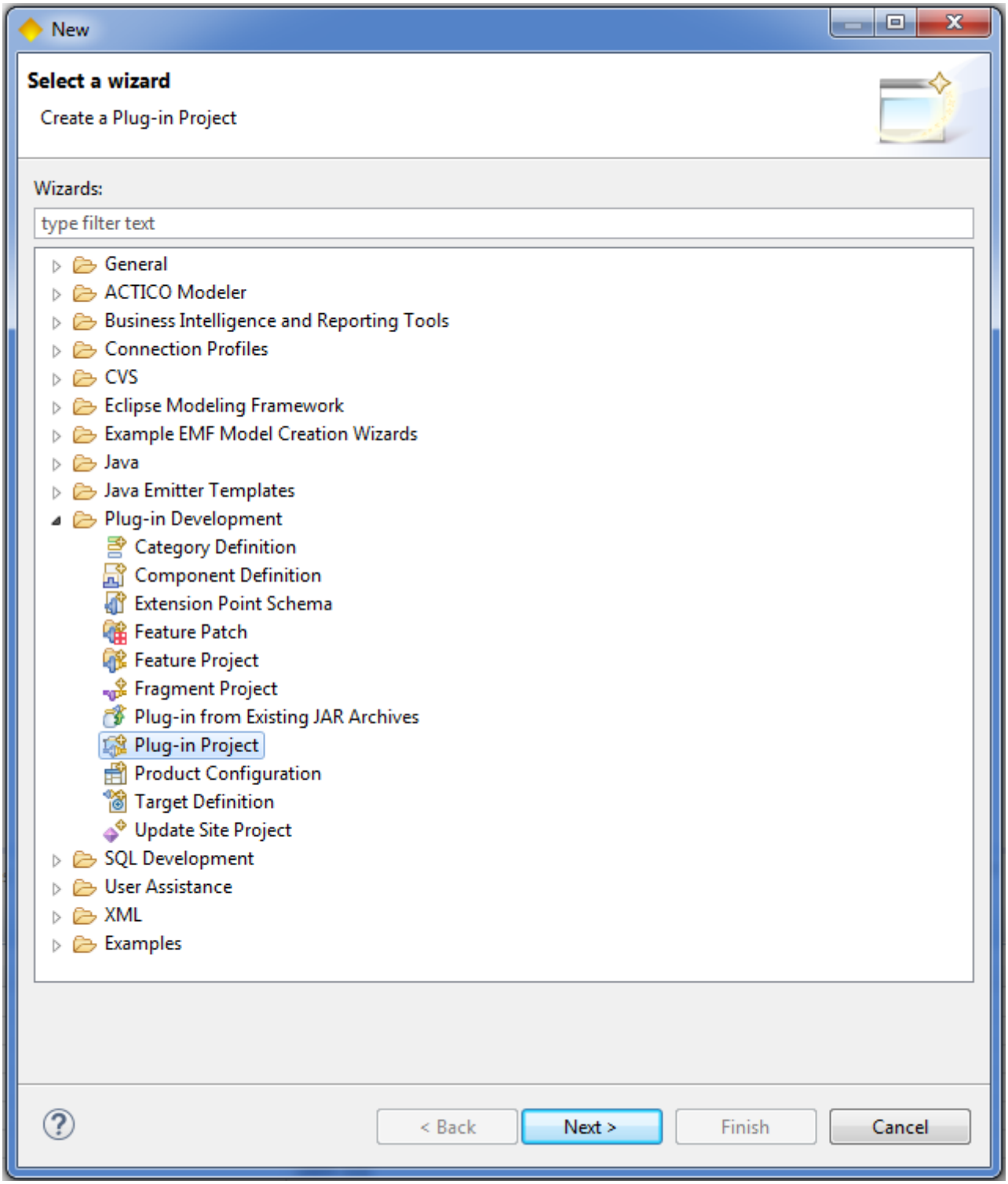
Solution 2

Two distinct workspaces are used, the "Development" workspace and the "Runtime" workspace. The "Development" workspace contains the "Custom Action UI" project. The running eclipse instance stays open during testing because it is used to start another Eclipse instance – the "Runtime" workbench that uses the "Runtime" workspace containing the other two projects.

Solution 1 is easier to understand but it has two severe drawbacks: Eclipse has to be shut down quite often and there is no possibility to debug the Plug-in being developed. Therefore, every Eclipse developer prefers solution 2.

Now perform the following steps. For steps specific to solution 2, there is a hint in the text:

1. Create an Eclipse plug-in project "Custom Action UI" using the New Project wizard with its default settings



2. The overview page of the [Plug-in Manifest Editor] is shown after the wizard has finished. You can re-open the editor later on by double- clicking the file `META-INF/MANIFEST.MF`

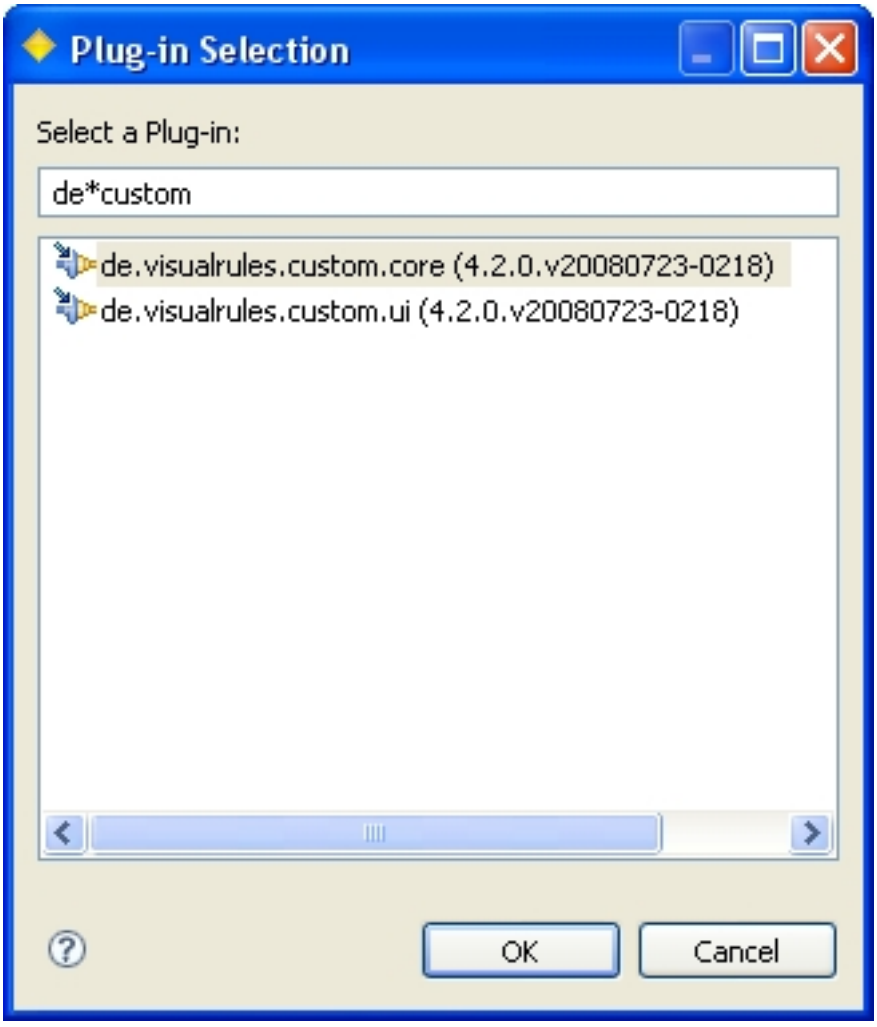
   Specific to solution 2:

   - Click the green button labeled [Launch an Eclipse application]. This creates and executes a new Launch configuration. You can edit the launch configuration in the Eclipse launch dialog. A lot of information is displayed in this dialog, including the location of the Runtime Workspace.

- We are now going to move the two other projects to the workspace of the Runtime Workbench that has started in the meantime. In a usual configuration, the first Runtime Workspace folder is a sibling of the Development Workspace folder, named `runtime-EclipseApplication`.
- Import the projects `Custom Action Tutorial` and `Custom Action Implementation` into the Runtime Workspace, but be sure to check the button Copy projects into workspace. Now you should shut down the [Runtime Workbench] and delete the two imported projects from the Development Workspace.

  Note: It's also possible to let the project implementing the action stay in the Development Workspace. However, doing this makes it necessary to create a jar file containing the action implementations and adding this jar file to the build path of the project containing the rule model. You may wonder why we didn't start the tutorial with the two workspaces separated. The reason is that for a user reading only the previous sections, this would be unnecessary work to do.

3. Open the Dependencies tab of the Plug-in Manifest Editor and add the following plug-in: `de.visualrules.custom.ui`



Now you can continue with any of the sections Adding a custom User Interface for the Execution Parameter, Adding a custom User Interface for the (Functional) Initialization Parameters, Adding a custom User Interface for the technical Initialization Parameters or Adding custom Icons.

8.3.1.4. Adding a custom User Interface for the Execution Parameter

Depends on sections Creating the "Text Output" Action Type to Preparing for Plug-In Contribution.

We are now going to add a custom user interface for the `text` execution parameter that specifies the string that is written to the configured destination.

To do this, perform the following steps:

1. Open the Extensions tab of the Plug-in Manifest Editor for the `Custom Action UI` project

2. Create an extension of `de.visualrules.custom.ui.customUI` and create the element `customActionUI` using the context menu. Enter `Text Output` (Name of Action Type) in the type field and `custom_action_ui.TextOutputExecUI` in the execUI textbox.

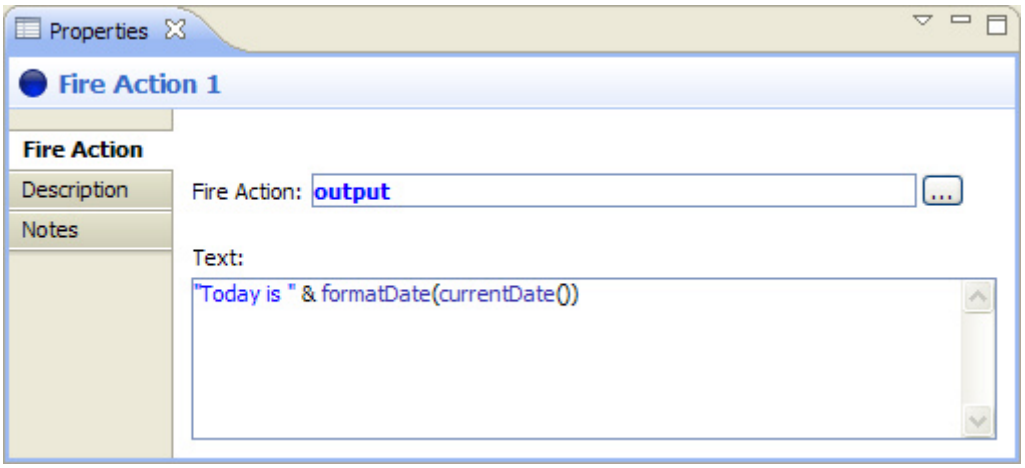
3. Create the class `custom_action_ui.TextOutputExecUI` by clicking the execUI hyperlink. Insert the following code:

```
package custom_action_ui;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.ui.forms.widgets.FormToolkit;

import de.visualrules.custom.core.IExternalParameterAccessor;
import de.visualrules.custom.core.IParameterTransactionManager;
import de.visualrules.custom.ui.ICustomUI;
import de.visualrules.custom.ui.IWidgetFactory;

public class TextOutputExecUI implements ICustomUI {

  public Composite createUI(Composite parent, FormToolkit toolKit, IWidgetFactory factory)
 {
    Composite comp = toolKit.createComposite(parent);
    GridLayout layout = new GridLayout();
    comp.setLayout(layout);
    toolKit.createLabel(comp, "Text:");
    Control control = factory.createExpressionViewer("text", comp,
        SWT.MULTI | SWT.V_SCROLL);
    control.setLayoutData(new GridData(GridData.FILL_BOTH));
    return comp;
  }

  public void refresh() {
    // not needed in this example
  }

  public void setFocus() {
    // noop
  }

  public void setParameterAccessor(IExternalParameterAccessor accessor) {
    // noop
  }

  public void setTransactionManager(IParameterTransactionManager manager) {
    // noop
  }
}
```

   The method `IWidgetFactory.createExpressionViewer()` creates a text box control that is bound to the value of the parameter with the specified name.

   The methods `refresh()`, `setParameterAccessor()` and `setTransactionManager()` are not needed as long as all widgets for parameter manipulation are created with the supplied `IWidgetFactory`. For a more advanced example, see Adding a custom User Interface for the (Functional) Initialization Parameters.

4. Start the Runtime Workbench again and open the `Text Output` rule. Click the Fire Action element and check its properties. You see a simple control for editing the output text:



For exporting your plug-in, see section Exporting the Plug-in.

8.3.1.5. Adding a custom User Interface for the (Functional) Initialization Parameters

Depends on sections Creating the "Text Output" Action Type to Preparing for Plug-In Contribution.

We are now going to add a custom user interface for the `tabWidth` initialization parameter. The user will have the following options of tab widths: 1, 2, 4 and 8.

To do this, perform the following steps:

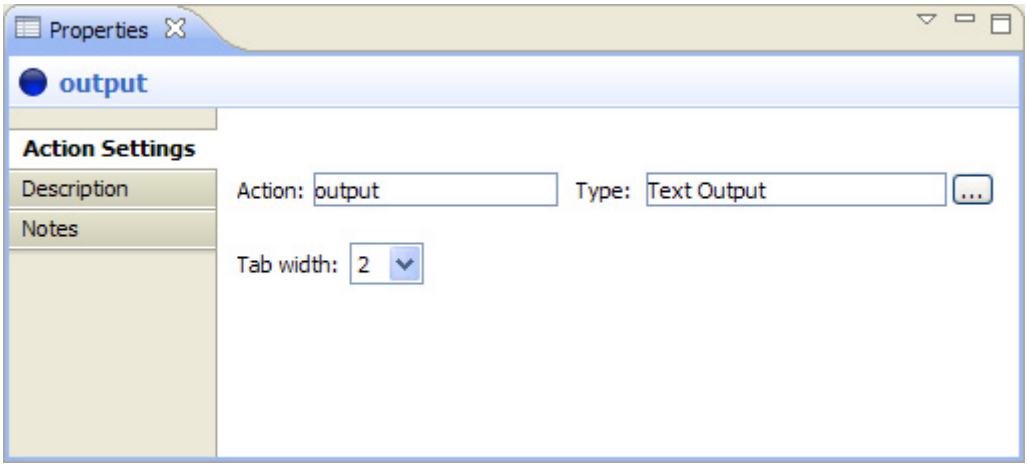
1. Open the Extensions tab of the Plug-in Manifest Editor for the `Custom Action UI` project. If you have not performed section Adding a custom User Interface for the Execution Parameter, create an extension of `de.visualrules.custom.ui.customUI` and add the element `customActionUI`. Enter `Text Output` in the type textbox and `custom_action_ui.TextOutputInitUI` in the initUI textbox.

2. Create the `class custom_action_ui.TextOutputInitUI` by clicking the label of the initUI textbox. Insert the following code:

```
package custom_action_ui;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Combo;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.forms.widgets.FormToolkit;

import de.visualrules.custom.core.IExternalParameterAccessor;
import de.visualrules.custom.core.IParameterTransactionManager;
import de.visualrules.custom.ui.ICustomUI;
import de.visualrules.custom.ui.IWidgetFactory;

public class TextOutputInitUI implements ICustomUI {
  private static final String TAB_WIDTH = "tabWidth";
  private Combo combo;
  private IExternalParameterAccessor parameterAccessor;

  public Composite createUI(Composite parent, FormToolkit toolkit,
      IWidgetFactory factory) {
    Composite comp = toolkit.createComposite(parent);
    comp.setLayout(new GridLayout(2, false));
    toolkit.createLabel(comp, "Tab width:");
    combo = new Combo(comp, SWT.READ_ONLY);
    combo.setItems(new String[] { "1", "2", "4", "8" });
    combo.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {
        parameterAccessor.setExpression(TAB_WIDTH, combo.getText());
      }
    });
    return comp;
  }

  public void setParameterAccessor(IExternalParameterAccessor accessor) {
    parameterAccessor = accessor;
  }

  public void refresh() {
    int tabWidth = parameterAccessor.getIntegerConstant(TAB_WIDTH);
    if (tabWidth > 0) {
      combo.select((int) (Math.log(tabWidth) / Math.log(2)));
    } else {
      combo.setItems(new String[] { "1", "2", "4", "8" });
    }
  }

  public void setFocus() {
    combo.setFocus();
  }

  public void setTransactionManager(IParameterTransactionManager manager) {
    // noop
  }
}
```

The method `IExternalParameterAccessor.getIntegerConstant(String)` converts an expression that is known to contain a simple integer literal like `3` to a Java int.

3. Start the Runtime Workbench again and open the `Text Output` rule. Select the action and check its properties. You see a simple control for editing the tab width:



For exporting your plug-in, see section Exporting the Plug-in.

8.3.1.6. Adding a custom User Interface for the technical Initialization Parameters

Depends on sections Creating the "Text Output" Action Type to Preparing for Plug-In Contribution.

We are now going to add a custom user interface for editing the `append` and `filename` parameters of the `FileOutputAction` implementation.

To do this, perform the following steps:

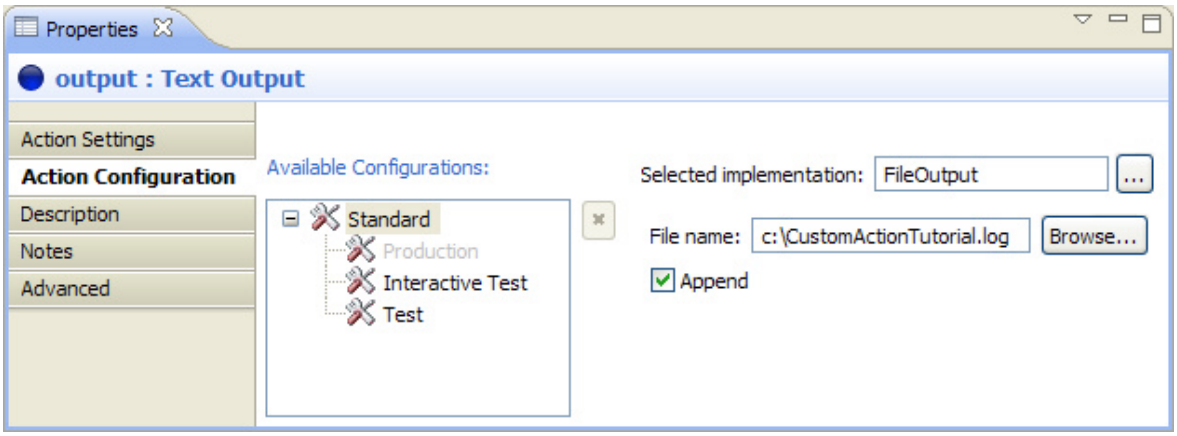
1. Open the Extensions tab of the Plug-in Manifest Editor for the `Custom Action UI` project. If you have not performed section Adding a custom User Interface for the Execution Parameter or Adding a custom User Interface for the (Functional) Initialization Parameters, create an extension of `de.visualrules.custom.ui.customUI` and add the element `customActionUI`. Because the technical init parameters are specific to an implementation, you have to create the element `technicalActionUI` below. Enter `FileOutput` in the `implementationName` textbox and `custom_action_ui.FileOutputTechnicalInitUI` in the initUI textbox.

2. Create the class `custom_action_ui.FileOutputTechnicalInitUI` by clicking the label of the initUI textbox. Insert the following code:


```
package custom_action_ui;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.FileDialog;
import org.eclipse.ui.forms.widgets.FormToolkit;

import de.visualrules.custom.core.IExternalParameterAccessor;
import de.visualrules.custom.core.IParameterTransactionManager;
import de.visualrules.custom.ui.ICustomUI;
import de.visualrules.custom.ui.IWidgetFactory;

public class FileOutputTechnicalInitUI implements ICustomUI {

  private static final int COLUMNS = 3;
  private Composite comp;
  private IExternalParameterAccessor parameterAccessor;

  public Composite createUI(Composite parent, FormToolkit toolkit,
      IWidgetFactory factory) {
    comp = toolkit.createComposite(parent);
    comp.setLayout(new GridLayout(COLUMNS, false));
    toolkit.createLabel(comp, "File name:");
    factory.applyHorizontalSpan(factory.createTextbox("filename", comp,
        SWT.SINGLE), COLUMNS - 2, true, false);
    toolkit.createButton(comp, "Browse...", SWT.PUSH).addSelectionListener(
        new SelectionAdapter() {
          public void widgetSelected(SelectionEvent e) {
            handleBrowseButtonPressed();
          }
        });
    factory.applyHorizontalSpan(factory
        .createCheckBox("append", comp, "Append"), COLUMNS);
    return comp;
  }

  private void handleBrowseButtonPressed() {
    FileDialog dialog = new FileDialog(comp.getShell());
    String file = dialog.open();
    if (file != null) {
      parameterAccessor.setStringConstant("filename", file);
    }
  }

  public void setParameterAccessor(IExternalParameterAccessor accessor) {
    parameterAccessor = accessor;
  }

  public void refresh() {
    // not needed in this example
  }

  public void setFocus() {
    // noop
  }

  public void setTransactionManager(IParameterTransactionManager manager) {
    // noop
  }
}
```

By using `IExternalParameterAccessor.createTextbox()` instead of `IExternalParameterAccessor.createExpressionViewer()`, we hide the fact that every parameter is internally an expression from the user.

3. Start the Runtime Workbench again and select the output action. Select the Action Configuration tab from its properties. You see a simple control for editing the parameter values:



For exporting your plug-in, see section Exporting the Plug-in.

8.3.1.7. Adding custom Icons

Depends on sections Creating the "Text Output" Action Type to Preparing for Plug-In Contribution.

In order to use custom icons for the `Text Output` action, perform the following steps:

1. Create a folder `icons` in the `Custom Action UI` project
2. Put an icon `large.png` with a size of 100x72 pixels into the `icon` folder
3. Put an icon `large_disabled.png` with a size of 100x72 pixels into the `icon` folder
4. Put an icon `small.gif` with a size of 16x16 pixels into the `icon` folder
5. Open the file `plugin.xml` either by double clicking it or by selecting the appropriate tab of the Plug-in Manifest editor. Paste the following text before the closing tag `</plugin>`:

```
<extension point="de.visualrules.custom.ui.iconFactory">
   <static type="Text Output"
           largeDisabledIconPath="icons/large_disabled.png"
           largeEnabledIconPath="icons/large.png"
           smallIconPath="icons/small.gif">
   </static>
</extension>
```

   Of course you can enter the data in the same fashion as described in the preceding sections, using the context menu in the Extensions tab of the [Plug-in Manifest editor].

6. Start the Runtime Workbench again to see your icons in action.

For exporting your plug-in, see section Exporting the Plug-in.

8.3.1.8. Exporting the Plug-in

Depends on Preparing for Plug-In Contribution.

You may want to create a zip file from the `Custom Action UI` plug-in so that end users can install it.

To do this, perform the following steps:

1. Start the Eclipse Export Wizard from within the Development Workbench
2. From the category Plug-in Development, choose the wizard [Deployable plug-ins and fragments]
3. Make sure that the plug-in `Custom Action UI` is selected and export it to a file or a folder

8.3.1.9. Troubleshooting your custom UI

If you have contributed a custom UI, it may happen that it is not shown after you have started ACTICO Modeler and selected the action or a Fire Action element. In such a situation, it is recommended to use the tracing functionality of eclipse. We assume that you are using solution 2 from section Preparing for Plug-In Contribution.

1. Open the Run or Debug dialog and select the Tracing tab.
2. Activate the check box Enable tracing for the selected plug-ins and select all plug-ins, just to be sure.

When you run or debug ACTICO Modeler, you will see information written to the console. You will see what classes are found for your Action Type. You will also see a message if a class cannot be loaded or instantiated. In the latter case, it's very likely that your class either does not have a default constructor (public, without parameters) or that an exception occurred during instantiation of the class.

Chapter 9. Integrating with the ACTICO Modeler

ACTICO Modeler offers a couple of APIs and extension points that allow it to be integrated with other Eclipse based tools or applications. This way it is possible to include the graphical rule modeling and code generation capabilities into other tools or applications.

Specifically, with these APIs it is possible for an Eclipse application to

- create rule projects, rule models, rule packages and rules
- define the input and output parameters of newly created rules
- open the rule editor
- create, remove and synchronize a data model with some other data model
- configure and trigger the code generation
- control the model elements available for the user
- apply some customization to the user interface
- introduce a custom permission scheme for rule models

In order for a plug-in to use the APIs it must import the plug-ins `de.visualrules.integration`, `de.visualrules.integration.model` and `de.visualrules.ui.integration`.

Use class `VisualRulesIntegration` to access the different APIs. The best way to learn about the API is to look at the Javadoc in `IRuleIntegration`, `IDataModelIntegration` and `IJavaIntegration`.

If you are interested in integrating ACTICO Modeler and or rule code generation and execution into your Eclipse based application or server-based solution, please contact us for further information.

Visit http://www.actico.com or send an email to support@actico.com.

Chapter 10. Built-in Service- and Resources Types

10.1. Service Types

Rule models have the concept of a service which is usually used to call external systems or existing functionality. For this, ACTICO Modeler can be extended by defining custom service types or using one of the following built-in service types.

10.1.1. Standard Service Type

The service type Standard does not perform a operation. It is a so-called "No-op" implementation and thus useful for testing or for being exchanged at runtime. This is similar to the *action type Standard* described in the *Modeling Guide*.

10.1.2. Call HTTP Service Type

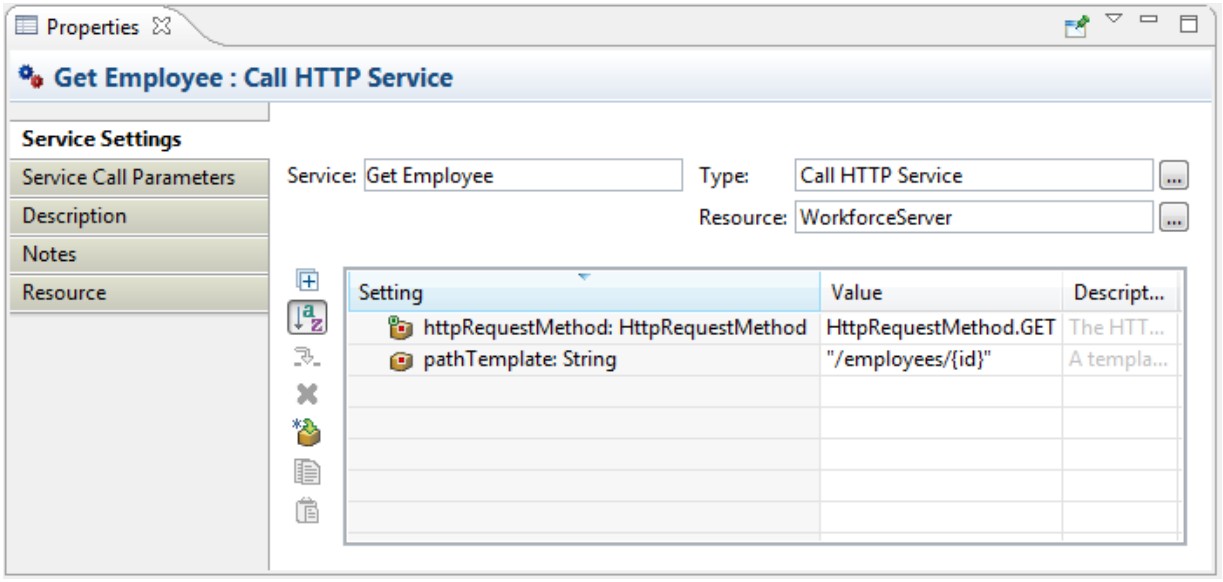
The service type Call HTTP Service is used to interact with web services of a web server. For this, services of this type need connection information to the web server which is stored in a resource of type HTTP Resource. In order to address a web service, a service of type Call HTTP Service needs to configure the HTTP method and a path.

The service setting for the HTTP method is called `httpRequestMethod` and accepts values from the `HttpRequestMethod` enumeration. The path is specified with a service setting called `pathTemplate` which is basically appended to the *base URL* provided by the resource. Together this results in a URL following the format: `scheme://hostname:port/path/?query_string`. It is possible to use placeholders in the path and query string, which is explained in the following section.

For debugging the HTTP service call set the program argument `-l debug` in the run configuration.

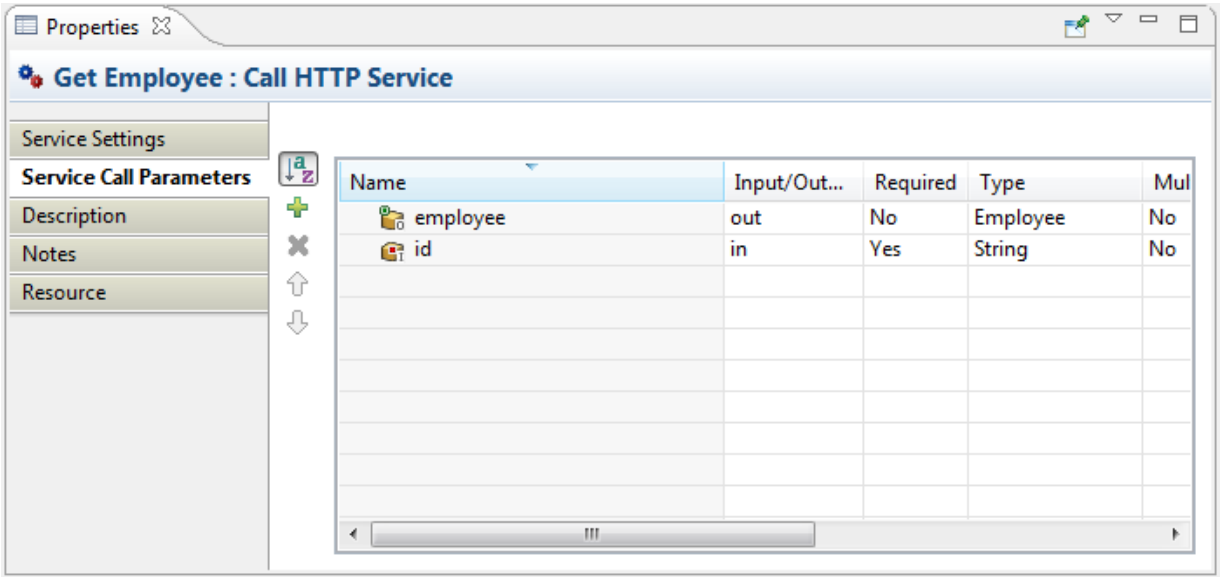
10.1.2.1. Configuring Placeholders in Path

It is often necessary to dynamically change parts of the path. For this, it is possible to define placeholders in the value for the `pathTemplate`. A placeholder is written as `{name}`. Hereby `name` is the name of a *Service Call Parameter* and at runtime the placeholder will be replaced with the value of the parameter. The following example illustrates how this is done:



In this example the path addresses a web service with an id like `"/employees/42"`. In order to allow a dynamic replacement, the value is replaced by a placeholder with a meaningful name: `{id}`

As next step, an according *Service Call Parameter* must be defined:



Note that Service Call Parameters that are used in a placeholder must be *Input* parameters and it is highly recommended to mark them as *required*. The service can have additional Service Call Parameters which are then used to send or receive data. In the above example, the `employee` Service Call Parameter is the result of the unmarshalled response. More information on this is in Sending and Receiving Data.

Example 10.1. Path for Call HTTP Service

It is also possible to specify placeholders to formulate query strings in the path. For example if the path contains query strings like `"/employees/?minAge=23&maxAge=42"`, it is possible to use placeholders like this: `/employees/?minAge={minAge}&maxAge={maxAge}`

Similar to placeholders in the path, these will be replaced with Service Call Parameters of the same name.

The fields and values of query strings are often optional. Despite that, the path should contain a full query string with placeholders. The according Service Call Parameters should then be marked as *not required*. When the value of the Service Call Parameter is `NULL`, the field and its value will be omitted from the request. If the value is empty, the field without value will be contained. For example when the empty string is provided for the value of *myvalue*, this would result in: `"../?myvalue="`

When creating a Service Call Parameter for a placeholder, consider following points:

- Only a *Basic Data Type* (except *Any*) or a *Type Alias* thereof can be meaningful used as type
- Needs to be an Input parameter
- It is recommended to set *required* when used within the path and *not required* when it is used in the query string and is optional

It may be advantageous for users to work with *Structures* or data types like *Date* in a rule. However it can be rather tricky to map this to a placeholder and thus it is difficult to provide the service as is. In this case it is recommended to provide the users a rule instead of the service. The rule can then do the actual service call and the mapping can be modelled, for instance by formatting a *Date* to the correct format as a *String*. As a rule is reusable, this allows users to work with domain objects and it is less error prone, as the mapping is provided.

10.1.2.2. Setting HTTP Header Fields

The service setting `httpRequestHeaders` allows the specification of HTTP headers for a specific HTTP service call. The name and value of header fields, as defined by Hypertext Transfer Protocol -- HTTP/1.1, can be provided as key/value pair in this optional map.

The HTTP Resource is the recommended location to define the HTTP headers of a connection. It can be shared by multiple HTTP service calls. The specific headers of the HTTP service override those of the HTTP Resource and the runtime configuration.

Related References.

- HTTP Resource Type

10.1.2.3. Sending and Receiving Data

The called, external web service may expect or return data as content of request or response. When using the *Call HTTP Service* type, *Service Call Parameters* represent the data. *Input* parameters are used when data is to be sent with the request and *Output* parameters if data is expected as response. For instance, the `employee` Service Call Parameter in the previous example is a result of the external web service.

The request can be directly controlled by the HTTP headers `Content-Type` and `Content-Encoding`, whereas the use of the HTTP headers `Accept`, `Accept-Charset` and `Accept-Encoding` may possibly influence the response from the web service:

- If data is sent the HTTP header `Content-Type` is mandatory. The *Input* parameters are marshalled to the request according to the media type specified in this header. This header may also include information about the *charset* of the request.
- The response is unmarshalled and mapped to the *Output* parameters according to the `Content-Type` of the response from the web service. The response can be indirectly influenced by the HTTP headers `Accept*`.
- The following media (sub)types are supported for marshalling/unmarshalling during the processing of the HTTP request/response:
  - `XML` - This media subtype can be used for marshalling any number of input parameters to the XML request or for unmarshalling the XML response to any number of output parameters.
  - `JSON` - This media subtype can be used for marshalling any number of input parameters to the JSON request or for unmarshalling the JSON response to any number of output parameters.
  - `TEXT` - This media type can be used to send exactly one input parameter of type `String` as request or to receive response in exactly one output parameter of type `String`.

For a service call of the service type 'Call HTTP Service' imported XML data types can be used to define the type of the input/output parameters. You can do so on the tab XML Representation in the properties view for the relevant parameter.

Related References.

- Service Call Parameter

10.1.2.4. Configuring Marshalling/Unmarshalling Options

Marshalling and Unmarshalling take place at the call of a HTTP service where instances of parameters have to be converted into XML or JSON representations and vice versa. The service setting `marshallingOptions` can be used to configure these conversions.

The following options are available and can be specified as key/value pairs in the optional map `marshallingOptions`:

Option	Description
`vr.bind.json.skipSingleParameterName`	If you use `JSON` as `Content-Type` and the service has only *one* parameter then it's possible to skip the parameter name. Then it won't be inserted into the JSON representation of the request by the marshaller. Additionally the unmarshaller won't expect the parameter name during processing the JSON representation of the response. If you want a different behavior of the marshaller or the unmarshaller concerning this property you can use the options `vr.bind.marshaller.json.skipSingleParameterName` or `vr.bind.unmarshaller.json.skipSingleParameterName` which will override the setting with the more general option. Possible values: `true`, `false` (default: `true`)
`vr.bind.marshaller.json.nullRepresentation`	If you use `JSON` as `Content-Type` you can specify how parameter fields with `null` value will be treated. Possible values: • `skip` - Fields with value `null` will be skipped and aren't part of the JSON representation. • `null` - Fields with value `null` are part of the JSON representation. (default: `null`)
`vr.bind.map.representation`	If you use `JSON` as `Content-Type` you can specify how a map should be representated. Possible values: • `array` - Map will be produced and consumed as array. `[{ "key": "x", "value": 1 },{ "key": "y", "value": true }]` • `object` - Map will be produced and consumed as object.

Option	Description

	<pre>{
	  "x": 1,
	  "y": true
	}</pre>
	(default: `array`)

`vr.bind.xml.envelopeQName`	If you use `XML` as `Content-Type` and your HTTP Service uses *multiple* parameters then you have to provide the name of the enveloping XML root element in the form of a qualified name (QName). It will be inserted into the XML representation of the request by the marshaller. Additionally the unmarshaller will expect the same XML root element during processing the XML representation of the response. If you want to specify a different XML root element for the request (marshalling) and for the response (unmarshalling) you can use the options `vr.bind.marshaller.xml.envelopeQName` or `vr.bind.unmarshaller.xml.envelopeQName` which will override the setting with the more general option.
`vr.bind.marshaller.xml.nullRepresentation`	If you use `XML` as `Content-Type` you can specify how elements/attributes with `null` values will be treated. Possible values: • `skip` - Elements/attributes with `null` values will be skipped and aren't part of the XML representation. • `empty` - Writes an empty element tag or an empty string as attribute value. • `nil` - Writes `xsi:nil=true` for elements and `null` as attribute value. Default: It depends on the specification of the elements/attributes within the assigned XML schema definition (XSD) which representation for `null` is used.
`vr.bind.marshaller.xml.omitXMLDeclaration`	If you use `XML` as `Content-Type` you can specify that the XML declaration is omitted in the request. Possible values: `true`, `false` (default: false)
`vr.bind.marshaller.xml.namespace.prefixes`	Several namespaces might be involved when marshalling `XML`. Therefore each namespace is usually assigned to a namespace prefix which is build upon a default pattern. This option allows to specify custom prefixes to be used for a namespace. The value of this option has to be given as `xmlns:prefix=namespaceUri` and a comma can be used to separate more than one binding. For example a value of this option could look like:

Option	Description
	`xmlns:ns1=http://www.visual-rules.com/namespace1/, xmlns=http://www.visual-rules.com/defaultNamespace/`

Related References.

- Chapter 11, *Marshalling and Unmarshalling*

10.1.3. Java Integration Service Type

The service type Java Integration is used to integrate Java functionality. An import is provided to create services of this type. Refer to Integration into the Java Service Landscape for more information.

10.2. Resource Types

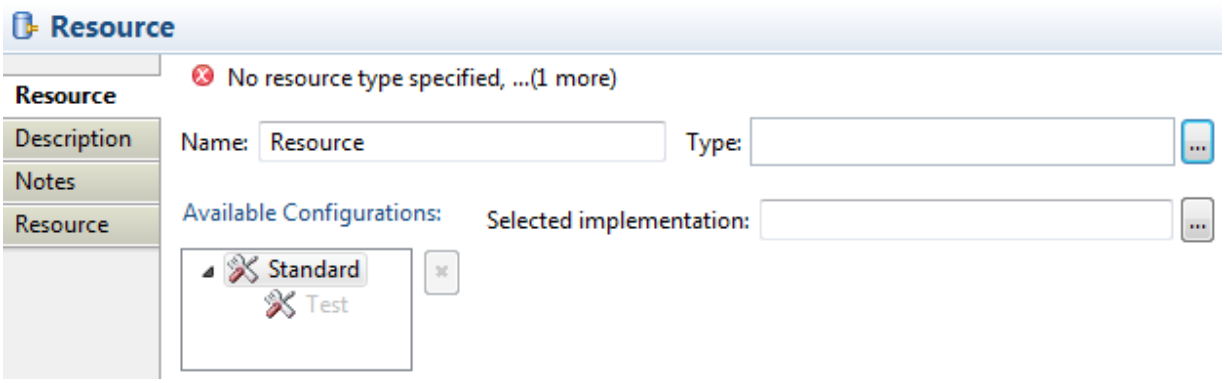
A *resource type* is an abstraction for a configuration needed by external, technical systems. For instance, a resource type can represent the connection information for a web server or for a database as described in the *Database Integrator User Guide*.

In order to provide values for a configuration, a `resource` needs to be created. Values of resource types have in common that they often need to be changed at runtime. For this, configuration values of a resource can either be switched by a Configuration or be override at runtime.

10.2.1. Creating and Configuring a Resource

A *resource* can be created from the context menu of a *Rule Model*. To do so, right-click on the model and select New Element -> Resource. A wizard dialog appears asking for a name. After finishing the dialog, the newly created resource is found in External Data underneath the *Rule Model* in the *Project Explorer* or *Rule Explorer*.

After creation, a resource needs to be further configured. Select the resource and the settings are shown in the Properties View:



The options, which are common for every resource, are:

Option	Description
Name	The name of the resource.
Type	Shows the current resource type of the resource. Click on ... to browse the available types.
Available Configurations	Lists all the configurations defined on the rule model (this is represented as a tree to visualize the fallback hierarchy). Select a configuration to see and change the values per configuration.
Selected Implementation	If the resource type has multiple implementation, this is used to select a specific implementation. Click on ... to browse available implementations.

Depending on the selected resource type, various further options may become available.

10.2.2. HTTP Resource Type

The resource type HTTP Resource is used to configure a connection to a web server and is used by services of type Call HTTP Service. When this resource type is selected, the following options become available:

Option	Description
Base URL	The URL used to address the web server and a path available via HTTP. The entry needs to be a valid URL and follows the format: `scheme://hostname:port/path` For instance, `http://localhost:8088/workforce/restapi` would be a valid entry.
Authentication Method	Specifies which kind of authentication method is used by the server. The supported methods are: • No Authentication • Basic Authentication - The HTTP authentication method. When this is selected, a user name and password can be entered • Identity Management Context-ID Forwarding - When this is selected, the authentication is done based on the *Context-ID* of an existing and logged-in user from *Identity Management*. This replaces Basic Authentication as method and can be used with server applications supporting *Identity Management*. In order for this to work, the execution of a service using this resource must be performed in the context of a logged-in user.  When selecting Identity Management Context-ID Forwarding, rule tests using services which use the resource will not work.
User Name	The user name used when Basic Authentication is selected as Authentication Method.
Password	The password used when Basic Authentication is selected as Authentication Method.  The password is stored in the rule model. Consider using passwords for testing purposes only when using Basic Authentication and override at runtime.
HTTP Header Fields	Allows setting HTTP Header as specified by Hypertext Transfer Protocol -- HTTP/1.1.
Maximum Response Content Size	Specifies the maximum content size of the HTTP response in megabytes (default: 32 MB). A negative number corresponds to "unlimited".

The mentioned options can be changed at runtime. Refer to Override Configuration of HTTP Resource Type for details.

Additionally, the HTTP Header Fields can be overridden by related settings at the service type Call HTTP Service (see Setting HTTP Header Fields for details).

10.2.3. Database Connection Resource Type

The resource type Database Connection is provided by the *Database Integrator* extension and is described in the *Database Integrator User Guide*.

10.3. Override Resource Configuration at Runtime

The most configuration values on a resource work well in all runtime environments and need not be changed. However there are scenarios where it is necessary to change some of them. For example changing the URL of a web server or replacing a password. For this it is possible to override the resource configuration at runtime without the need to regenerate rule code. There are two ways to do this: Override via Rule Execution API or by using a properties file.

The override via Rule Execution API has higher precedence than the properties file.

Both approaches have the usage of key-value pairs in common, which have the same meaning as the ones used by the resource type. Regardless which approach is chosen, it suffices to specify only those values that are to be overwritten. For example, if only the *password* is set, all other settings are used as defined on the resource.

Keys and values are case-sensitive.

The override will affect the setting in every configuration. It is also possible to specify a override specifically for one configuration. This is simply done by prepending the key with the configuration name it applies to. For instance, if the *connectionUrl* is different for the configuration *test*, the key *test.connectionUrl* should be used. Be aware that key-values, that are not specific to a configuration, will still be overwriten even if running with a specific configuration. If in the above example the key-value pair for *password* is set, then this would still apply to the configuration *test*.

10.3.1. Override via Rule Execution API

The Rule Execution API provides the possibility to register values in order to replace values for a resource configuration. This is done by using the `#registerResourceOverrides(..)` method on ISession. When using this to override values, it is necessary to specify for which rule model and resource this is effective, as a `ISession` can be reused for multiple rule executions involving multiple rule models.

10.3.2. Override using Properties File

A properties file can be used to replace values for a resource configuration. The name of the file must match the resource name, for instance `MyResource.properties` if the resource is named `MyResource`. The file must be loadable from the class path by the generated code. For testing purposes, this can be achieved by putting it into the source folder of a rule project.

When using a properties file, special characters need to be quoted. For example a backslash must be quoted by a second backslash. See JavaDoc on Properties for details.

10.3.3. Override Configuration of HTTP Resource Type

The following keys and values are used to override the resource configuration for the HTTP resource type:

Key	Allowed Values	Matching UI Option
`baseUrl`	A valid URL	Base URL
`httpHeader.<header-name>`	Comma separated list of values	An entry from HTTP Header Fields, where `<header name>` is a concrete header. For instance, this would be "httpHeader.Content-Type" when specifying the content type.

Key	Allowed Values	Matching UI Option
`authenticationMethod`	A string with one of the following values: • `None` • `BasicAuthentication` • `IMIdentityContextForwarding`	Authentication Method
`BasicAuthentication.username`	A string denoting the user name	User Name
`BasicAuthentication.password`	A string denoting the password	Password
`option.maxResponseContentSize`	A number denoting the maximum content size of the HTTP response in megabytes. A negative number corresponds to "unlimited".	Maximum Response Content Size

When changing the `authenticationMethod`, it may be necessary to provide additional configuration. When switching to `BasicAuthentication`, it is necessary to also provide `BasicAuthentication.username` and `BasicAuthentication.password`. When switching to `IMIdentityContextForwarding` it may be necessary to provide the Context-ID using `ISession.setImContextId(String)` when using Rule Execution API for execution.

Chapter 11. Marshalling and Unmarshalling

Rules have the ability to convert instances of simple data types and structures into XML representations and back again. The technical term for this is marshalling and unmarshalling. This chapter explains how various types are represented in XML when they are not the result of an XSD import.

11.1. XML Representation of Data Types

11.1.1. Simple Types

Simple types are mapped to the following XML schema types:

Rule data type	XML schema type
`String`	`xsd:string`
`Integer`	`xsd:integer`
`Float`	`xsd:decimal`
`Boolean`	`xsd:boolean`
`Date`	`xsd:date`
`Time`	`xsd:time`
`Timestamp`	`xsd:dateTime`

`String` values are simply specified as text (without quotes). For example, this XML fragment specifies a string value `"Peter"` for a data element `name`.

```
<name>Peter</name>
```

`Integer` values are specified as numbers. For example, this XML fragment specifies 199 as the integer value for a data element `seat_no`.

```
<seat_no>199</seat_no>
```

`Float` values are specified as numbers using a point as decimal separator. For example, this XML fragment specifies float values for different data elements. Please see an XML schema documentation for a more detailed format description of `xsd:decimal`.

```
<price>7.45</price>
<rate>-.4453</rate>
```

`Boolean` values are specified as the word `true` or `false`.The XML type is `xsd:boolean`.

```
<student>true</student>
<coupon>false</coupon>
```

`Date` values are specified with this format: `YYYY-MM-DD`, four digits for the year, two digits for the month, two digits for the day, all separated by dashes (-). Please see an XML schema documentation for a more detailed format description of `xsd:date`.

```
<show_date>2008-08-22</show_date>
```

`Time` values are specified with this format: `hh:mm:ss`, two digits for hour, minutes and seconds, separated by colons (:). No part may be omitted. Please see an XML schema documentation for a more detailed format description of `xsd:time`.

```
<alarm>15:47:23</alarm>
```

`Timestamp` values are specified according to the XML schema type `xsd:dateTime`. The format is `YYYY-MM-DDThh:mm:ss`, which is a date and a time separated by the letter `T`. No part may be omitted. Please see an XML schema documentation for a more detailed format description of `xsd:dateTime`.

```
<startTimestamp>2009-05-30T09:30:10</startTimestamp>
```

If values are to be explicitly set as empty (represented in Java by the keyword `null`), this can be achieved by setting the `xsi:nil` attribute. For this to work, the request must also import the namespace `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`.

```
<myDate xsi:nil="true" />
```

Please note that using the `xsi:nil` attribute only works for datatypes which support this. For example it is useless to specify this, when the datatype is a primitive Java type like `int`, because by definition they always have a value.

11.1.2. Structures

Values for structures in a request or a response are simply represented by elements for each attribute. For example, the following XML fragment represents the value of a data element named `customer` with two attributes `name` and `address`. `address` itself has an attribute named `zip`.

```
<customer>
   <name>John Doe</name>
   <address>
      <zip>12345</zip>
   </address>
</customer>
```

11.1.3. Lists and Sets

Lists and sets in a request or a response are represented by a sequence of `element` tags, each specifying one value in the list or set.

For example, this is a list (or set) of Strings:

```
<names>
   <element>John Doe</element>
   <element>Peter Pan</element>
   <element>Captain Hook</element>
</names>
```

And this is a list (or set) of customers, where each customer has the attributes `name` and `address`:

```
<customers>
   <element>
      <name>John Doe</name>
      <address>
         <zip>12345</zip>
      </address>
   </element>
   <element>
      <name>Peter Pan</name>
      <address>
         <zip>99999</zip>
      </address>
   </element>
   <element>
      <name>Captain Hook</name>
      <address>
         <zip>99996</zip>
      </address>
   </element>
</customers>
```

11.1.4. Maps

Maps are represented by a sequence of `entry` tags, each having two elements named `key` and `value`. For example, this is a `String -> Customer` mapping with two entries:

```
<customerMap>
   <entry>
      <key>8847-736-90</key>
      <value>
         <name>John Doe</name>
         <address>
            <zip>12345</zip>
         </address>
      </value>
   </entry>
   <entry>
      <key>2234-993-77</key>
      <value>
         <name>Peter Pan</name>
         <address>
            <zip>99999</zip>
         </address>
      </value>
   </entry>
</customerMap>
```

11.1.5. Enumerations

A value for an enumeration is specified simply by stating the desired literal. The following example specifies the value `GOLD` for a enumeration data element named `bonus_card`:

```
<bonus_card>GOLD</bonus_card>
```

Chapter 12. Reference

12.1. Properties Tabs

The following sections describe the Java- and XML-specific tabs available in the Properties view.

All the technical tabs listed here are *not* available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective.

12.1.1. Data Elements and Attributes

For some of the built-in data types it is possible to specify additional settings specific to the Java code generation. These settings can be done on the Java Implementation tab which is available for input/output data elements, internal data elements, constant data elements, as well as for attributes of structures.

For data types that have been imported from XML it is possible to see their original XML properties. These are displayed on the XML Representation tab, which is available for structures, attributes of structures, type aliases and service call parameters.

12.1.1.1. Java Implementation Tab

Depending on the data type of the data element this tab shows different options or none at all.

Integer

If the data element or attribute has the type Integer, the following settings are available on the Java Implementation tab:

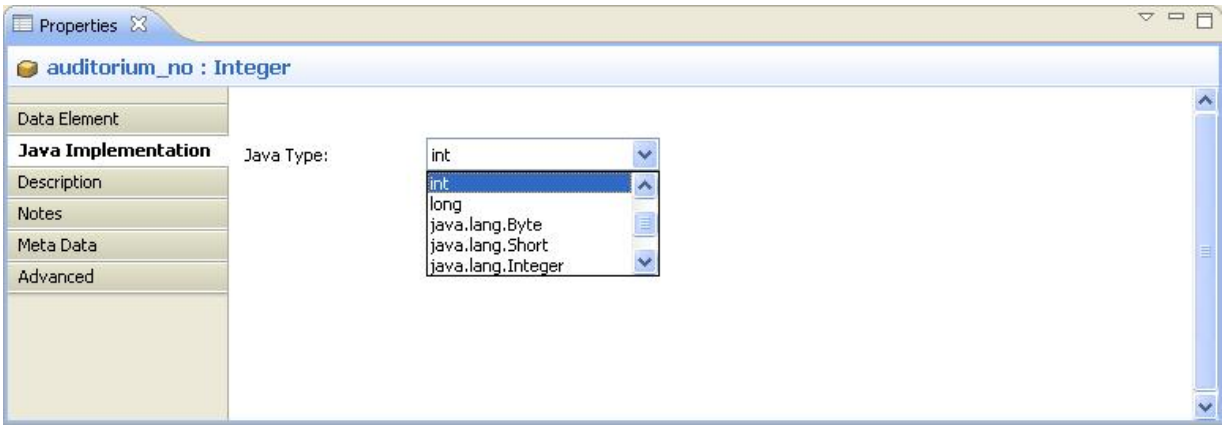


Figure 12.1. Java Implementation tab for an Integer

Option	Description
Java Type	Select one of the Java types available for an Integer: `byte`, `short`, `int`, `long`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.math.BigInteger` This will be the Java implementation used in the generated code. The default Java type used for an Integer is `int`.

Float

If the data element or attribute has the type Float, the following settings are available on the Java Implementation tab:

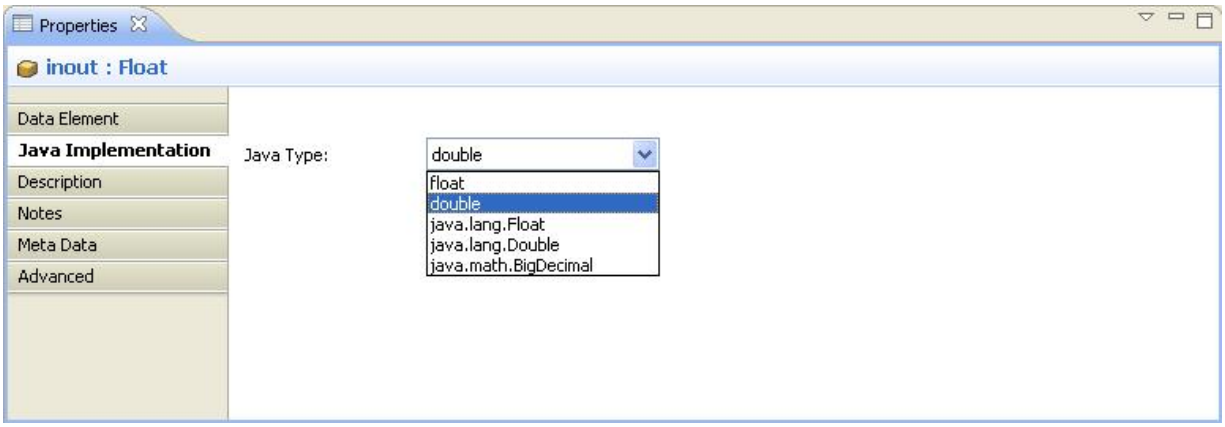


Figure 12.2. Java Implementation tab for a Float

Option	Description
Java Type	Select one of the Java types available for a Float: `float`, `double`, `java.lang.Float`, `java.lang.Double`, `java.math.BigDecimal` This will be the Java implementation used in the generated code. The default Java type used for a Float is `double`.

String

If the data element or attribute has the type String, the following settings are available on the Java Implementation tab:

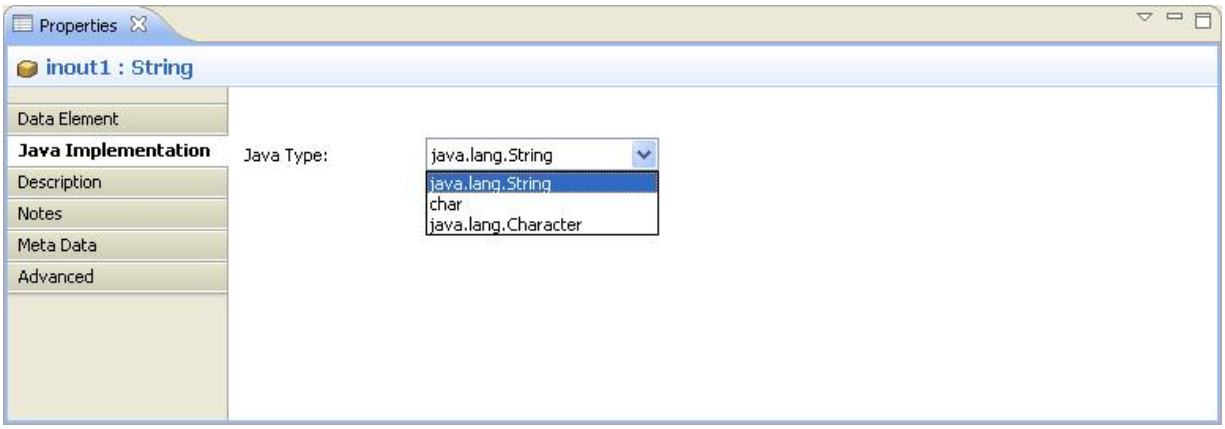


Figure 12.3. Java Implementation tab for a String

Option	Description
Java Type	Select one of the Java types available for a String: `char`, `java.lang.Character`, `java.lang.String` This will be the Java implementation used in the generated code. Note that `char` and `java.lang.Character` can only store one character and should be avoided whenever possible.

Option	Description
	The default Java type used for a String is `java.lang.String`.

Date

If the data element or attribute has the type Date, the following settings are available on the Java Implementation tab:

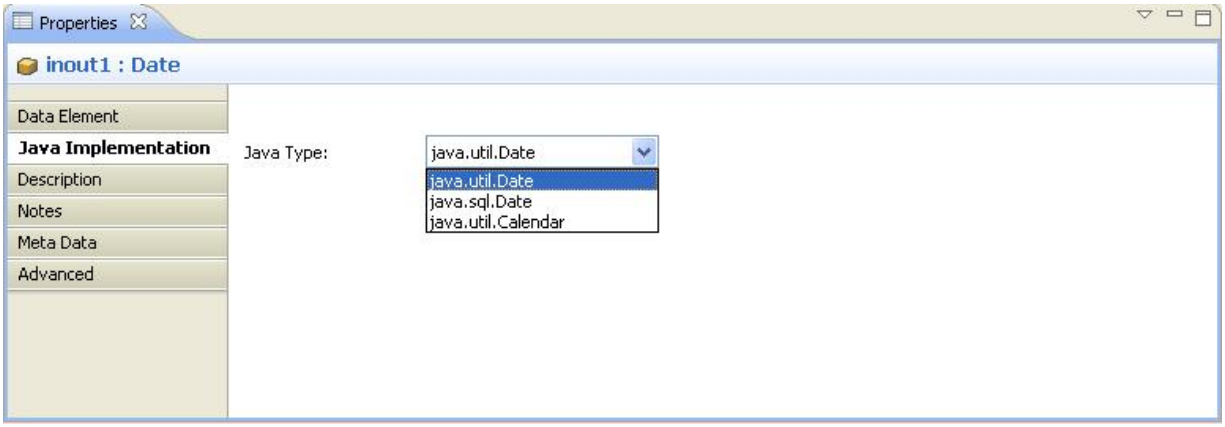


Figure 12.4. Java Implementation tab for a Date

Option	Description
Java Type	Select one of the Java types available for a Date: `java.util.Date`, `java.util.Calendar`, `java.sql.Date` This will be the Java implementation used in the generated code. The default Java type used for a Date is `java.util.Date`.

Boolean

If the data element or attribute has the type Boolean, the following settings are available on the Java Implementation tab:

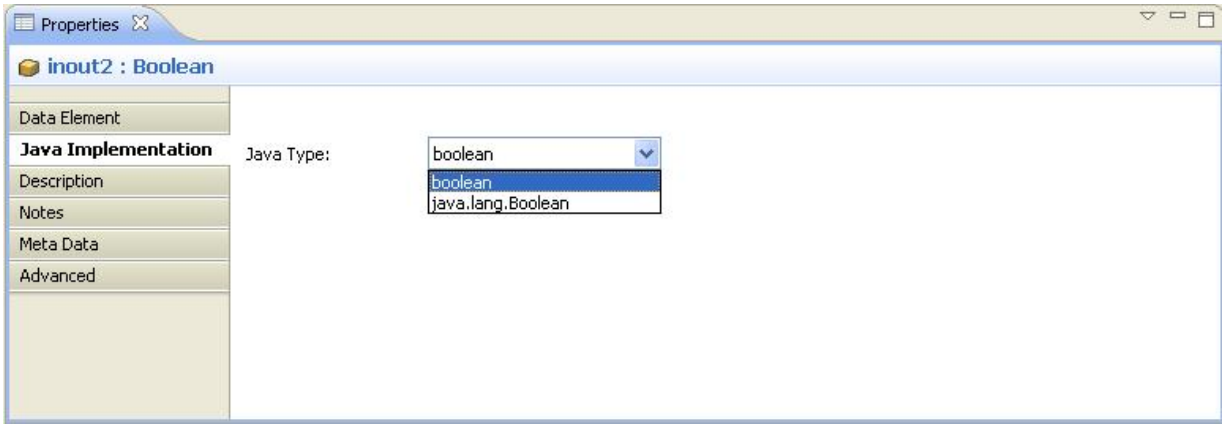


Figure 12.5. Java Implementation tab for a Boolean

Option	Description
Java Type	Select one of the Java types available for a Boolean: `boolean`, `java.lang.Boolean` This will be the Java implementation used in the generated code.

Option	Description
	The default Java type used for a Boolean is `boolean`.

List

If the data element or attribute is a list (multiple = true, unique = false), the following additional settings are available on the Java Implementation tab:

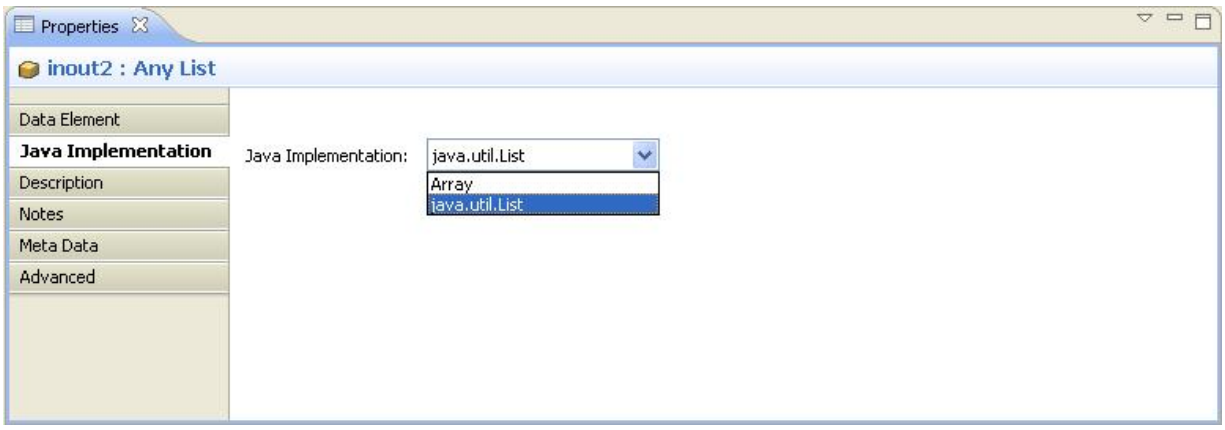


Figure 12.6. Java Implementation tab for a List

Option	Description
Java Implementation	Select an implementation for the list: `java.util.List`, `Array` This will be the Java implementation used in the generated code. The default Java implementation used for a list is `java.util.List`.

12.1.1.2. XML Representation Tab

Depending on the respective element this tab shows different information.

Structure

For a structure, the XML Representation tab provides information on the XML complex type, from which it was created:

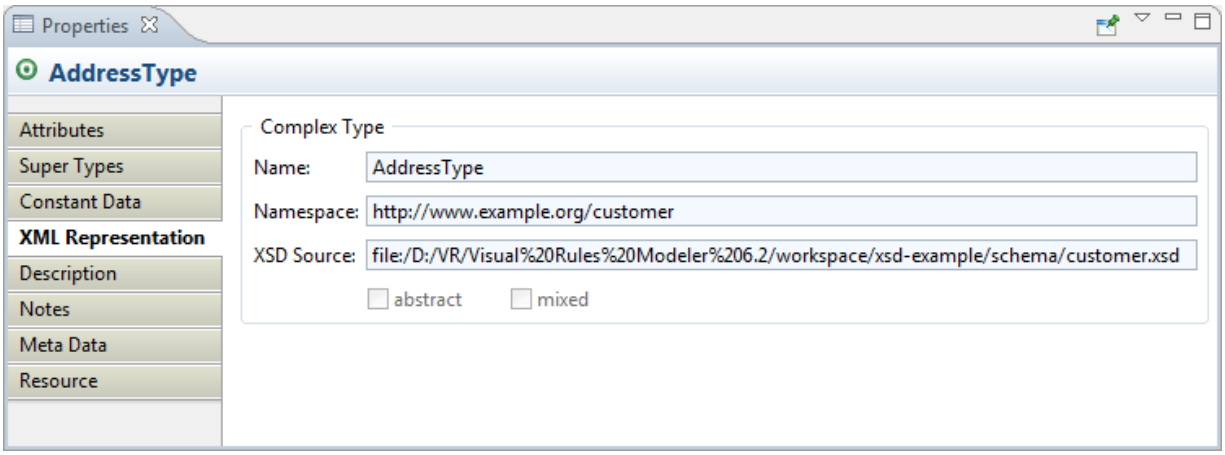


Figure 12.7. XML Representation tab for a Structure

Field	Description
Name	Name of the XML complex type
Namespace	Namespace of the XML complex type
XSD Source	URI of the resource that contained the XML complex type
abstract	Value of the attribute 'abstract' of the XML complex type
mixed	Value of the attribute 'mixed' of the XML complex type

Attribute

For an attribute of a structure, the XML Representation tab provides information on the XML element or attribute declaration, from which it was created:

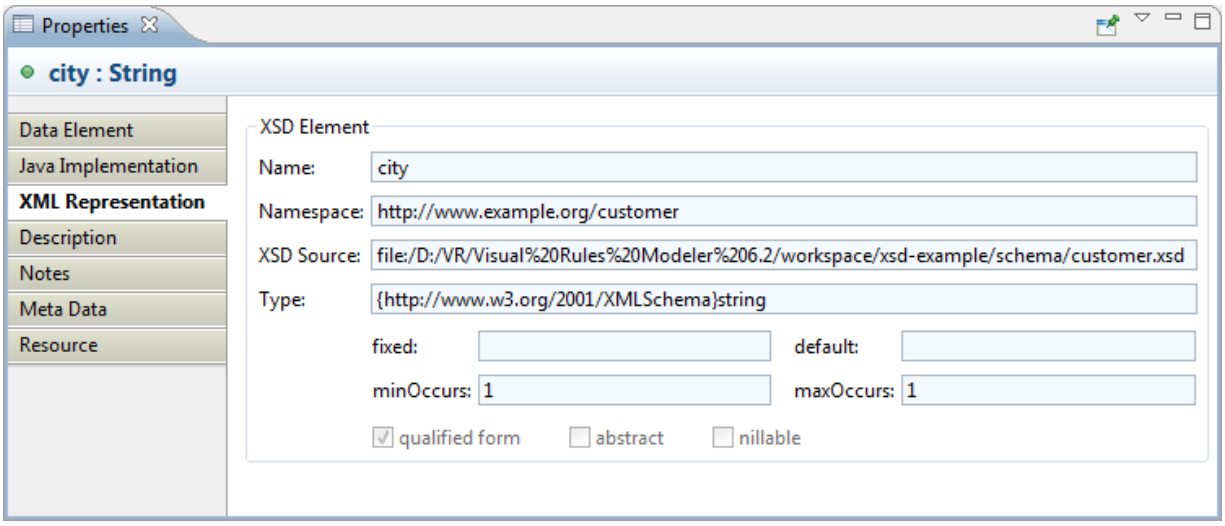


Figure 12.8. XML Representation tab for an Attribute

Some of the information is provided for XML element declarations only, i.e. not for XML attribute declarations.

Field	Description
Name	Name of the XML element/attribute declaration
Namespace	Namespace of the XML element/attribute declaration
XSD Source	URI of the resource that contained the XML element/attribute declaration
Type	Value of the attribute 'type' of the XML element/attribute declaration
fixed	Value of the attribute 'fixed' of the XML element/attribute declaration
default	Value of the attribute 'default' of the XML element/attribute declaration
minOccurs	Value of the attribute 'minOccurs' of the XML element declaration
maxOccurs	Value of the attribute 'maxOccurs' of the XML element declaration
qualified form	Value of the attribute 'form' of the XML element/attribute declaration
abtract	Value of the attribute 'abstract' of the XML element declaration
nillable	Value of the attribute 'nillable' of the XML element declaration

Type Alias

For a type alias, the XML Representation tab provides information on the XML simple type, from which it was created:

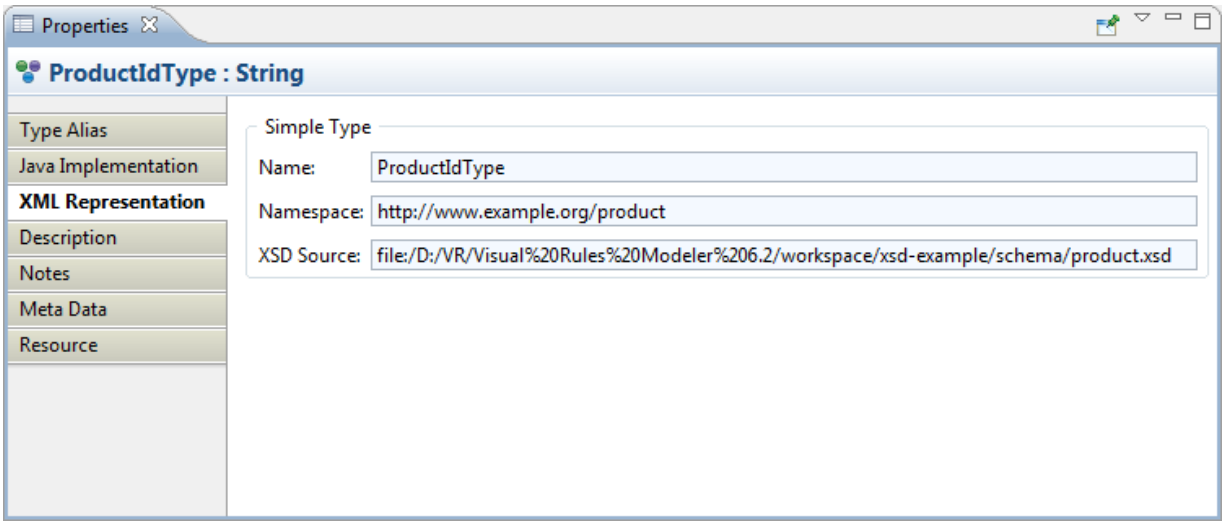


Figure 12.9. XML Representation tab for a Type Alias

Field	Description
Name	Name of the XML simple type
Namespace	Namespace of the XML simple type
XSD Source	URI of the resource that contained the XML simple type

Service Call Parameter

For a service call parameter, the XML Representation tab holds information on the XML element to which it corresponds. An XML element has a name and type:

Figure 12.10. XML Representation tab for a Service Call Parameter

Field	Description
Element Name	The full qualified name of the XML element.
Element Type	The full qualified name of the XML type (optional).

Full qualified names can be specified in the following format: `{namespace-URI}local-name`, e.g. `{http://www.visual-rules.com/namespace1}element1`. If there is no namespace, it is also sufficient to just specifiy the local name.

12.1.2. Services

The following properties tabs are available when a service is selected.

12.1.2.1. Service Settings Tab

This tab is shown when a service is selected. You can define settings specific to the selected service type here.

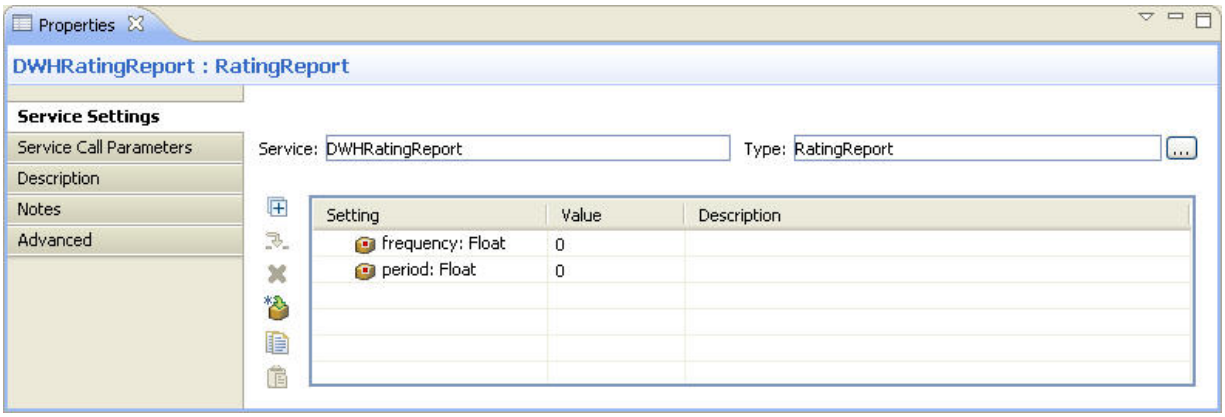


Figure 12.11. Service Settings tab

Option	Description
Name	Enter the name of the service.
Type	Enter the type of the service. Code Assist (**Ctrl**+**Space**) is available.

Table Column	Description
Setting	This column shows the available settings for this service type and their types.
Value	This column shows the expression you are assigning to the specific setting. You can edit cells in this column by double-clicking them.
Description	The description for this row. You can edit cells in this column by double-clicking them.

12.1.2.2. Service Call Parameters Tab

This tab is shown when a service is selected in the Rule Explorer or in the Rule Context palette of the rule editor. Here you can specify the parameters which will have to be specified when the service is called.

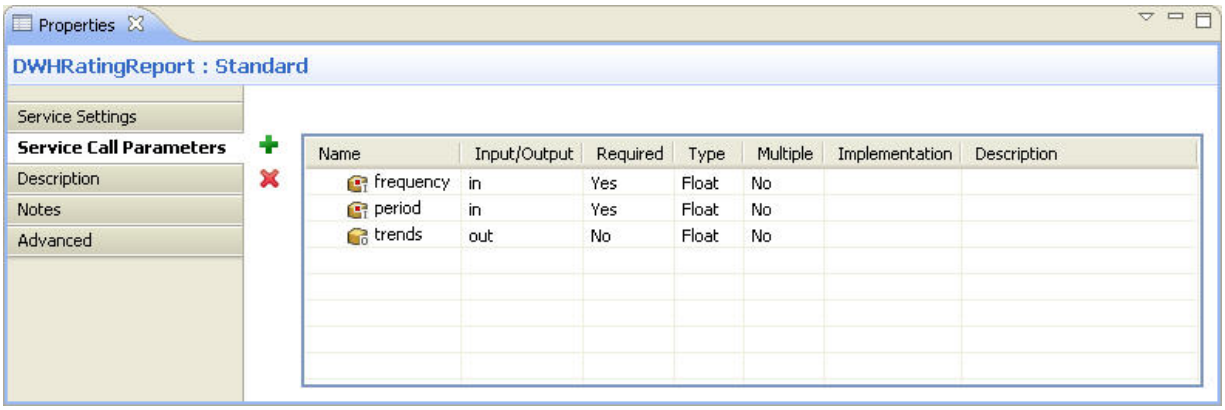


Figure 12.12. Service Call Parameters tab

Icon	Description
	Adds another parameter

Icon	Description
	Removes the currently selected parameter.

Table Column	Description
Name	This column shows the name of the service call parameter. You can edit cells in this column by double-clicking them.
Input/Output	This column shows whether the parameter is used for input, for output or for both. You can edit cells in this column by double-clicking them.
Required	This column shows whether the parameter is required. You can edit cells in this column by double-clicking them. Only input and input/output parameters can be required.
Type	The data type of the parameter. You can edit cells in this column by double-clicking them.
Multiple	This column shows whether the data element holds a single or multiple values of this type. You can edit cells in this column by double-clicking them.
Implementation	In this column you can specify the collection implementation to be used for a data element that will have multiple entries. You can edit a cell by double-clicking on it. The behavior of the collection depends on the implementation type you select, which can be either of the following three types available: List, Set, or Map. If selecting an implementation the Multiple column value will automatically be set to 'Yes' if it hasn't been already. Note: There is a fourth implementation called Collection, that can only appear after importing e.g. a JavaBean. It can neither be set nor edited.
Description	The description for this row. You can edit cells in this column by double-clicking them.

12.1.3. Actions

The following properties tabs are available when an action is selected.

All the technical tabs listed here are *not* available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective.

12.1.3.1. Action Configuration

This tab is used to technically configure the action. This includes selecting the action type implementation and setting the technical initialization parameters. If the rule model defines multiple configurations, then the implementation and the initialization parameters can be configured differently for each of these configurations.

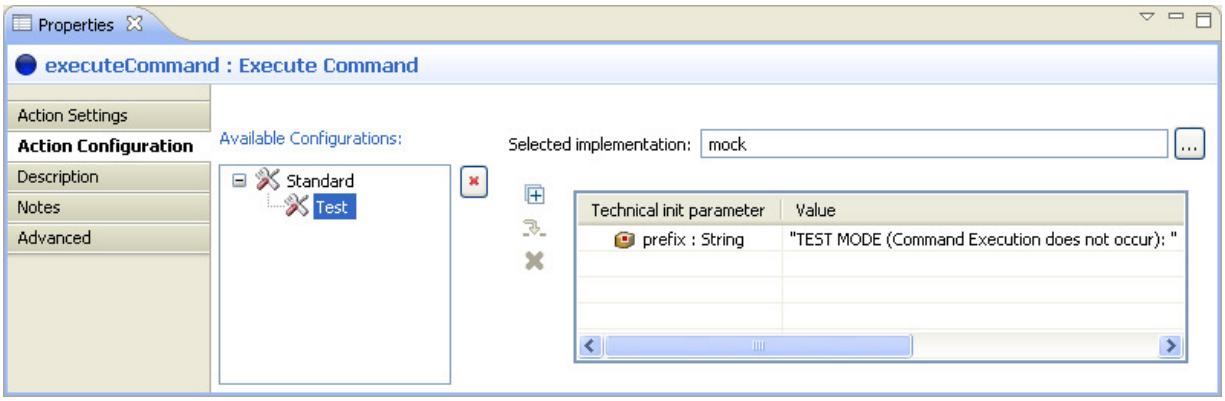


Figure 12.13. Action Configuration tab

Option	Description
Available Configurations	This lists all the configurations defined on the rule model (this is represented as a tree to visualize the fallback hierarchy). Select a configuration to see and change the action configuration for that configuration.
Selected implementation	If the action type has multiple implementation, this is used to select a specific implementation. Click on ... to browse available implementations.

Icon	Description
	Unconfigures the action for the selected configuration. Rules will then use the fallback configuration (the first not greyed-out configuration up the hierarchy).
	Initially all technical init parameters are collasped. This button will expand them so that all specified values are visible.
	Inserts another list element to an init parameter. This is only available when the currently selected init parameter is a collection, list or set.
	Removes the currently selected init parameter value. This is only available when the init parameter does not exist anymore.

Table Column	Description
Technical init parameter	This column lists all the technical init parameters for the selected implementation.
Value	The value for the technical initialization parameter. This can be an expression, but only include input/output data elements and constants.
Description	The description for this value.

Related Concepts.

- Configurations

Related Tasks.

- Defining a custom Action Type
- Selecting Configurations

Related References.

- Action Types
- Configurations Tab

12.1.4. Action Types

The following properties tabs are available when an action type is selected.

Some of the technical tabs listed here are *not* available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective.

12.1.4.1. Java Implementation Tab

This tab is used to define one or multiple Java implementations for an action type.

When you specify multiple Implementations you can configure which implementation you want to use according to the configuration that is in effect on the Action Configuration tab of an action.

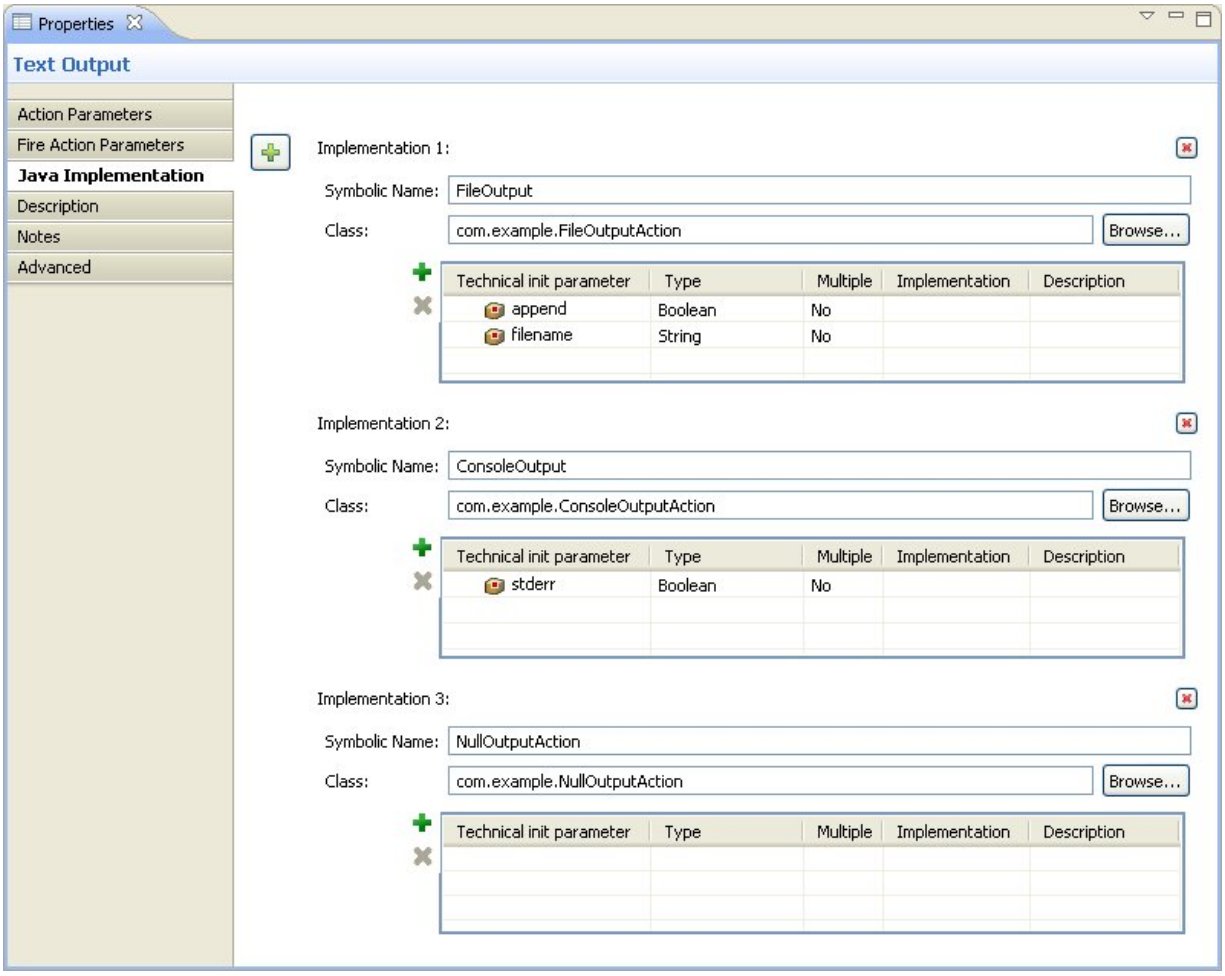


Figure 12.14. Java Implementation tab for an Action Type

Icon	Description
	Adds another Java implementation.
	Removes the corresponding Java implementation.
	Adds another technical init parameter for the given implementation.

Icon	Description
	Removes the selected technical init parameter.

Option	Description
Symbolic Name	The symbolic name of this implementation. This is used when an action is configured on the Action Configuration tab.
Class	This is the fully qualified class name of the Java class with the action implementation (implementing `IAction`).

Table Column	Description
Technical init parameter	This column shows the technical init parameters for this implementation. You can edit cells in this column by double-clicking them.
Type	The type of the init parameter. You can edit cells in this column by double-clicking them.
Multiple	This column shows whether the data element holds a single or multiple values of this type. You can edit cells in this column by double-clicking them.
Implementation	This column shows the collection type being used. Depending on the selection the collection can or can not hold duplicate values. You can edit cells in this column by double-clicking them. If selecting an Implementation then the Multiple column will automatically be set to Yes.
Description	The description for this technical init parameter. You can edit cells in this column by double-clicking them.

The parameters you specify in this table will be passed in a `java.util.Map` to the `initTechnical()` method of your action implementation.

Related Tasks.

- Defining a custom Action Type

12.1.4.2. Action Parameters Tab

This tab is shown when an action type is selected in the Rule Explorer. Here you can specify the parameters (init parameters) that have to be specified for every action of this action type.

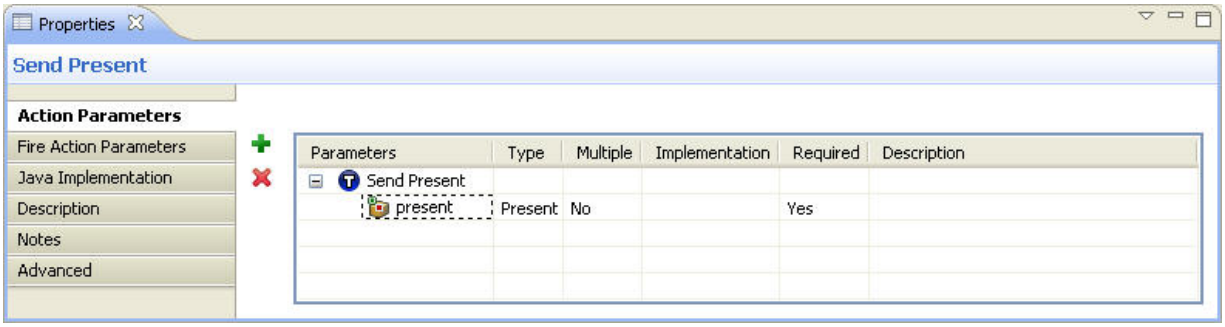


Figure 12.15. Action Parameters tab

Icon	Description
	Adds another action parameter.

Icon	Description
	Removes the selected action parameter.

Table Column	Description
Parameters	This column shows the parameters for actions of this type. You can edit cells in this column by double-clicking them.
Type	This column shows the type of the parameter. You can edit cells in this column by double-clicking them.
Multiple	This column shows whether the parameter holds a single or multiple values of this type. You can edit cells in this column by double-clicking them.
Implementation	In this column you can specify the collection implementation to be used for a data element that will have multiple entries. You can edit a cell by double-clicking on it. The behavior of the collection depends on the implementation type you select, which can be either of the following three types available: List, Set, or Map. If selecting an implementation the Multiple column value will automatically be set to 'Yes' if it hasn't been already. Note: There is a fourth implementation called Collection, that can only appear after importing e.g. a JavaBean. It can neither be set nor edited.
Description	The description for this parameter. You can edit cells in this column by double-clicking them.

12.1.4.3. Fire Action Parameters Tab

This tab is shown when an action type is selected in the Rule Explorer. In it you can specify the parameters that can be specified every time an action of this action type is invoked in a rule.

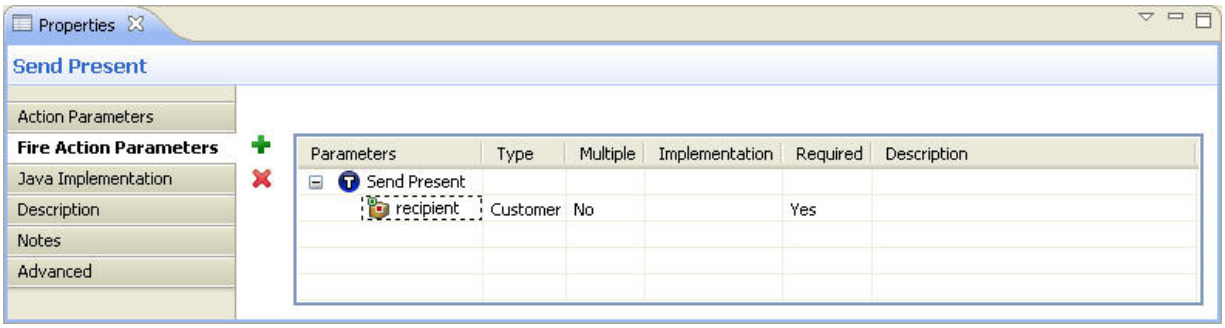


Figure 12.16. Fire Action Parameters tab

Icon	Description
	Adds another action parameter.
	Removes the selected action parameter.

Table Column	Description
Parameters	This column shows the action execution parameters for actions of this type. You can edit cells in this column by double-clicking them.
Type	This column shows the type of the parameter. You can edit cells in this column by double-clicking them.

Table Column	Description
Multiple	This column shows whether the parameter holds a single or multiple values of this type. You can edit cells in this column by double-clicking them.
Implementation	In this column you can specify the collection implementation to be used for a data element that will have multiple entries. You can edit a cell by double-clicking on it. The behavior of the collection depends on the implementation type you select, which can be either of the following three types available: List, Set, or Map. If selecting an implementation the Multiple column value will automatically be set to 'Yes' if it hasn't been already. Note: There is a fourth implementation called Collection, that can only appear after importing e.g. a JavaBean. It can neither be set nor edited.
Description	The description for this parameter. You can edit cells in this column by double-clicking them.

12.1.5. Functions

The following properties tabs are available when a function is selected.

Some of the technical tabs listed here are *not* available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective.

12.1.5.1. Java Implementation Tab

This tab is used to define one or multiple Java implementations for a function.

It is possible to provide multiple Java implementations for one function, typically if there are multiple methods which take similar types of parameters, e.g. either a `double` or a `java.math.BigDecimal`.

The code generator will always use the implementation which matches best in each case. It will select the implementation that does not require it to do any type conversion if possible. If a type conversion is necessary, it tries to do it with the smallest possible loss of precision.

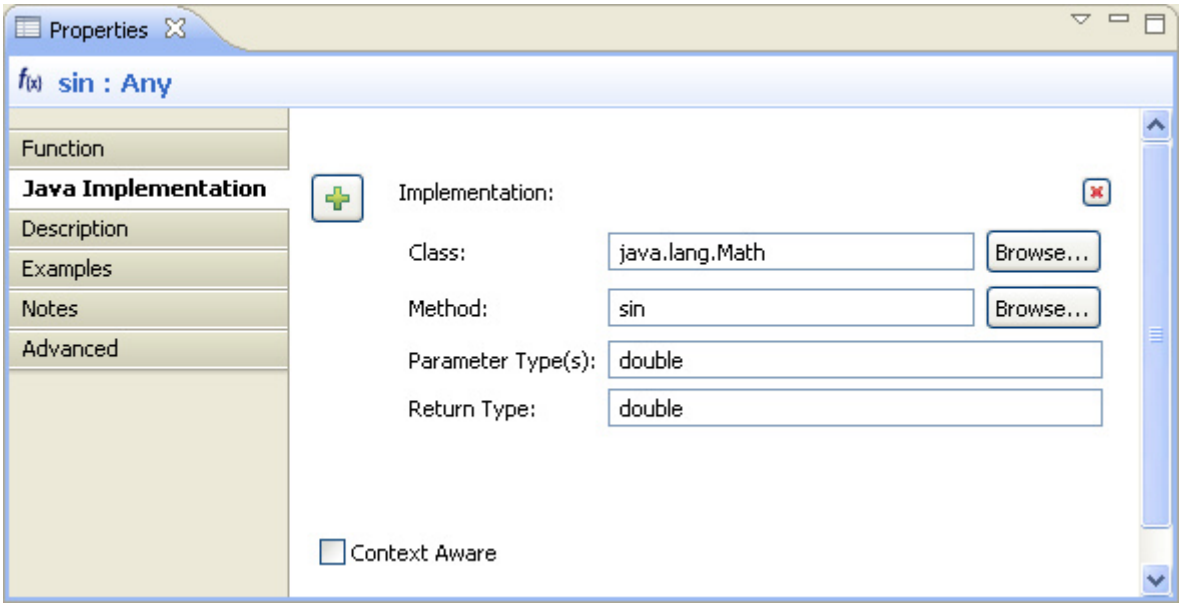


Figure 12.17. Java Implementation tab for a Function

Icon	Description
	Adds another Java implementation.

Icon	Description
	Removes the corresponding Java implementation.

Option	Description
Class	This is the fully qualified classname of a Java class that contains the function implementation. Browse... opens a class browser to choose from.
Method	This is the name of the method which contains the function implementation. Browse... opens a method browser to choose from.
Parameter Type(s)	This is a semicolon separated list of the Java types of the parameters. Use fully-qualified Java names for class/interface type parameters. For example: `int; com.acme.Customer; java.lang.BigDecimal`
Return Type	The return type of the method. Use a fully-qualified Java name for a class/interface return type. For example: `int, com.acme.Customer, java.lang.BigDecimal`
Context Aware	Activate this checkbox if the method has an additional parameter of type `IContextAwareRequestData` at the end of its parameter list. Through this interface it will have access to the current rule being executed, the node id, and all data elements, the request and the session. This setting is common to all implementations.

Related Tasks.

- Defining a custom Function

12.1.5.2. Function Tab

This tab is shown when a function is selected. It lists the parameters and the return value of the function, along with their data types and descriptions.

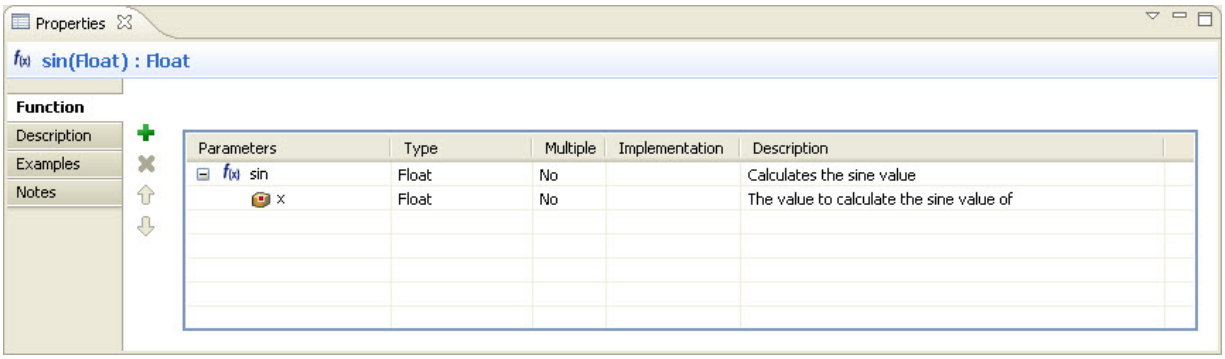


Figure 12.18. Function tab

Icon	Description
	Adds another parameter.
	Removes the selected parameter(s).
	Moves the selected parameter up in the list.

Icon	Description
	Moves the selected parameter down in the list.

Table Column	Description
Parameters	This column shows the parameters of the function. You can edit cells in this column by double-clicking them.
Type	This column shows the type of the parameter. In the first row this column shows the return type of the function. You can edit cells in this column by double-clicking them.
Multiple	This column shows whether the parameter holds a single or multiple values of this type. You can edit cells in this column by double-clicking them.
Implementation	In this column you can specify the collection implementation to be used for a data element that will have multiple entries. You can edit a cell by double-clicking on it. The behavior of the collection depends on the implementation type you select, which can be either of the following three types available: List, Set, or Map. If selecting an implementation the Multiple column value will automatically be set to 'Yes' if it hasn't been already. Note: There is a fourth implementation called Collection, that can only appear after importing e.g. a JavaBean. It can neither be set nor edited.
Description	The description for this row. You can edit cells in this column by double-clicking them.

12.1.5.3. Examples Tab

This tab is shown when a function is selected. It shows the examples that illustrate the usage of the function. They each consist of sample code and a description and will be displayed in the code assist for the function.

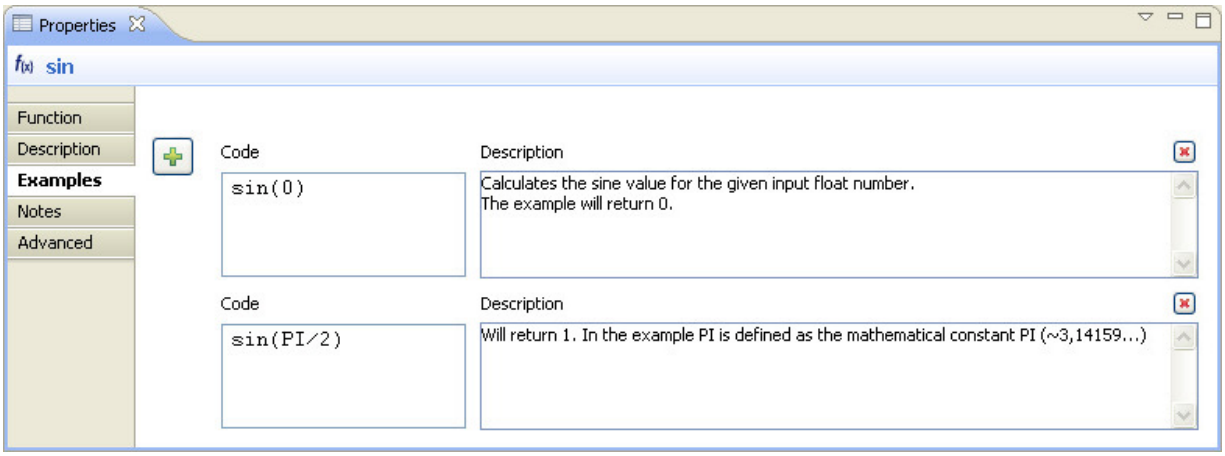


Figure 12.19. Examples tab

Icon	Description
	Adds another example.
	Removes the corresponding example.

Option	Description
Code	Enter sample code illustrating the function here, e.g. `stringMultiply(3, "Hello")`

Option	Description
Description	Enter a description for the sample code here, e.g. `will return the string "HelloHelloHello"`

12.1.6. Rule Packages

The following properties tabs are available when rule packages or rule models are selected.

All the technical tabs listed here are *not* available in the Rule Modeling perspective. You must switch to another perspective first, e.g. the Rule Integration perspective.

12.1.6.1. Java Code Generator Tab

This tab is used to configure the Java code generator within ACTICO Modeler. Some settings can be defined only for the whole rule model while others can be defined individually for every rule package.

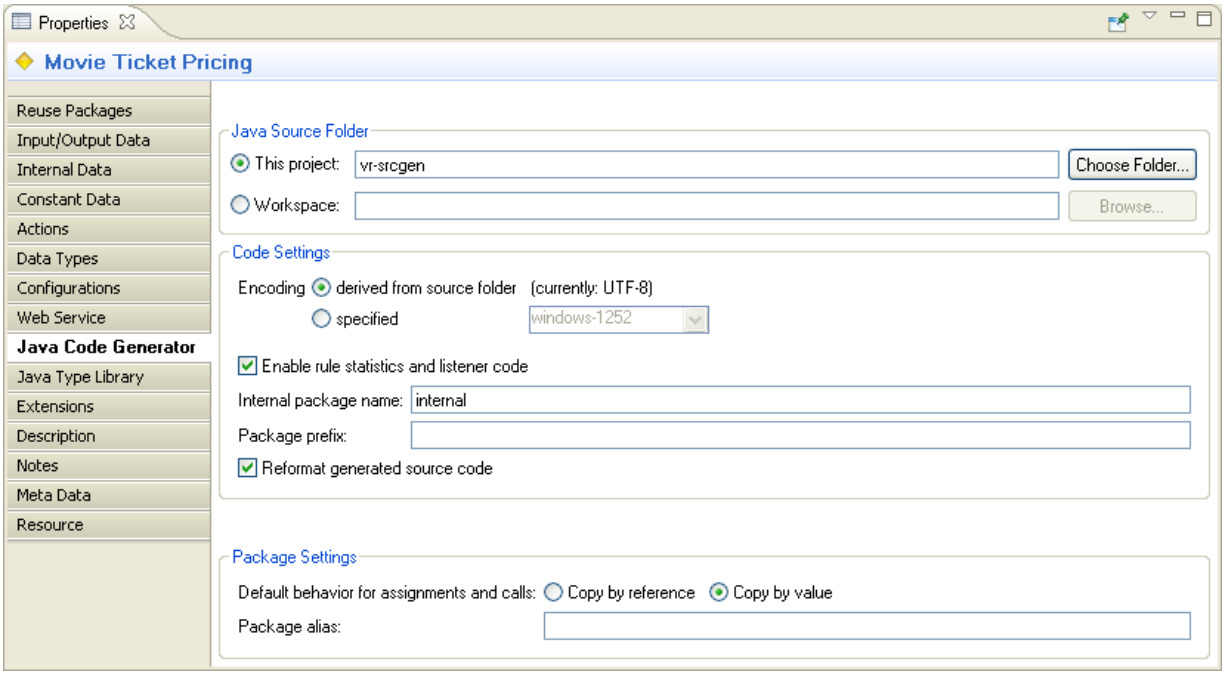


Figure 12.20. Java Code Generator tab

The following table lists the Java code generator options that are only globally available for the rule model.

Option	Description
Java Source Folder	Defines the output folder for the code generator where it puts the generated source code. In rule projects this defaults to the `vr-srcgen` folder in the same project, but it can be any folder of any project in the workspace.
Enable rule statistics and listener code	Enables the generation of the code necessary to support collection of execution statistics and custom listeners. Rule code generated with this option disabled, cannot produce statistics, but executes slightly faster.
Encoding	The encoding used for the generated source files. It can be derived from the source folder or can be specified. When derived, the encoding is taken from the source folder the code is generated in. This depends on the setting within Eclipse. When the encoding is specified, it is the same in every environment, however there may be the risk, that some encodings are not supported (e.g. Cp1252 is often not supported on Linux).

Option	Description
Internal package name	Name of the Java package used to separate the public classes and API from the internal classes and API. The default name is "internal". All classes in this Java package or in any subpackage thereof should be considered non-API.
Package prefix	If set, this prefix is prepended to all generated Java package names. For example, a prefix of `com.acme.rules` will generate all Java packages as subpackages of `com.acme.rules`.
Reformat generated source code	Specifies whether the generated source files should be formatted according to the Eclipse code formatter settings. See Window > Preferences > Java > Code Style > Formatter.

The following table lists the options that are available for every rule package.

Option	Description
Default behavior for assignments and calls	Defines the behavior of the generated code regarding assignments of structured values (objects). Two options are available: Copy by reference If an object is assigned to a data element, this data element afterwards simply refers to that object. This way, two different data elements may be referring to the same object. This is the preferred setting, as it is the default Java behavior and results in much faster rule execution. Copy by value If an object is assigned to a data element, the object is copied and the data element afterwards refers to that copy. Two different data elements do never refer to the same object. This option reduces the chance of mistakes in the rule logic and is best suited for non-technical rule authors. However, it usually has a significant impact on performance.  Please note that changing this setting may change the behavior of existing rules. Thorough testing should occur after changing this setting.
Package alias	If set, the package alias is used as the name of the Java package generated to represent this rule package. Per default, the Java package name is derived from the rule package name. Spaces and special characters are replaced with underscores and uppercase letters are converted to lowercase to comply with Java naming restrictions and conventions.

Related Concepts.

- Java Code Generation
- Copy-by-Reference / Copy-by-Value Semantics

12.1.6.2. Java Type Library Tab

In this tab you can import an external Java object model. Doing so allows you to interact with it directly from your rule logic.

Imported JavaBeans will appear in the rules project as if they were created within ACTICO Modeler. However imported types can't be edited by the user as no code will be generated for them by ACTICO Modeler.

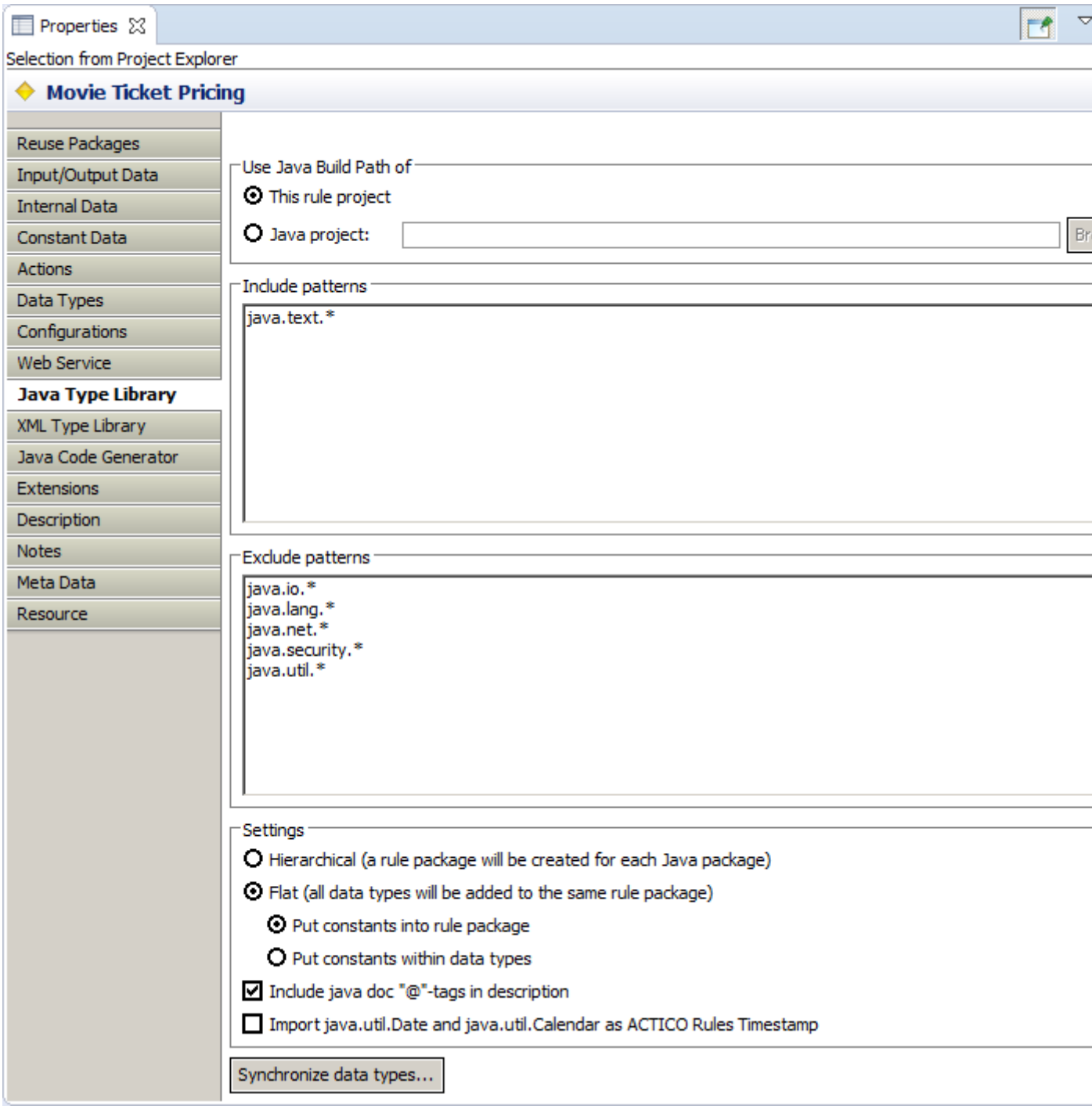


Figure 12.21. Java Type Library tab

Option	Description
Java Project	Defines the Java project from which the types should be imported. Any types you wish to import need to be on the classpath of that project.
Include patterns	Shows a list of include patterns. An include pattern can either be a fully qualified class name or a package prefix. A "*" acts as a wildcard and means all classes in the given package and all subpackages will the imported. Include patterns can be added and removed using the buttons to the right of the pattern list.
Exclude patterns	Shows a list of exclude patterns. An exclude pattern can either be a fully qualified class name or a package prefix. Single attributes can also be excluded by appending a hash (#) and the attribute name to the class name, e.g.

Option	Description
	`com.acme.IPersion#addresses` all attributes of a given class can be excluded by appending #* to the class name. Analogous to attributes constants can be excluded using a colon (:) instead of a hash. Exclude patterns can be added and removed using the buttons to the right of the pattern list. Additional exclude patterns can be added via the context menu in the preview dialog. Exclude patterns relating to constants and attributes can be added in the context menu of attributes and constants in the peview dialog exclusively.
Hierarchical	Defines that every imported Java Type will be placed within a rule package structure that correlates to the Java Type's package (resulting in a rule package hierarchy looking exactly like the Java package hierarchy).
Flat	Defines that all the imported classes will directly go into the same rule package the import is started on (resulting in a flat list of data types). When choosing Flat, it is nescessary to decide whether imported constants should be put into the rule package or within the data types defining them.
Include java doc "@"-tags in description	Defines whether java doc "@"-tags (for example `@author`, `@since`) should appear in the description of the imported elements. When this is unchecked these tags are omitted.
Import java.util.Date and java.util.Calendar as Timestamp	Defines whether the Java Types java.util.Date and java.util.Calendar, if used as the type of a property, are imported as Timestamp. If this option is not selected, the these Java Types are imported as Date.

Icon	Description
	This button is used to toggle the alphabetically sorting of the elements.
	Adds a new class to the include/exclude patterns respectively.
	Adds a new package to the include/exclude patterns respectively.
	Removes the selected include/exclude pattern.

Button	Description
Synchronize data types...	Starts the data type synchronisation and opens a preview dialog when done in which additional exclude patterns can be added via the context menu and the imported data types can be previewed.

Related Concepts.

- Java Type Library

Related Tasks.

- Importing Java Object Models

12.1.6.3. XML Type Library Tab

In this tab you can import external XML data types. Doing so allows you to interact with these directly from your rule logic.

Here for each individual namespace a rule package will be created. Imported XML data types will appear in the rules project as if they were created within ACTICO Modeler. However imported types can't be edited by the user.

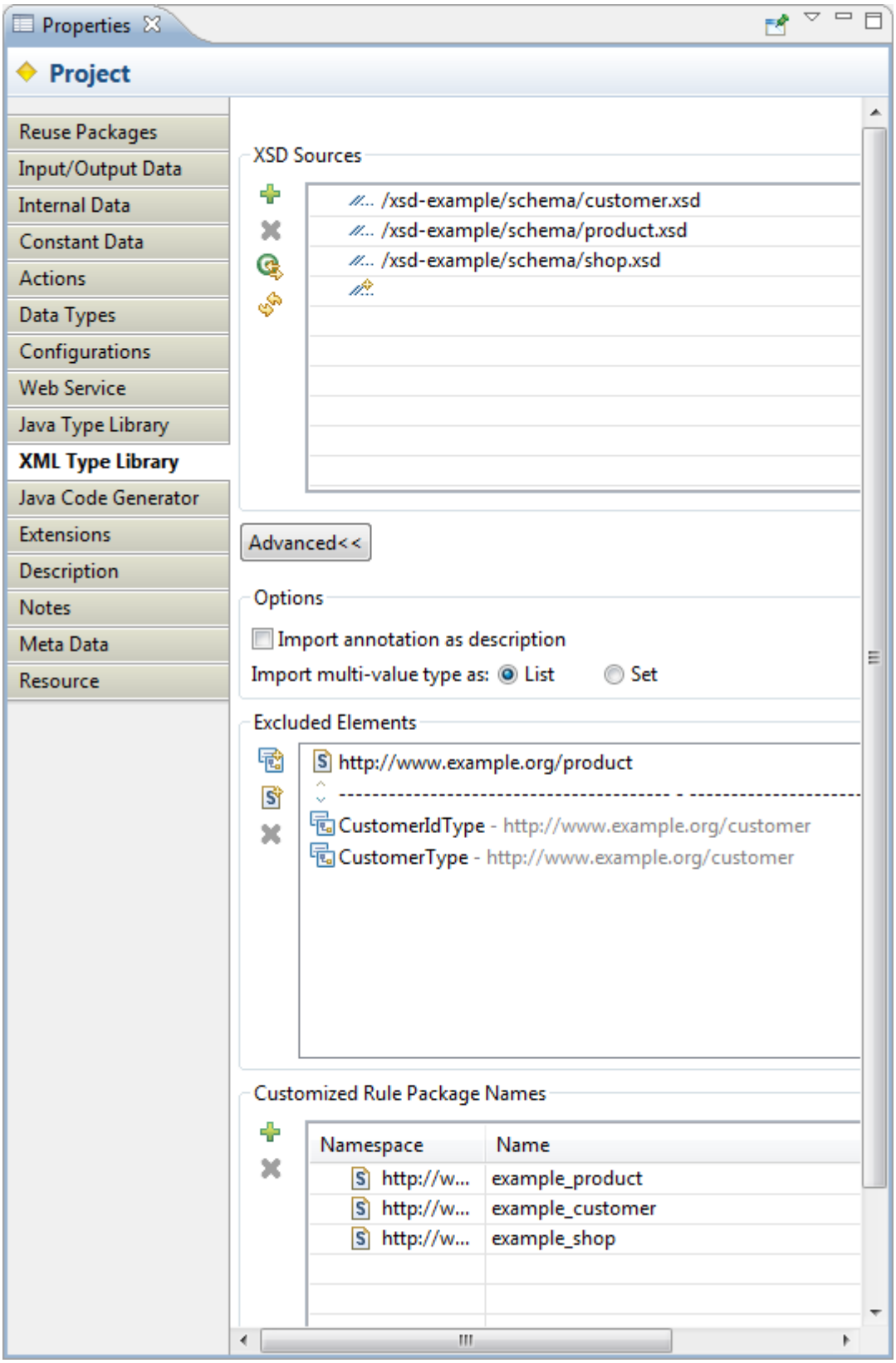


Figure 12.22. XML Type Library tab

Option	Description
XSD Sources	Shows a list of the available XSD sources.
Options	Shows:

Option	Description
	- whether documentation annotations in the XSD sources will be imported as descriptions - whether a multi-value data type will be imported as a list or as a set
Excluded Elements	Shows a list of namespaces and types, which are excluded from the import.
Customized Rule Package Names	Shows those rule packages to result from the import, for which the default names can be replaced by custom names.

Icon	Description
	Adding elements
	Deleting elements
	Starting the import (or synchronizing data types respectively)
	Reloading the view
	Adding XML types
	Adding XML namespaces

Related Concepts.

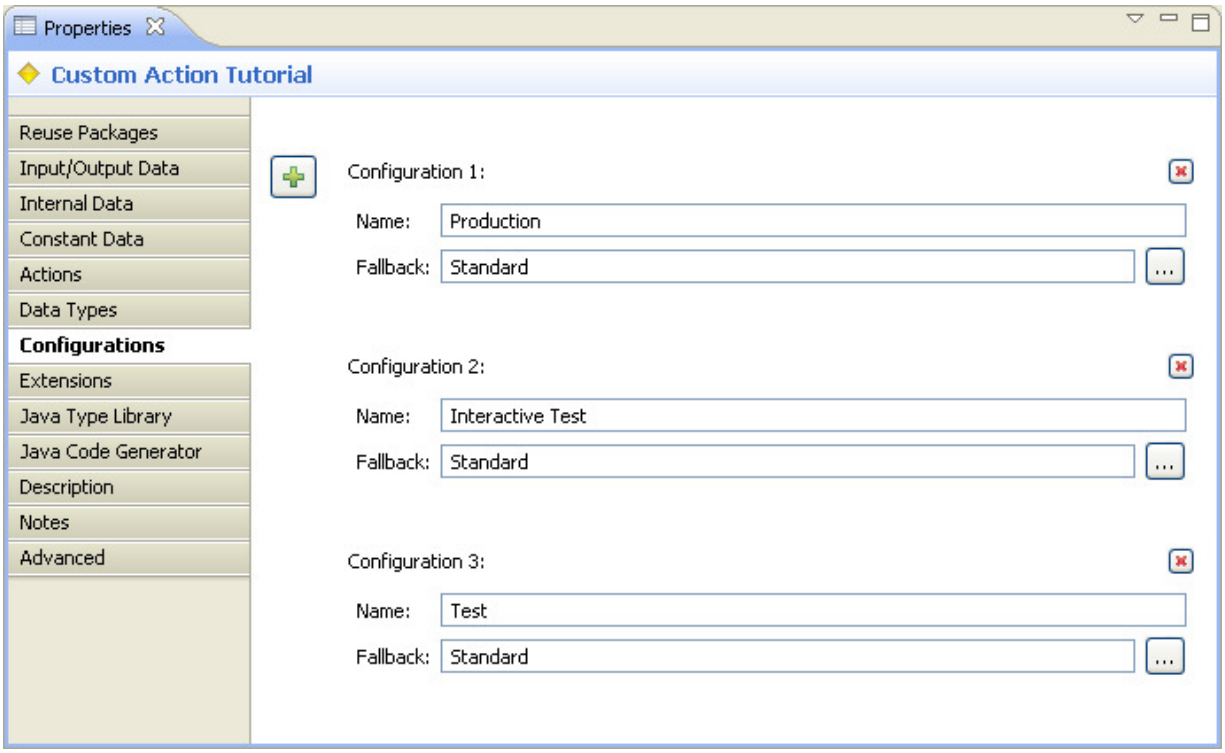
- XML Type Library

Related Tasks.

- Importing XML Data Types

12.1.6.4. Configurations Tab

In this tab you can specify configurations for your rule model. The definition of a configuration alone does not change the behaviour of the rule model. In the configuration of actions you can specify a different technical initialisation and implementation for a given configuration.



Option	Description
Name	The name of the configuration. You will refer to this configuration with this name. It is recommended to choose a name that describes the type of the configuration.
Fallback	The fallback for this configuration. If an action is not specifically configured for a configuration, then this fallback configuration is used instead. This allows to define which configuration is a specialization of which other configuration.

Related Concepts.

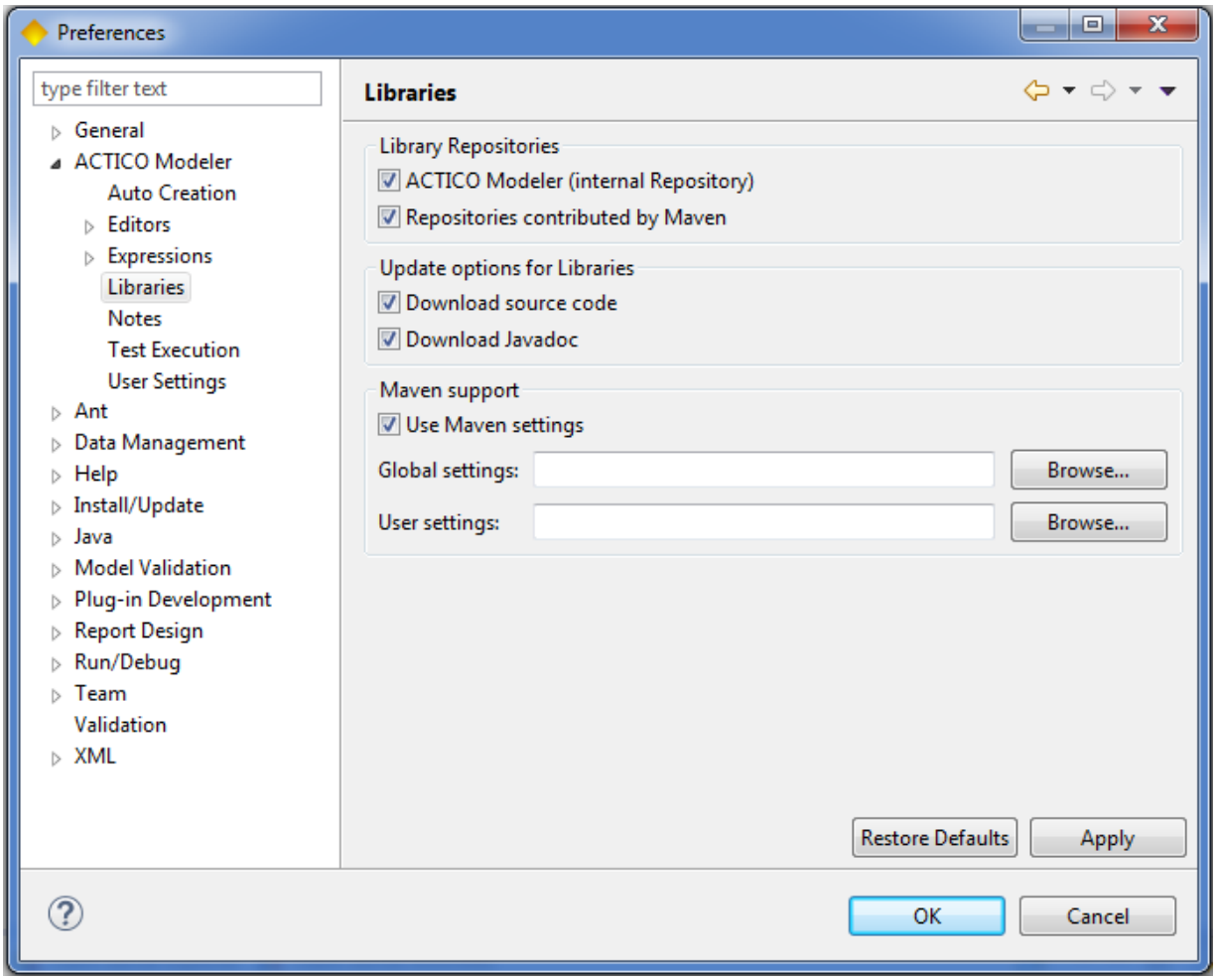
- Configurations

12.2. Preferences

The following sections describe ACTICO Modeler specific pages available in the Preferences dialog in the ACTICO Modeler section. The Preferences dialog can be opened with the menu Window > Preferences....

12.2.1. Libraries Preferences

The page Libraries contains settings for the handling of Libraries when resolving rule project dependencies.



Option	Description
ACTICO Modeler (internal Repository)	This option is used to enable or disable the ACTICO Modeler's internal repository. The internal repository contains the ACTICO Rules Runtime libraries and should only be deactivated if you have your own Maven repository containing the ACTICO Rules Runtime libraries.
Repositories contributed by Maven	By default, ACTICO Modeler includes a public repository into the resolution process for Libraries. This public repository is accessible via the internet and includes many technical Libraries like SQL database drivers, testing frameworks or other Java utilities. If you don't want these Libraries to be obtained from there (or to be accessed via the Internet), deactivate the checkbox.
Download source code	Whenever a Rule Project Dependency is resolved, the related Libraries are downloaded. If this checkbox is activated, the related source code is then downloaded as well.
Download Javadoc	Whenever a Rule Project Dependency is resolved, the related Libraries are downloaded. If this checkbox is activated, the related Javadoc is then downloaded as well.
Use Maven settings	Use this option if you have your own Maven settings and you want to use them in ACTICO Modeler.
Global settings	Here you can specify your own global Maven settings.
User settings	Here you can specify your own user Maven settings.

Related Concepts.

- Rule Project Identifiers, Dependencies and `ruleproject.vr`