

# **IBM Rational Rhapsody Developer RulesComposer Add On Tutorial**



## Notices

© Copyright IBM Corporation 1997, 2009.

US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to:

IBM Director of Licensing IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation Licensing  
2-31 Roppongi 3-chome  
Minato-ku Tokyo 106-0032, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you. ii This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software  
IBM Corporation  
1 Rogers Street  
Cambridge, Massachusetts 02142 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1997, 2009.

IBM, the IBM logo, ibm.com, Rhapsody, and Statemate are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at [www.ibm.com/legal/copytrade.html](http://www.ibm.com/legal/copytrade.html).

## Table of Contents

Notices .....	2
Overview .....	6
Features demonstrated .....	6
Create the RulesComposer project.....	6
Browse the source Rational Rhapsody model sample .....	7
Create a Java generation template.....	10
Create a ruleset to launch the Java template .....	12
Run the generation .....	14
Inspect the generated files.....	16
Add fields declaration .....	16
Add relations declaration .....	17
Add a script to handle relation multiplicity .....	18
Call the script from the template.....	21
Add methods declaration .....	21
Add return type declaration .....	24
Add arguments declaration .....	25
Add operation body.....	27
Use the debug hierarchy.....	29
Include a Javadoc template .....	30
Deploy the Launch Configuration .....	31
Use filenames associated to objects in Rational Rhapsody .....	34
Either use “On-Demand” or “Application” model reader .....	37
Customize filenames associated to Rational Rhapsody objects .....	38
Launch an external ruleset .....	41
Import, run and deploy "Rhapsody to Excel" sample .....	43
Import this ruleset .....	43
Run this ruleset .....	44
Deploy this ruleset .....	44

## Overview

In this tutorial, you will generate simple Java source files from an input Rational Rhapsody model.

The finished tutorial project is found in the default workspace in:

C:\Documents and Settings\<username>\workspace.

## Features demonstrated

This tutorial shows:

- How to navigate in models to determine the structure of the information to generate.
- How to generate code based on an input model using a text template.
- How to extend metatype capabilities using scripts.
- How to use a debug hierarchy to understand the flow of a generation.

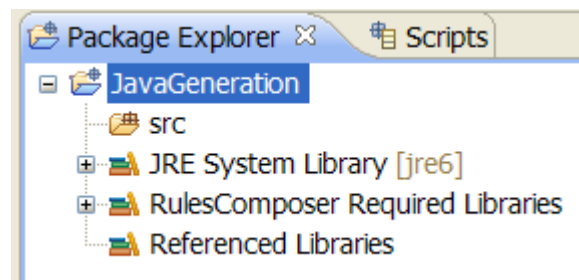
## Create the RulesComposer project

Generation rules must be contained in a RulesComposer project.

To create a RulesComposer project:

1. Click **File > New > Project...**
2. Select **RulesComposer Project** and click **Next**.
3. Type the name of the project to create (**JavaGeneration**) in the **Project name** field.
4. Click **Finish**.

The created project shows up in the Project Explorer view.



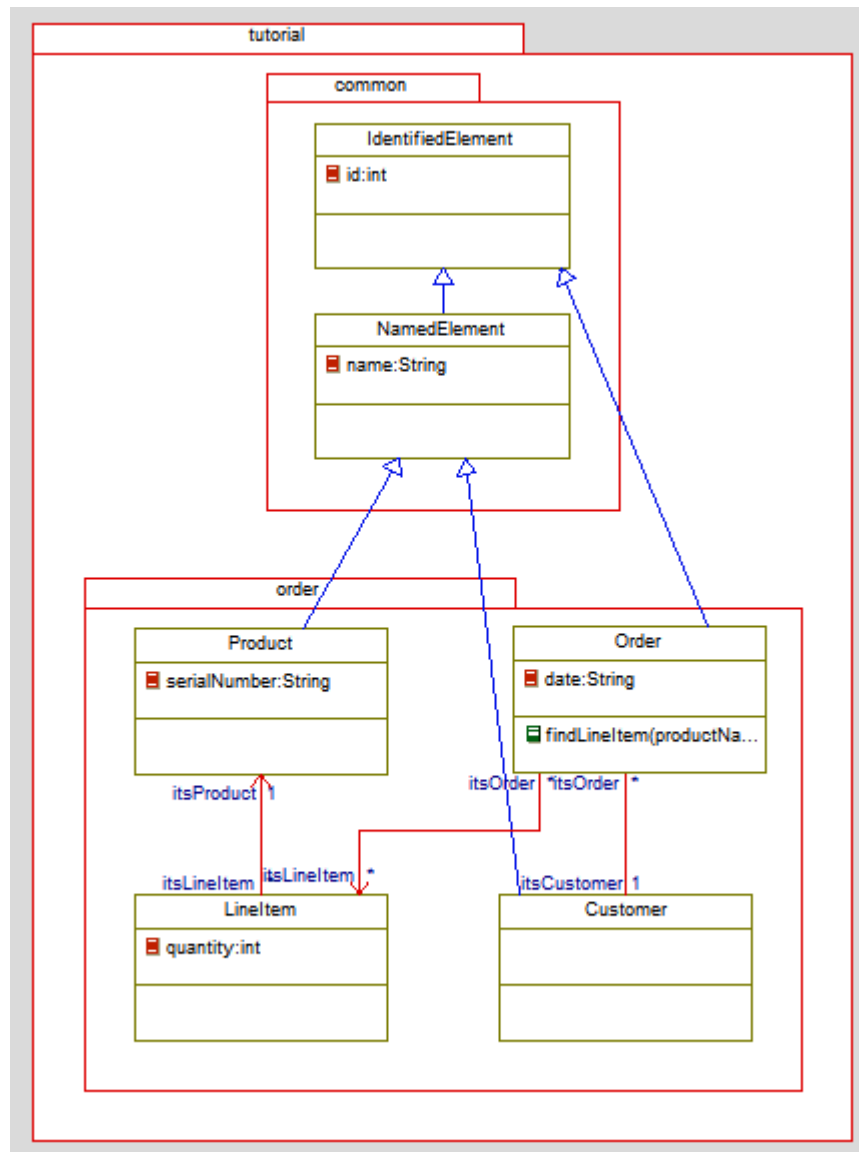
The project contains a **src** folder, where we will put the RulesComposer source files. It also references libraries:

- JRE System Library: runtime libraries required to compile and run Java code.
- RulesComposer Required Libraries: runtime libraries and metamodels required to compile and run RulesComposer files.

## Browse the source Rational Rhapsody model sample

To be able to write a RulesComposer program, we need to know the structure of the information to handle. We will open the sample model found in the `<rhapsodyInstallationDirectory>\Sodius\RulesComposer\help\tutorial\RhapsodyModel` in the Rational Rhapsody installation.

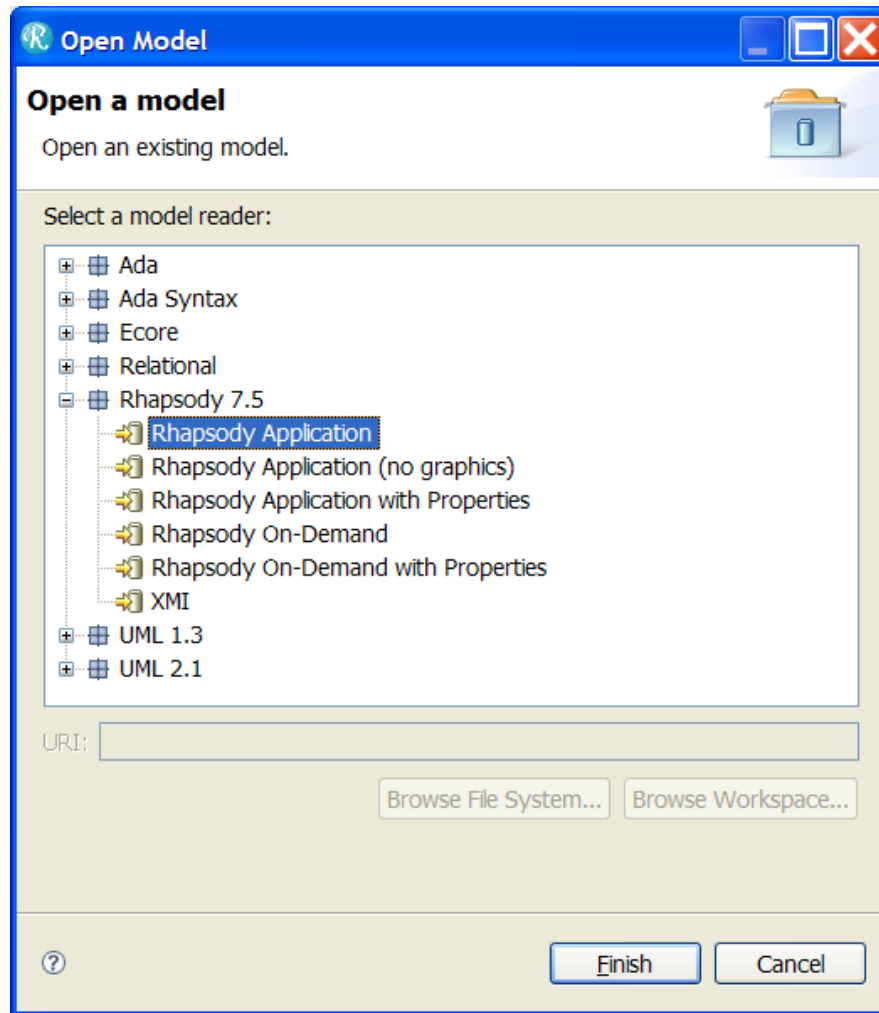
Here is a diagram of this model.



To open the sample model in the RulesComposer, the project must first be open in Rational Rhapsody. Start this application and open the `<rhapsodyInstallationDirectory>\Sodius\RulesComposer\help\tutorial\RhapsodyModel\RulesComposer_Tutorial.rpy` project.

Next read the model into the RulesComposer:

1. In the **Models** tab, right click on **Rhapsody**, and then **Open Model...**
2. Expand the **Rhapsody** label in the **Model reader** tree.
3. Select the **Rhapsody Application** element in the expanded tree.

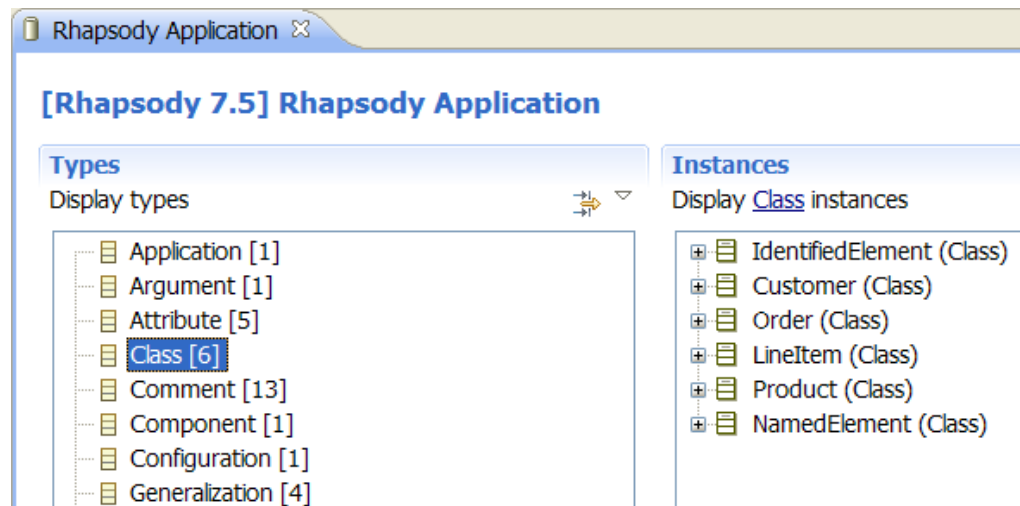


4. Click **Finish**.

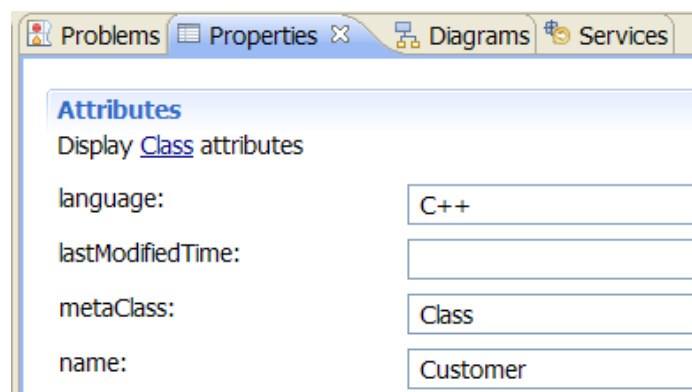
The Rational Rhapsody project is loaded and shows up in a model editor:

- The **Types** section displays types (e.g. **Class**) defined in the Rational Rhapsody metamodel, for which instances are defined in the loaded project.
- The **Instances** section displays instances (e.g. **Customer**) of the selected metatype.

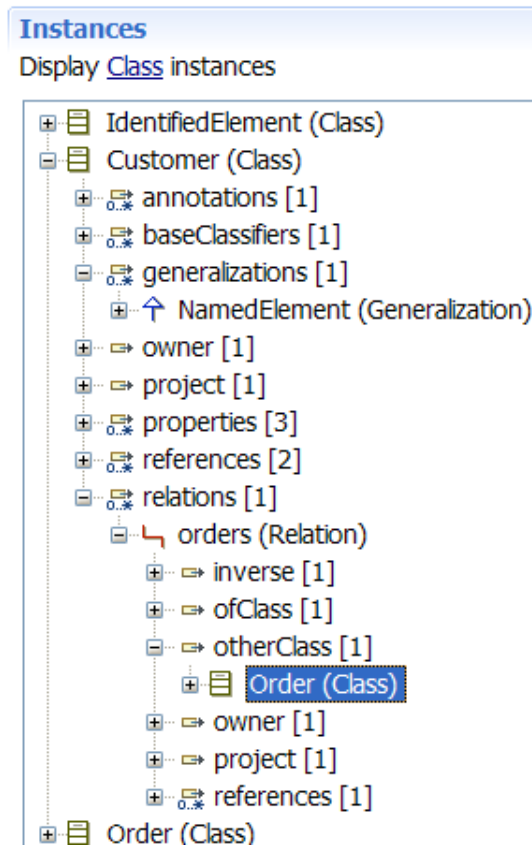




When you select an instance (e.g. **Customer**), you can display its properties in the **Properties** view (bottom of the screen):



The Properties view displays attributes (e.g. the name **Customer**) and references (to other instances) of the selected instance. The references of the selected instance can also be inspected in the **Instances** section of the model editor. Navigating through the children of the **Customer** instance for example is shown below:



You can see the Class **Customer** is linked to the Class **Order** through the reference **relations**.

Note: references starting with a ' / ' are derived references (computed from other attributes or references).

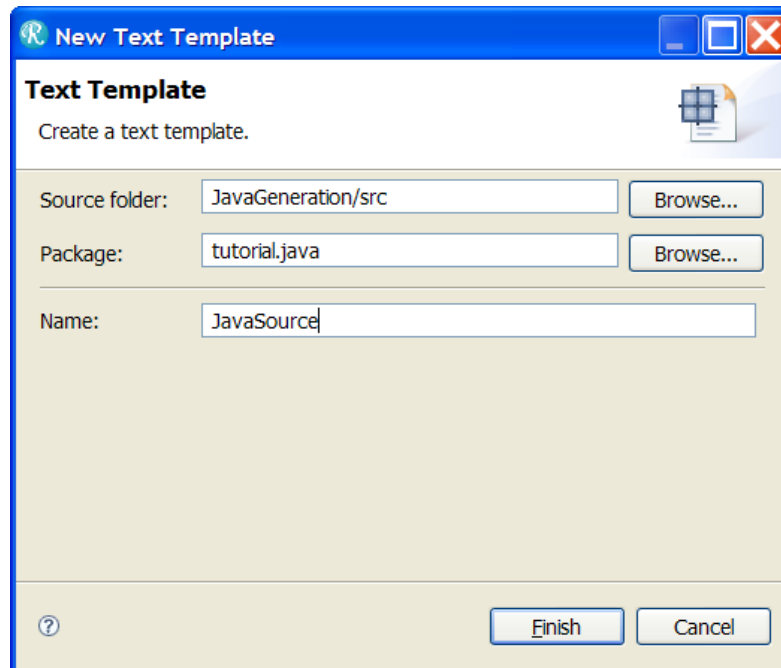
## Create a Java generation template

In previous steps, we created an RulesComposer project and inspected the structure of model information we need to handle. Now we are ready to create a generation program using a text template.

A text template specifies the information to generate in a given file as well as the name of the file.

To create a text template:

1. Click **File > New > Text Template**.
2. Type **JavaGeneration/src** in the **Source folder** field.
3. Type **tutorial.java** in the **Package** field.
4. Type **JavaSource** in the **Name** field.



5. Click **Finish**.

A file **JavaSource.tgt** is created in the folder **JavaGeneration/src/tutorial/java** and is opened in an editor.

```
[#package tutorial.java]

[#template public JavaSource()]
[#file]file.txt[/#file]
generated text
[/#template]
```

The text template **JavaSource** is defined in the package **tutorial.java** and generates the static text "**generated text**" in the file **file.txt**.

We want to generate a Java source file based on a Rational Rhapsody Class. Therefore we need to add a Rational Rhapsody Class parameter to the text template:

```
[#package tutorial.java]

[#template public JavaSource(class : rhapsody.Class)]
[#file]file.txt[/#file]
generated text
[/#template]
```

Add the code "`class : rhapsody.Class`" and click **File > Save**.

**rhapsody** is the metamodel identifier of the Rational Rhapsody metamodel, **Class** is the name of a metatype defined in this metamodel, and **class** is the name of the parameter.

The generated file name and contents must depend on the Rational Rhapsody Class name. So we need to change the contents of the **template** and **file** directives to reflect the name of the **class** parameter, using dynamic text. Dynamic text is a section delimited by `{` and `}` replaced with a calculated value in the output.

Change the text template contents to:

```
[#package tutorial.java]

[#template public JavaSource(class : rhapsody.Class)]
[#file]generated/${class.name}.java[/#file]
public class ${class.name} {
}
[/#template]
```

Now the generated contents will be the static text "public class ", followed by the **name** of the **class** parameter and by brackets. This generated contents will be written out in a file whose name is the **name** of the **class** parameter, extension is ".java", in a "generated" folder.

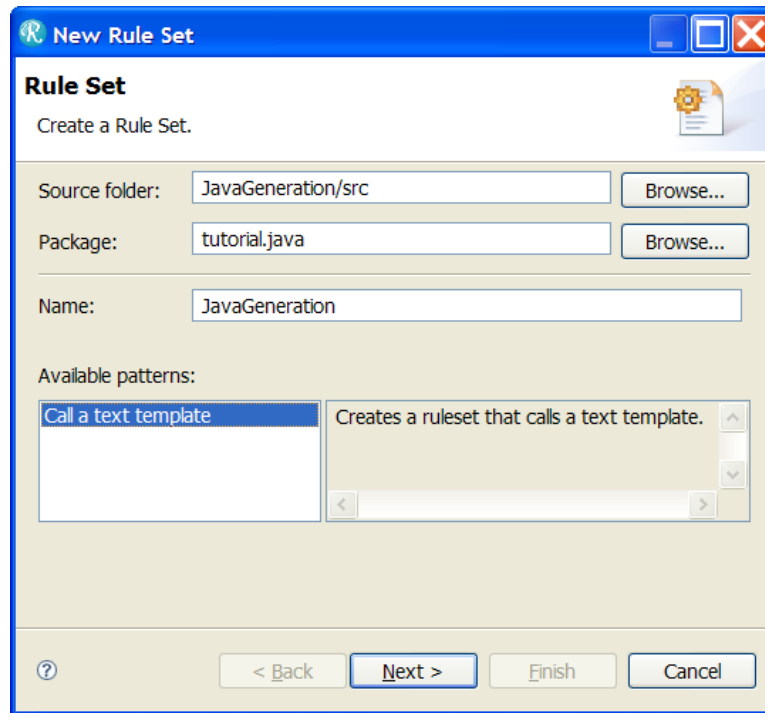
## Create a ruleset to launch the Java template

In previous step, we created a text template expecting a UML Class parameter to generate a Java source file. Now we want to iterate on all classes of a UML model and generate the Java source file for each of these classes. This is done using a ruleset.

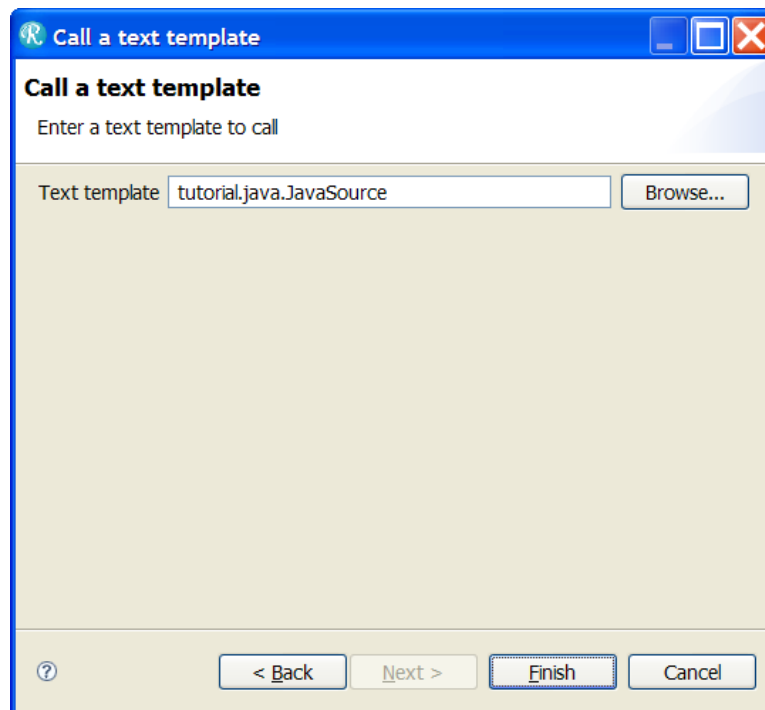
A ruleset is a group of logically interdependent rules, where a rule defines a set of procedural expressions that query or update model elements.

To create a ruleset:

1. Click **File > New > Rule Set**.
2. Type **JavaGeneration/src** in the **Source folder** field.
3. Type **tutorial.java** in the **Package** field.
4. Type **JavaGeneration** in the **Name** field.
5. Select **Call a text template** in the **Available patterns** section.



6. Click **Next**.
7. Type **tutorial.java.JavaSource** in the **Text template** field, on second page.



8. Click **Finish**.

A file **JavaGeneration.mqr** is created in the folder **JavaGeneration/src/tutorial/java** and is opened in an editor.

```
package tutorial.java;

public ruleset JavaGeneration(in model : rhapsody) {

    public rule main() {
        foreach (class : rhapsody.Class in model.getInstances("Class"))
        {
            $JavaSource(class);
        }
    }
}
```

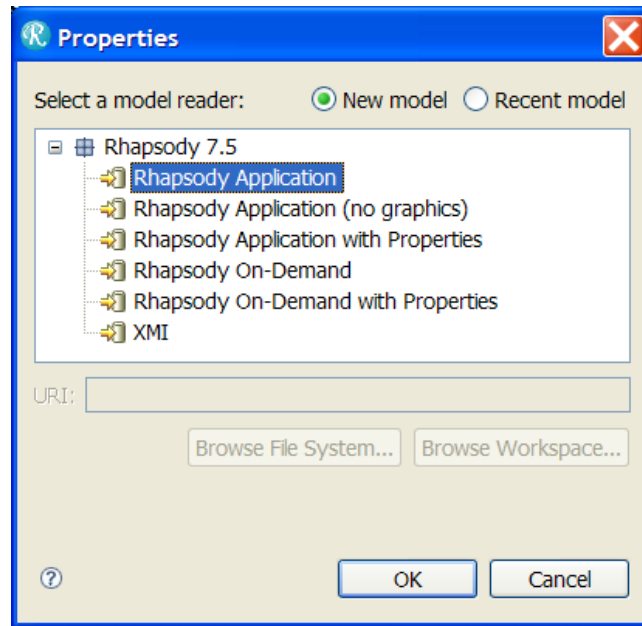
Here is the behavior of this ruleset:

1. Expects an input Rational Rhapsody model (the direction **in** tells RulesComposer the model parameter is expected to read from Rational Rhapsody).
2. Retrieves the list of instances of the type `Class` in the input Rational Rhapsody project, using `model.getInstances("Class")`.
3. Iterates on this instance list using **foreach** and the loop variable `class` of type Rational Rhapsody `Class`.
4. Calls the text template `JavaSource` with the `class` variable as argument ('`$`' is the notation used to call a text template). The referenced template is evaluated and the generated contents is written on disk.

## Run the generation

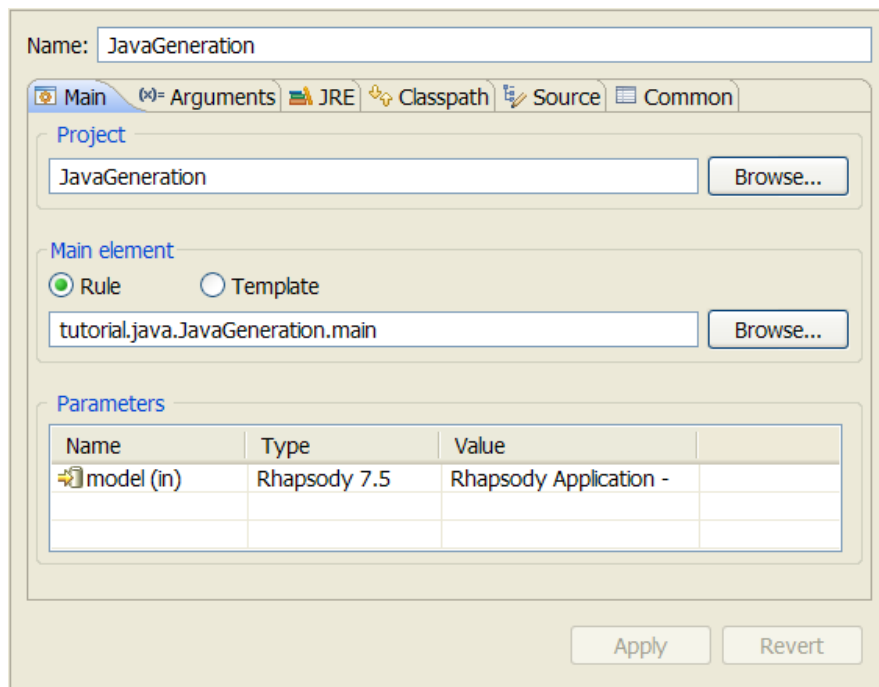
We can now run the ruleset using a sample model:

1. Click **Run > Run...**
2. Select **RulesComposer** in the left hand list of launch configuration types, and press **New**.
3. Type **JavaGeneration** in the **Project** field.
4. Click **Rule** in the **Main element** section.
5. Type **tutorial.java.JavaGeneration.main** in the **Main element** field.
6. Select the Rational Rhapsody model:
  - a. Click on the **model** parameter (first line of the **Parameters** section) to display the model selection dialog.
  - b. Select **Rhapsody** and then **Rhapsody Application** in the tree.



c. Click **OK**.

7. Click **Run**. The launch configuration should look like this:



RulesComposer evaluates the ruleset using the specified models and prints messages in the **Console** view:

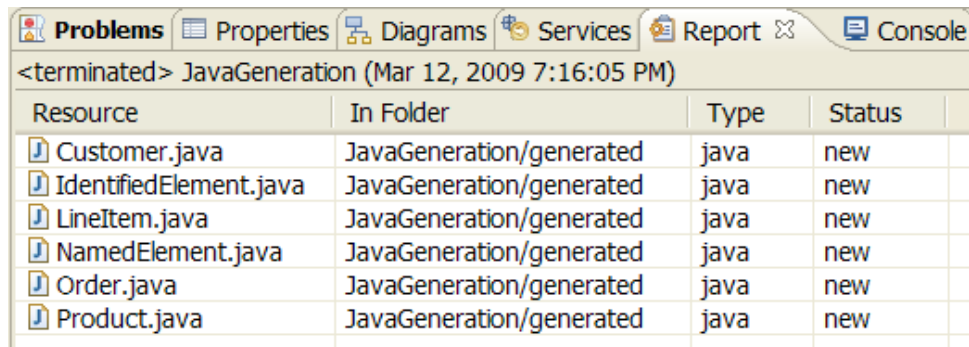
```

[progress] Evaluation of JavaGeneration.main
[progress] Reading Rhapsody Application
[progress] Reading Application
[progress] Reading Project : RulesComposer_Tutorial
...
[progress] Generating generated\Customer.java
[progress] Generating generated\Order.java
[progress] Generating generated\Product.java
[progress] Generating generated\LineItem.java
[progress] Generating generated\IdentifiedElement.java
[progress] Generating generated\NamedElement.java
[progress] Done.

```

## Inspect the generated files

When the generation is completed, an evaluation report is created and displayed in the **Report** view.



Resource	In Folder	Type	Status
Customer.java	JavaGeneration/generated	java	new
IdentifiedElement.java	JavaGeneration/generated	java	new
LineItem.java	JavaGeneration/generated	java	new
NamedElement.java	JavaGeneration/generated	java	new
Order.java	JavaGeneration/generated	java	new
Product.java	JavaGeneration/generated	java	new

This report shows the generated file locations, as well as the status of the generated file (**new** for newly generated file, that didn't exist before the generation). You can open a generated file from this Report view double-clicking the **Customer.java** file for example

The Java editor shows up and displays the created Java class:

```

public class Customer {
}

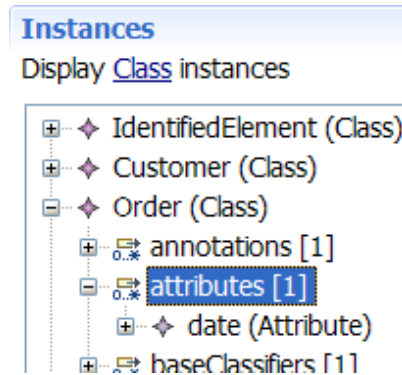
```

## Add fields declaration

Now we want to generate a Java field for each attribute defined on a Class:

1. Read the Rhapsody project.
2. Select the type **Class**.
3. Select the Class **Order** and show its children.





You can see the attributes are linked to their class with the reference **attributes**. The text template has to iterate on the **attributes** list and print the corresponding Java field declaration:

```
[#package tutorial.java]

[#template public JavaSource(class : rhapsody.Class)]
[#file]generated/${class.name}.java[/#file]
public class ${class.name} {

    [#-- Attributes declaration --]
    [#foreach attr : rhapsody.Attribute in class.attributes]
        private ${attr.type.name} ${attr.name};
    [/#foreach]
}
[/#template]
```

The template loops on each attribute and prints the name of its type (dynamic text `${attr.type.name}`) followed by its name.

`[#-- Attributes declaration --]` is a comment. A comment is delimited by `[#--` and `-]` and is not written to the output. Note that due to automatic whitespace stripping, whitespaces and linefeeds on line containing only directives and comments are ignored.

Relaunch the generation and open the file **Order.java**:

```
public class Order {

    private String date;

}
```

## Add relations declaration

Now we want to generate a Java field for each relation defined on a Class like we did for attributes.

You can see the relations are linked to their class with the reference **relations**. The text template has to iterate on the **relations** list and print the corresponding Java field declaration:

```
[#package tutorial.java]

[#template public JavaSource(class : rhapsody.Class)]
[#file]generated/${class.name}.java[/#file]
public class ${class.name} {

  [#-- Attributes declaration --]
  [#foreach attr : rhapsody.Attribute in class.attributes]
    private ${attr.type.name} ${attr.name};
  [/#foreach]

  [#-- Relations declaration --]
  [#foreach rel : rhapsody.Relation in class.relations]
    private ${rel.otherClass.name} ${rel.name};
  [/#foreach]
}
[/#template]
```

Relaunch the generation and open the file **Order.java**:

```
public class Order {

    private RhpString date;

    private Customer customer;
    private LineItem items;

}
```

## Add a script to handle relation multiplicity

In the previous step, we generated a Java field for each relation, regardless of its multiplicity. For example the relation **items** of the Class **Order** in the sample model has a multiplicity \*. Its corresponding Java field type should be `java.util.Collection`, not `LineItem`.

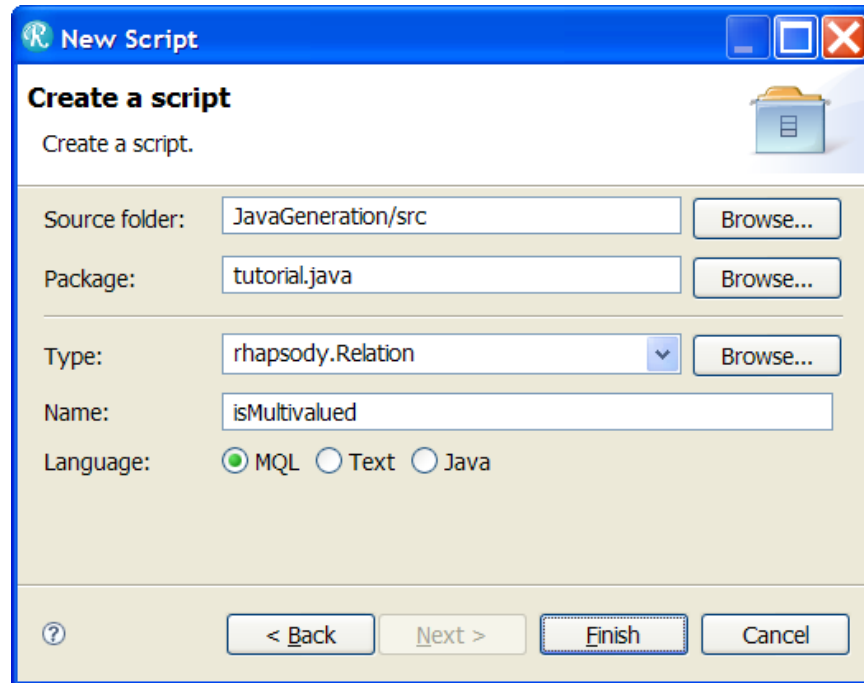
The type **Relation** has an attribute `multiplicity` that contains this information. We have to update the generation to introduce an `if` directive to test the `multiplicity` value for each `Relation`. If the multiplicity is not equal to 1, we will consider the relation to be multivalued. We could add this logic directly inside the text template, but because the text to generate depends on each `Relation`, we will introduce two scripts to maintain the readability of the script. A script is a method dynamically added to a metatype.

The first script we will add will be called `isMultivalued`.

To add a script:

1. Click **File > New > Script**.

2. Type **JavaGeneration/src** in the **Source folder** field.
3. Type **tutorial.java** in the **Package** field.
4. Type **rhapsody.Relation** in the **Type** field.
5. Type **isMultivalued** in the **Name** field.
6. Click **MQL** in the **Language** group.



7. Click **Finish**.

A file `rhapsody_Relation.mqs` is created: it allows us to define MQL scripts on the metatype `Relation`. Change its contents to:

```
package tutorial.java;

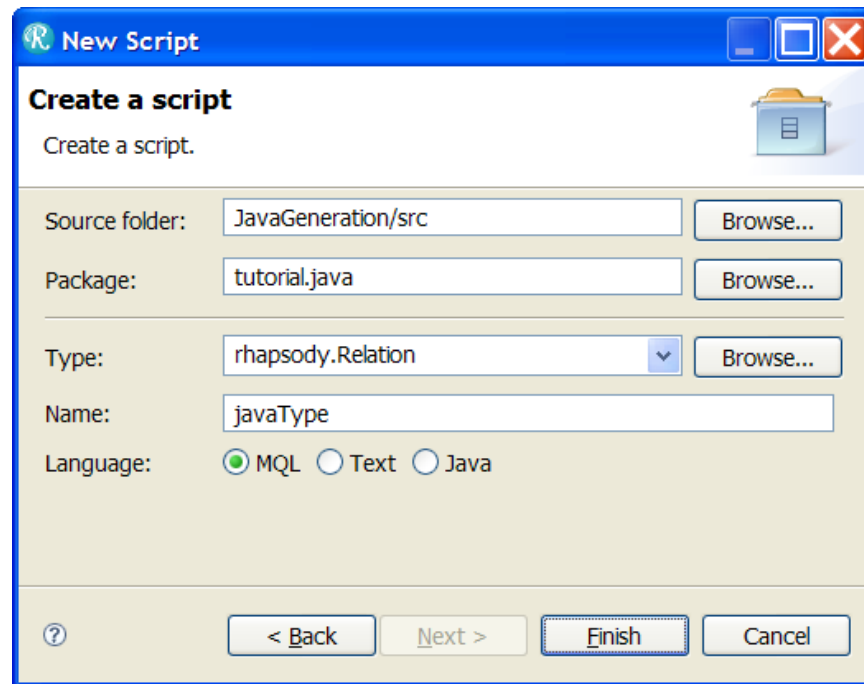
metatype rhapsody.Relation;

public script isMultivalued() : boolean {
    if (self.multiplicity == "1") {
        return false;
    } else {
        return true;
    }
}
```

The **self** variable is available in the script contents. This variable is the instance the script is evaluated on (instance of the script's metatype).

Now we will add the `javaType` script.

1. Click **File > New > Script**.
2. Type **JavaGeneration/src** in the **Source folder** field.
3. Type **tutorial.java** in the **Package** field.
4. Type **rhapsody.Relation** in the **Type** field.
5. Type **javaType** in the **Name** field.
6. Click **MQL** in the **Language** group.



7. Click **Finish**.

A new script has been created in the `rhapsody_Relation.mqs` file. Change its contents to:

```
public script javaType() : String {  
    if (self.isMultivalued()) {  
        return "java.util.Collection";  
    } else {  
        // if the otherClass is not set, returns "Object"  
        return self.otherClass.name ? "Object";  
    }  
}
```

The script behavior is the following:

1. If the `isMultivalued()` method answers true, the String `"java.util.Collection"` is returned.
2. Otherwise the expression `self.otherClass.name` is evaluated. The String `"Object"` is returned if the `otherClass` reference returns null, due to the default value expression and the null management of RulesComposer.

Now the script **javaType** can be used in a text template, as you do with any predefined feature of the type **Relation** (e.g. `${myRelation.javaType}`).

Relaunch the generation and open the file **Order.java**:

```
public class Order {  
  
    private RhpString date;  
  
    private Customer customer;  
    private java.util.Collection items;  
}
```

## Call the script from the template

We need to update the text template contents to call the newly created **javaType** script:

```
[#package tutorial.java]  
  
[#template public JavaSource(class : rhapsody.Class)]  
[#file]generated/${class.name}.java[/#file]  
public class ${class.name} {  
  
    [#-- Attributes declaration --]  
    [#foreach attr : rhapsody.Attribute in class.attributes]  
        private ${attr.type.name} ${attr.name};  
    [/#foreach]  
  
    [#-- Relations declaration --]  
    [#foreach rel : rhapsody.Relation in class.relations]  
        private ${rel.javaType} ${rel.name};  
    [/#foreach]  
}  
[/#template]
```

Relaunch the generation and open the file **Order.java**:

```
public class Order {  
  
    private String date;  
  
    private Customer customer;  
    private java.util.Collection items;  
}
```

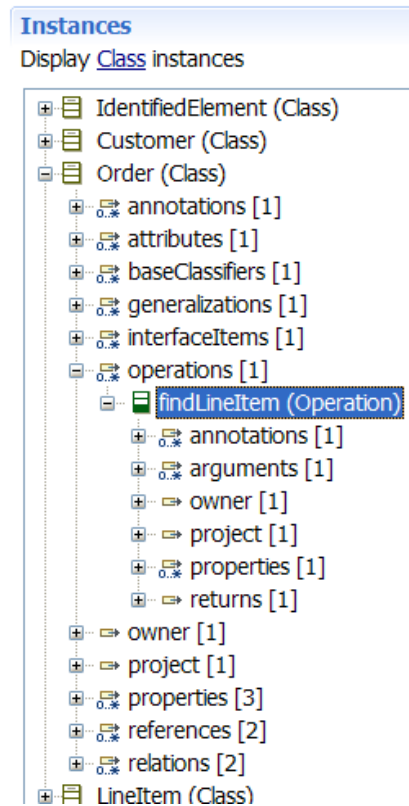
## Add methods declaration

Now we want to generate a Java method for each operation defined on a Class:

1. Read the Rational Rhapsody model using the **Rhapsody Application** reader.

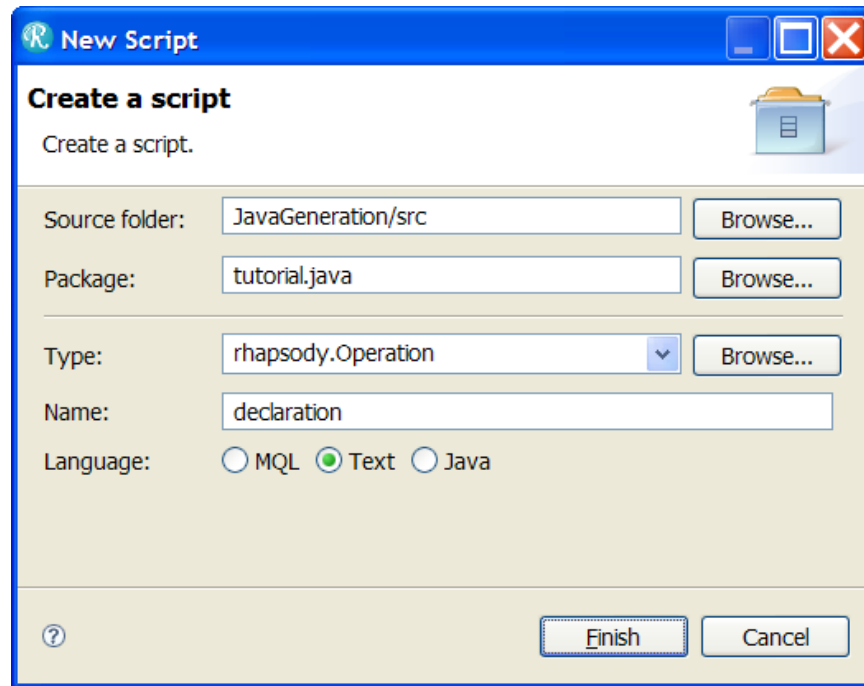
2. Select the type **Class**.
3. Select the Class **Order** and show its children.

You can see the Class `Order` has an operation `findLineItem` linked through the reference operations.



We will start by adding a declaration script on operation to generate the declaration for an operation.

1. Click **File > New > Script**.
2. Type **JavaGeneration/src** in the **Source folder** field.
3. Type **tutorial.java** in the **Package** field.
4. Type **rhapsody.Operation** in the **Type** field.
5. Type **declaration** in the **Name** field.
6. Click **Text** in the **Language** group.



7. Click **Finish**.

A file `rhapsody_operation.tgs` is created that allows us to define TGL scripts on the metatype Operation. Change its contents to:

```
[#package tutorial.java]

[#metatype rhapsody.Operation]

[#script public declaration]
    public Object ${self.name}() {
        return null;
    }

[/#script]
```

The declaration script can now be called from the main template.

```
[#package tutorial.java]

[#template public JavaSource(class : rhapsody.Class)]
[#file]generated/${class.name}.java[/#file]
public class ${class.name} {

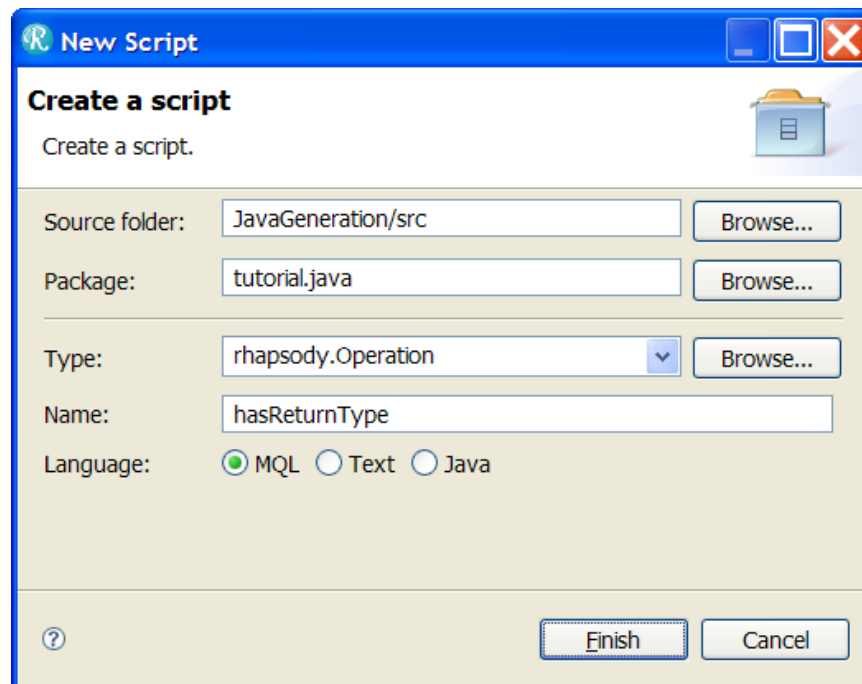
    [/-- Attributes declaration --]
    [#foreach attr : rhapsody.Attribute in class.attributes]
        private ${attr.type.name} ${attr.name};
    [/#foreach]
```

```
[#-- Relations declaration --]
[#foreach rel : rhapsody.Relation in class.relations]
    private ${rel.javaType} ${rel.name};
[/#foreach]

[#-- Operations declaration --]
[#foreach operation : rhapsody.Operation in class.operations]
    ${operation.declaration} [#trim]
[/#foreach]
}
[/#template]
```

## Add return type declaration

We will now change the return type of the generated method so that the return type is not set to always be `Object`. We will create a new MQL script on `Operation` called `returnType` to return `Object` if the operation does not have a return type, or the name of the classifier being returned. In order to facilitate the creation of this script we will first create a script called `hasReturnType` which will check if the operation returns a value.

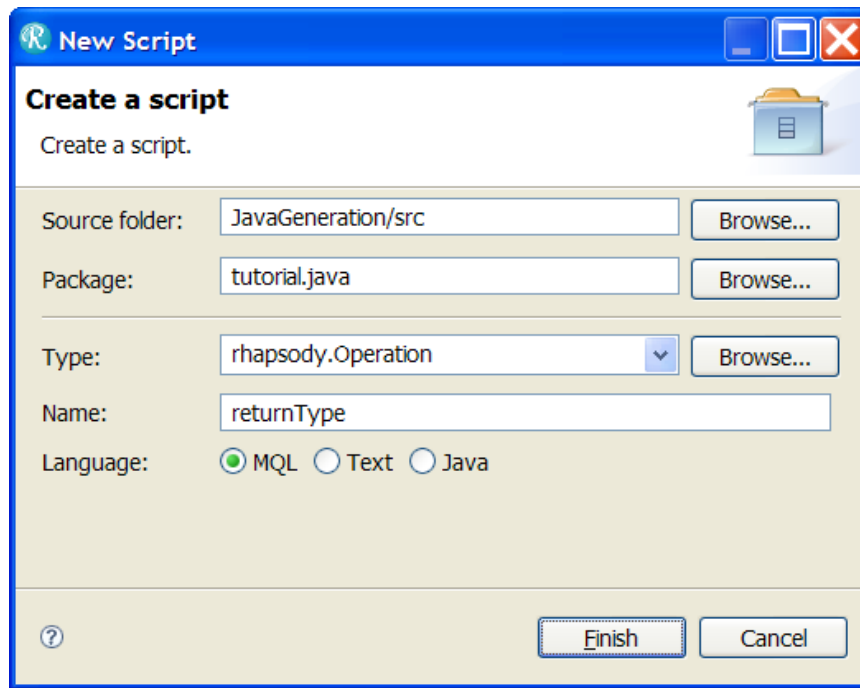


The `hasReturnType` is an MQL script which is implemented as follows.

```
public script hasReturnType() : boolean {
    return self.returns != null;
}
```



The `hasReturnType` script is now called from a new `returnType` MQL script created as shown below.



Set the contents of this script to be:

```
public script returnType() : String {  
    if (self.hasReturnType()) {  
        return self.returns.name;  
    } else {  
        return "Object";  
    }  
}
```

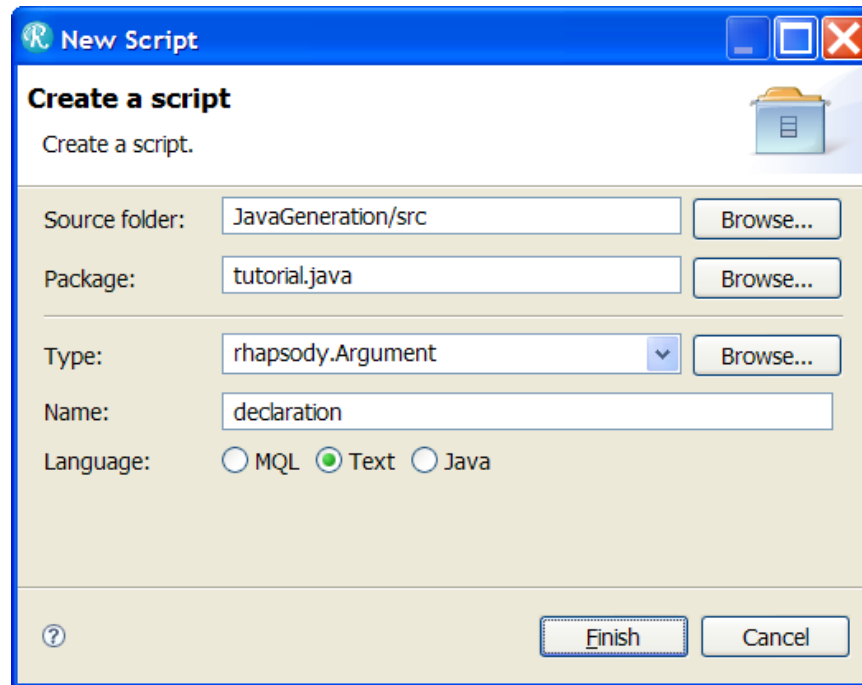
The `returnType` script is now called from the declaration script on `Operation`.

```
[#package tutorial.java]  
  
[#metatype rhapsody.Operation]  
  
[#script public declaration  
    public ${self.returnType} ${self.name}() {  
        return null;  
    }  
[/#script]
```

## Add arguments declaration

Now we want to handle the arguments of the operations.

Arguments are linked to their owner operation through the reference arguments. An Argument has an attribute `name` and a reference to its type. Create a declaration script as a Text script that returns the name of the type of the argument and the name of the argument.



Change its contents to:

```
[#package tutorial.java]

[#metatype rhapsody.Argument]

[#script public declaration]
${self.type.name} ${self.name} [#rtrim]
[/#script]
```

The `rtrim` directive removes the whitespace at the end of the argument declaration so that all the arguments can be printed on the same line.

The declaration script is now called from the declaration script on Operation.

```
[#package tutorial.java]

[#metatype rhapsody.Operation]

[#script public declaration]
    public ${self.returnType}
    ${self.name} (${self.arguments.concat("declaration", ", ")}) {
        return null;
    }
```

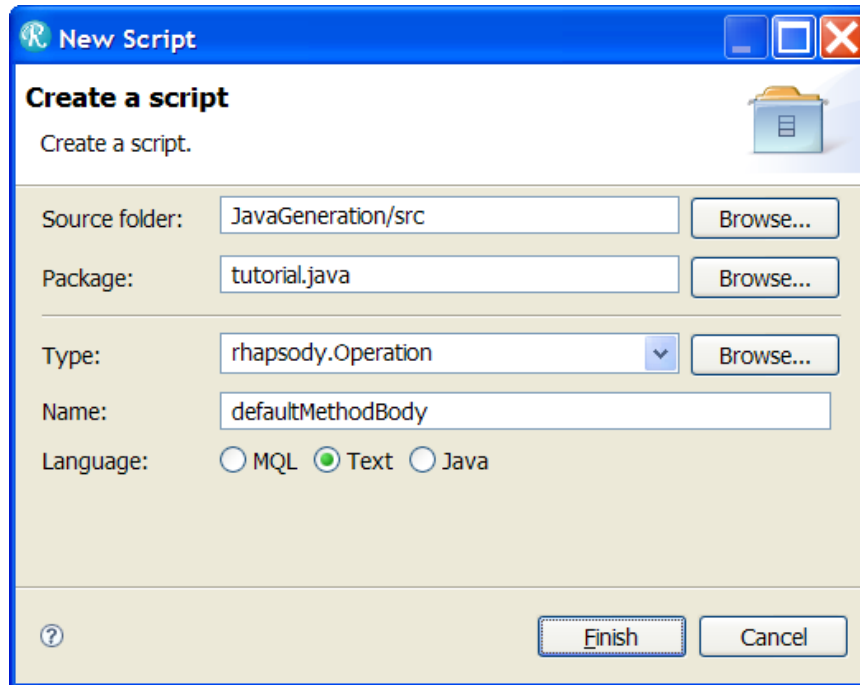
```
[/#script]
```

The `concat` operation appends the results of the `declaration` script for each of the arguments in the `arguments` collection. The `" , "` argument will insert a comma between each of the declarations in the case when more than one argument exists.

## Add operation body

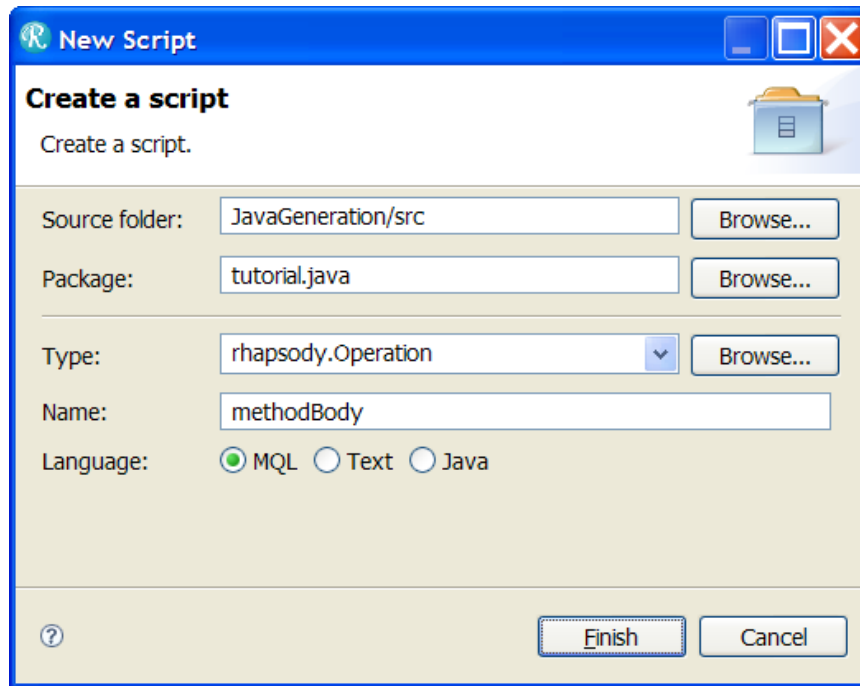
Now we will change the body of the java method so that it uses the `body` attribute of the operation. In the case where `body` is empty, we will provide a default implementation.

The first step is to define a `defaultMethodBody` TGL script on `Operation` which will return null if the operation has a return type, or return an empty string.



```
[#script public defaultMethodBody]
  [#if self.hasReturnType]
    return null;
  [/#if]
[/#script]
```

We then need to create a `methodBody` MQL Script on `Operation` that will call the `defaultMethodBody` script.



```
public script methodBody() : String {
    if (self.body.length() > 0) {
        return self.body;
    }
    else {
        return self.defaultMethodBody();
    }
}
```

The methodBody script is now called from the declaration script on Operation.

```
[#package tutorial.java]

[#metatype rhapsody.Operation]

[#script public declaration]
    public ${self.returnType} ${self.name} () {
        ${self.methodBody} [#ltrim]
    }
[/#script]
```

Relaunch the generation and open the file **Order.java**:

```
public class Order {

    private RhpString date;

    private Customer customer;
```

```

private java.util.Collection items;

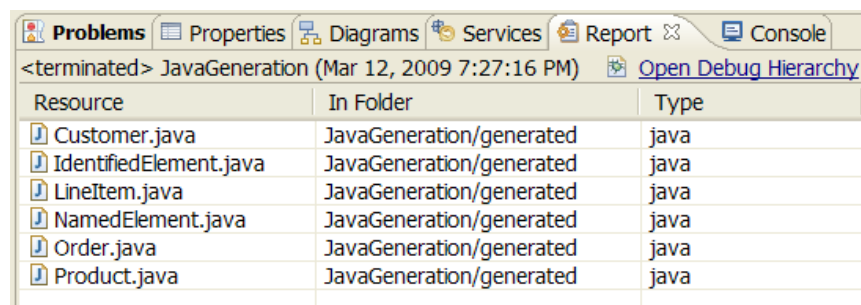
public LineItem findLineItem(RhpString productName) {
    return null;
}
}

```

## Use the debug hierarchy

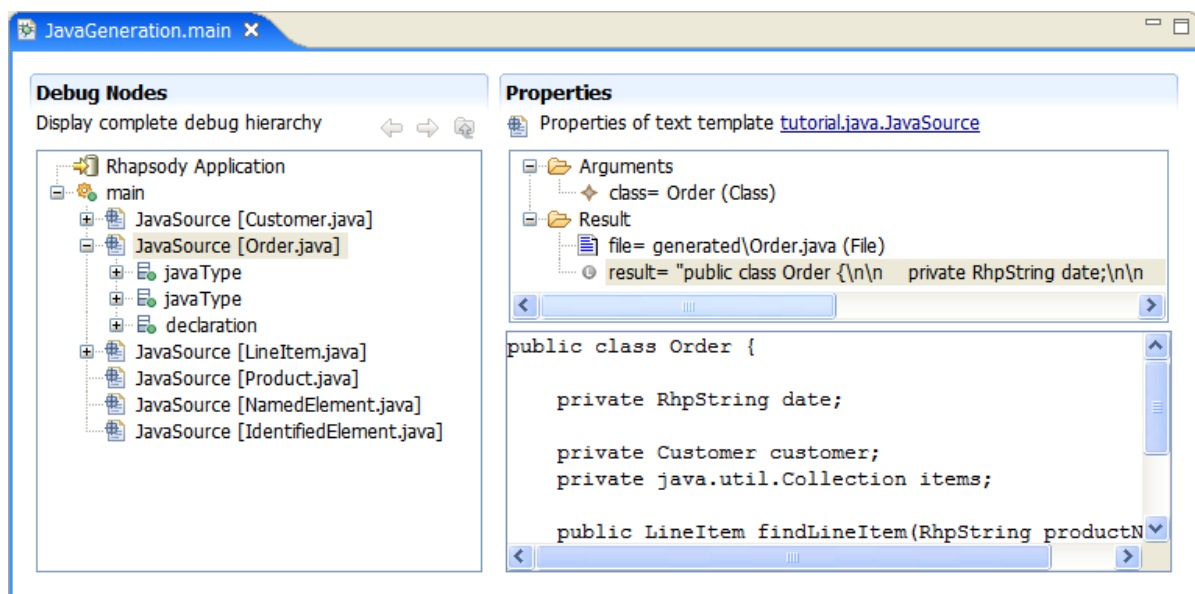
We will now use debug facilities of the RulesComposer to analyze the flow of the generation:

1. Click **Run > Debug Last Launched** to launch the generation in debug mode.
2. Show the **Report** view.
3. Right-click the file **Order.java** in this view.
4. Click **Open Debug Hierarchy**.



Resource	In Folder	Type
Customer.java	JavaGeneration/generated	java
IdentifiedElement.java	JavaGeneration/generated	java
LineItem.java	JavaGeneration/generated	java
NamedElement.java	JavaGeneration/generated	java
Order.java	JavaGeneration/generated	java
Product.java	JavaGeneration/generated	java

The debug hierarchy shows up in an editor. A debug hierarchy is a tree of nodes, each node representing an evaluated element (template, rule or script):



The screenshot shows the **JavaGeneration.main** editor with two main panels:

- Debug Nodes:** A tree view showing the hierarchy of the application. The root is **Rhapsody Application**, followed by **main**. Under **main**, there are several **JavaSource** nodes for **Customer.java**, **Order.java**, **LineItem.java**, **Product.java**, **NamedElement.java**, and **IdentifiedElement.java**. The **Order.java** node is currently selected.
- Properties:** A panel showing the properties of the selected **Order.java** node. It displays the **Arguments** (class= Order (Class)) and the **Result** (file= generated\Order.java (File)). Below this, the generated Java code is shown:
 

```

public class Order {

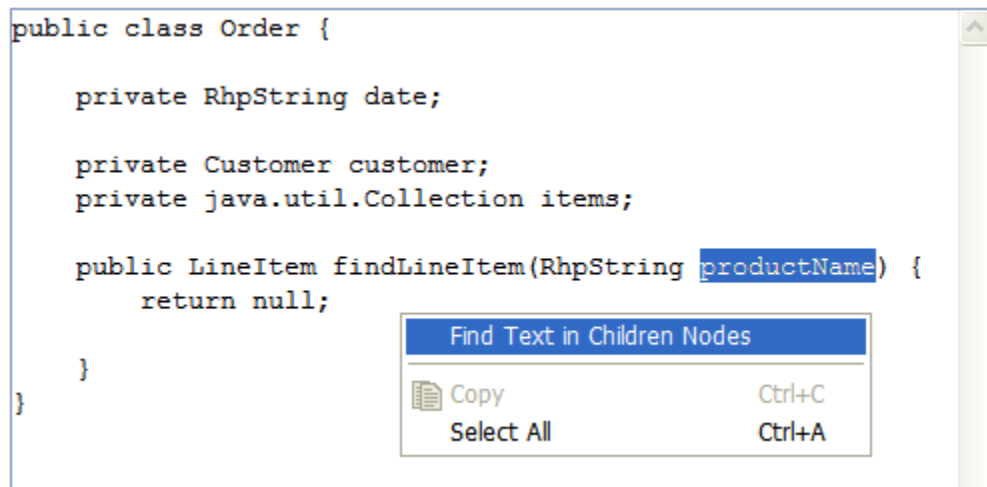
    private RhpString date;

    private Customer customer;
    private java.util.Collection items;

    public LineItem findLineItem(RhpString productN
      
```

We will now find the script that generated a part of the file `Order.java`:

1. Select the node **main\JavaSource [Order.java]** in the debug hierarchy **Debug Nodes** section.
2. Select the node **Result\result** in the **Properties** section.
3. Select the text **productName** in the details pane, bottom part of the **Properties** section.
4. Right-click to open the context menu.
5. Click **Find Text in Children Nodes**.



The debug hierarchy focus is moved to the node **main\JavaSource\declaration\declaration**. This is the script defined on **rhapsody.Argument** whose evaluation produced the text `"RhpString productName"`.

The debug hierarchy is very useful in two situations:

- You want to find the script that generated a part of a file in a complex generation.
- You want to see the flow (debug nodes) of a complex generation to understand how the generator is built.

## Include a Javadoc template

In this step, we will add a javadoc to the generated Java file, introducing template inclusion.

First create a Javadoc template:

1. Click **File > New > Text Template**.
2. Type **JavaGeneration/src** in the **Source folder** field.
3. Type **tutorial.java** in the **Package** field.
4. Type **Javadoc** in the **Name** field.
5. Click **Finish**.

Change the `Javadoc.tgt` contents to:

```
[#package tutorial.java]

[#template public Javadoc()]
/*
 * Generated by ${System.getProperty("user.name")}
 * on ${java.util.Calendar.getInstance().getTime()}
 */
[/#template]
```

Note that this template does **not** define a file directive: it means this template is not a generation entry-point, it's a fragment designed to be included in another template. This template does not expect any parameter.

This template uses the `getProperty()` method of the class `java.lang.System` to retrieve the name of the current user. It also uses the class `java.util.Calendar` to compute the current time.

We can include this Javadoc in the template **JavaSource.tgt**:

```
[#package tutorial.java]

[#template public JavaSource(class : rhapsody.Class)]
[#file]generated/${class.name}.java[/#file]
[#include Javadoc()]
public class ${class.name} {
...
}
```

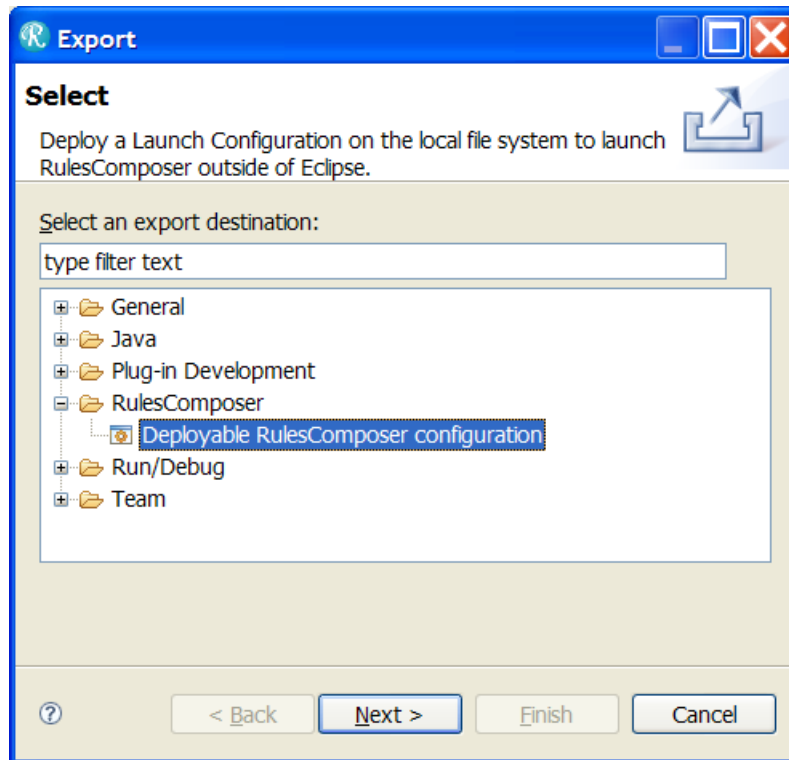
Relaunch the generation and open the file **Order.java**:

```
/*
 * Generated by IBM
 * on Thu Mar 12 19:29:12 CEST 2009
 */
public class Order {
...
}
```

## Deploy the Launch Configuration

The final step is to deploy the launch configuration that we have been executing so that Rational Rhapsody uses our generator.

1. Click **File > Export...**
2. Open the **IBM Rational Rhapsody Developer RulesComposer Add On** folder, select **Deployable RulesComposer Add On configuration**, and press **Next**.

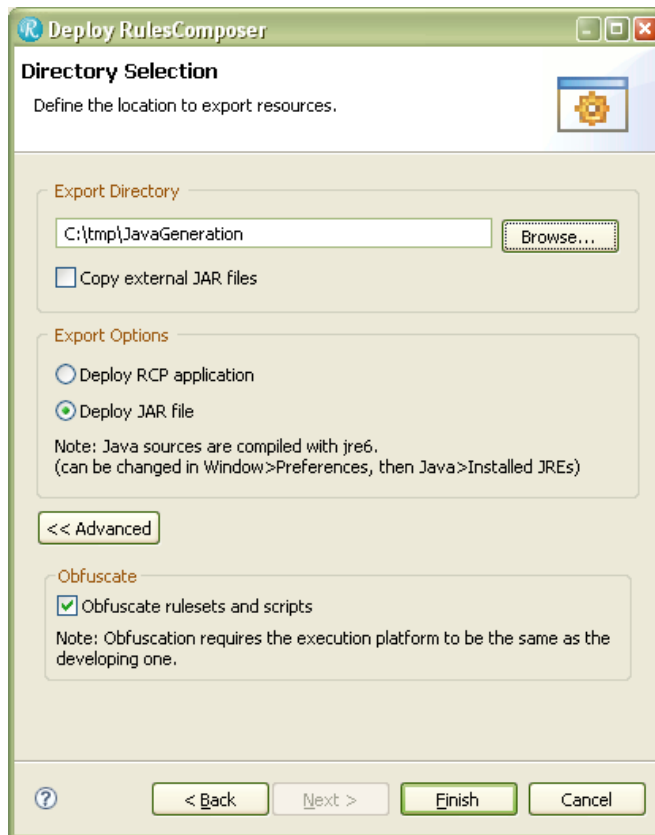


3. Choose the **JavaGeneration** launch configuration and press **Next**.





4. Enter the location where to deploy the rules, click **Deploy JAR file**, and press **Finish**.



**Note:** If you wish to encrypt .JAR files, you can press button **Advanced** and check option **Obfuscate rulesets and scripts**.

5. You can now:

1. Set the properties shown below on the Configuration in Rational Rhapsody so that our generator is used.

[-] <b>CG</b>	
[-] Configuration	
CodeGeneratorTool	External
[-] <b>JAVA_CG</b>	
[-] Configuration	
GeneratorRulesSet	c:\temp\JavaGeneration\JavaGeneration.classpath
GeneratorScenarioName	tutorial.java.JavaGeneration.main

2. Or insert a new **Helper** in Rational Rhapsody menu **Tools** using **Tools > Customize...** (see page 41).

## Use filenames associated to objects in Rational Rhapsody

In this step, we will change the Java template to generate Java files according to folders and filenames chosen by Rational Rhapsody.

For each object, Rational Rhapsody proposes two files: one for the specifications and a second for the body. Typically files with extension ADS and ADB in Ada, H and CPP in C++ and only JAVA in Java language.

To add a script:

1. Click **File > New > Script**.
2. Type **JavaGeneration/src** in the **Source folder** field.
3. Type **tutorial.java** in the **Package** field.
4. Type **rhapsody.ModelElement** in the **Type** field.
5. Type **getPropertyValueByKey** in the **Name** field.
6. Click **Java** in the **Language** group.
7. Click **Finish**.

A file `rhapsody_Class.java` is created: it allows us to define Java scripts on the metatype Class. Change its contents to:

```
package tutorial.java;

import java.util.Collection;
import java.util.Iterator;

import com.sodius.mdw.metamodel.rhapsody.Property;
import com.sodius.mdw.metamodel.rhapsody.scripts.ClassScriptContainer;

public class rhapsody_Class public class ClassScriptContainer

    public String getPropertyValueByKey(String key) {
        Collection properties = self.getProperties();
        Iterator it = properties.iterator();
        Property currentProperty = null;
        while ( it.hasNext() ) {
            currentProperty = (Property) it.next();
            if ( currentProperty != null &&
                currentProperty.getKey().equals(key) ) {
                return currentProperty.getValue();
            }
        }
        return null;
    }
}
```

The **self** variable is available in the script contents. This variable is the instance the script is evaluated on (instance of the script's metatype).

The method **getPropertyValueByKey** is able to retrieve properties associated to each object by RulesPlayer/RulesComposer.

To add a script:

8. Click **File > New > Script**.
9. Type **JavaGeneration/src** in the **Source folder** field.
10. Type **tutorial.java** in the **Package** field.
11. Type **rhapsody.Class** in the **Type** field.
12. Type **qualifiedPath** in the **Name** field.
13. Click **SQL** in the **Language** group.
14. Click **Finish**.

A file `rhapsody_Class.mqs` is created: it allows us to define SQL scripts on the metatype `Class`. Change its contents to:

```
package tutorial.java;

metatype rhapsody.Class;

public script qualifiedPath() : String {
    return self
        .getPropertyValueByKey("RHP.SpecificationFilename");
}
```

Method **qualifiedPath** is able to retrieve the specification name provided by Rational Rhapsody.

Use property **RHP.ImplementationFilename** to get body name (for C++ or Ada).

Note: property **RHP.SpecificationFilename** is set only when `Class` object should be re-generated under RulesPlayer, but always under RulesComposer.

Now we can update template **JavaSource.tgt**:

```
[#package tutorial.java]

[#template public JavaSource(class : rhapsody.Class)]
[#file]${class.qualifiedPath}[/#file]
[#include JavaDoc()]
public class ${class.name} {
    ...
}
```

Relaunch the generation and check that Java files are generated in the current configuration folder:

<ProjectDir>/<ComponentDir>/<ConfigurationDir>

Note: please find a full example implementing this solution in RulesComposer samples:

To add it to your workspace, please:

Click **File > New > Example > RulesComposer Sample > Next**

Click **Rhapsody Code Generation > C++ > Next > Finish**

## Either use “On-Demand” or “Application” model reader

Just change module **JavaGeneration.mqr**:

```
package tutorial.java;

public ruleset JavaGeneration(in model : rhapsody) {

    entry rule main() {
        // calls the proxy preloader when the connector
        // "On-Demand" is invoked.
        if (com.sodius.mdw.metamodel.rhapsody.proxy.
            ProxyPreloader.isProxyModel(model)) {
            var myProxyPreloader : com.sodius.mdw.metamodel.
                rhapsody.proxy.ProxyPreloader
            = com.sodius.mdw.metamodel.rhapsody
                .proxy.ProxyPreloader
                .getNewProxyPreloader(model);
            myProxyPreloader.preload();
        }

        foreach (class : rhapsody.Class in
                    model.getInstances("Class")) {
            $JavaSource(class);
        }
    }
}
```

If RulesComposer indicates a compilation error for the class ProxyPreloader, please open file /META-INF/MANIFEST.MF and add a dependency to:

**com.sodius.mdw.metamodel.rhapsody.proxy**

If file /META-INF/MANIFEST.MF is missed, please check if you project is an Eclipse plugin type. You can check it, using right-click on project node, and select command **PDE Tools > Convert Projects to Plug-in Projects...**

And add dependencies to:

**com.sodius.mdw.metamodel.rhapsody**  
**com.sodius.mdw.metamodel.rhapsody.proxy**  
**com.sodius.mdw.core**

Relaunch now the generation and check that generator allows indifferently **On-Demand** and **Application** model reader connector.

## Customize filenames associated to Rational Rhapsody objects

In this step, we will change the Java template to generate Java files in your own project workspace folder.

For each object, we can propose several files that Rational Rhapsody could be able to bind and edit on user demand.

Typically in Ada, we can propose four files for a **Class** object: two for specification and body, and two other to regroup nested **Port** objects.

In our tutorial, we present a case for Java language with a single filename per object.

To add a script:

1. Click **File > New > Script**.
2. Type **JavaGeneration/src** in the **Source folder** field.
3. Type **tutorial.java** in the **Package** field.
4. Type **rhapsody.ModelElement** in the **Type** field.
5. Type **rulesFileMapper** in the **Name** field.
6. Click **MQL** in the **Language** group.
7. Click **Finish**.

A file `rhapsody_ModelElement.mqs` is created: it allows us to define MQL scripts on the metatype `ModelElement`. Change its contents to:

```
package tutorial.java;

metatype rhapsody.ModelElement;

public script rulesFileMapper() : com.sodius.mdw.rhapsody.api.utils
                                .RhapsodyFileMapper {
    var mapper : com.sodius.mdw.rhapsody.api.utils
                .RhapsodyFileMapper = null;

    mapper = self.eMetamodel()
               .getModelReaderDescriptor("Rhapsody On-Demand")
               .getProperty("interfaceFileMapper");

    // Under RulesComposer, interface is inactive
    if (mapper == null) {
        return com.sodius.mdw.rhapsody.api.utils
               .NullRhapsodyFileMapper.INSTANCE;
    }
    return mapper;
}
```

Method **rulesFileMapper** retrieves Rational Rhapsody interface object to send the list of files to Rational Rhapsody application.

This interface provides several methods to send filename to Rational Rhapsody, the **filename** argument should contains only one file with its absolute path:

```
String addFileName(rhapsody.ModelElement element, String filename);
```

This can be called several times with the same element, but a different filename.

```
String addMainFileName(rhapsody.ModelElement element, String filename);
```

This can be called several times with the same element, but a different filename.

```
String addMakeFileName(rhapsody.Configuration element, String filename);
```

This should be called only one time per generation for a Rhapsody Configuration object only.

```
String addTargetFileName(rhapsody.Configuration element, String filename);
```

This should be called only one time per generation for a Rational Rhapsody Configuration object only.

To add a script:

8. Click **File > New > Script**.
9. Type **JavaGeneration/src** in the **Source folder** field.
10. Type **tutorial.java** in the **Package** field.
11. Type **rhapsody.ModelElement** in the **Type** field.
12. Type **qualifiedPath** in the **Name** field.
13. Click **SQL** in the **Language** group.
14. Click **Finish**.

This method is added to file `rhapsody_ModelElement.mqs`. Change its contents to:

```
public script qualifiedPath() : String {  
    return self.rulesFileMapper.addFileName(self,  
        "C:/workspace/Tutorial_Java/src/"+  
        self.name+".java");  
}
```

Method **qualifiedPath** computes the filename and sends it to the Rational Rhapsody interface.

Now we can update the template **JavaSource.tgt**:

```
[#package tutorial.java]

[#template public JavaSource(class : rhapsody.Class)]
[#file]${class.qualifiedPath}[/#file]
[#include JavaDoc()]
public class ${class.name} {
...
}
```

Relaunch the generation and check that Java files are generated in folder:

C:/workspace/Tutorial\_Java/src

Now, in Rational Rhapsody click the menu **File > Project properties** and select tab **Properties**.

The new file api interface is inactive under **RulesComposer**. We should deploy your project as a **launch configuration** under Rational Rhapsody to use it with the **RulesPlayer**.

For details, see chapter **Deploy the Launch Configuration**.

Change the following properties for your new launch configuration:

```
lang_CG:Configuration:CodeGeneratorTool = External
lang_CG:Configuration:ExternalGeneratorFileMappingRules = DefinedByGenerator
lang_CG:Configuration:GeneratorRulesSet = JavaGeneration.classpath
lang_CG:Configuration:GeneratorScenarioName = tutorial.java.JavaGeneration.main
```

Where *lang* is here JAVA.

Check that Rational Rhapsody displays the message “Loading external generator...”.

Now, you can execute command **Code > Re Generate > Entire Project**.

When the operation is completed, select a Class object and right-click **Edit Code**. Check that the source file is edited in folder C:/workspace/Tutorial\_Java/src.

Note: please find a full example implementing this solution in RulesComposer samples:

To add it to your workspace, please:

Click **File > New > Example > RulesComposer Sample > Next**  
Click **Rhapsody Code Generation > Java > Next > Finish**



## Launch an external ruleset

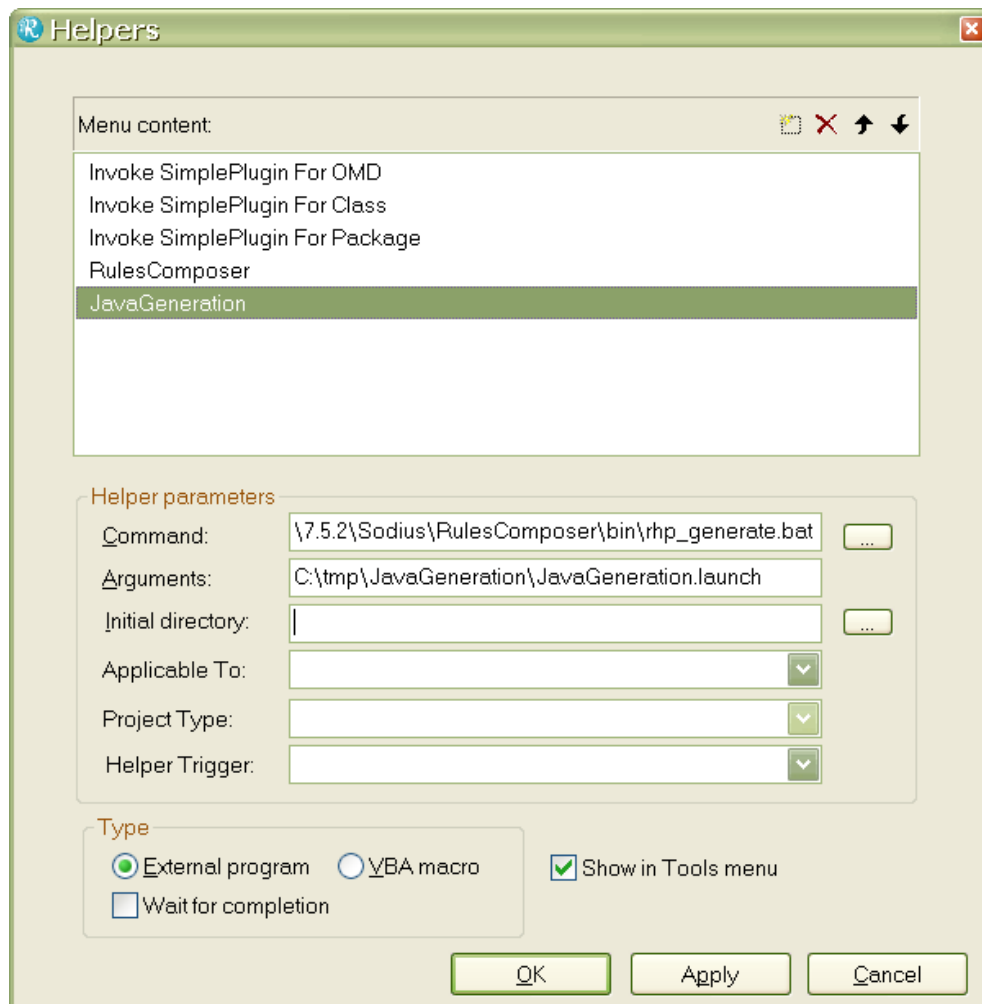
After deploying a launch configuration you wish to execute it in **IBM Rational Rhapsody** for all your own projects but you want invoke it without changing properties for each project. You will find below a way to perform this operation.

Note: you can launch rulesets in deployed mode that use only following metamodels:

- rhapsody
- matlab
- simulink
- excel

You can add as many commands as necessary rulesets in Rational Rhapsody as follows:

1. Click **Tools > Customize...**



2. Press button **New** on top of **Menu Content** box and enter for example **JavaGeneration**.
3. Choose type **External program** in **Type** box and select **Show in Tools menu** check box.
4. In **Helper parameters** box, on line **Command**, press on right button to find location of DOS script `rhg_generate.bat`.

You find this program at location:

```
<rhapsodyInstallationDirectory>\Sodius\RulesComposer\bin\rhg_generate.bat
```

5. On line **Arguments**, paste the `.launch` file associated to your own ruleset with its full folder path:  
`C:\tmp\JavaGeneration\JavaGeneration.launch`

Note: To avoid error, the field **Arguments** can be left empty. In this case Rational Rhapsody should launch ruleset declared in:

1. current project properties,
2. default language properties,
3. in file

```
<rhapsodyInstallationDirectory>\Sodius\RulesComposer\mdw.ini,
```

See values of properties **project** and **scenario**.

6. Press now button **OK** and check that command **JavaGeneration** is appeared in menu **Tools** above **Customize....**
7. You can now click on menu command **Tools > JavaGeneration**.  
 During ruleset evaluation, Rational Rhapsody opens a temporary black window but generation result is always reported in **Out** tab.

Note: To avoid error during execution, ruleset main entry must contain only one input argument of type rhapsody, example:

```
public ruleset Class2List(in rhp : rhapsody)
```

## Import, run and deploy "Rhapsody to Excel" sample

### Import this ruleset

1. Click **File > New > Example**.
2. Choose **Rulescomposer Sample** in the **New Example** window and press **Next**.
3. Expand folder **Rhapsody ruleset** in the **Import Rulescomposer Sample** window and press **Next**.
4. Choose **Rhapsody to Excel** in this folder and press **Next**.
5. In last window press **Finish**.

A project **com.sodius.mdw.rhapsody.rhapsody2excel** is created in the workspace.

This project contains in **src** folder only one package **com.sodius.mdw.rhapsody2excel** and only one file **Class2List.mqr**, please open this file.

Here is the behavior of this ruleset:

1. Code generator requires a single input argument of type `rhapsody` when we wish to deploy this ruleset:

```
public ruleset Class2List(in rhp :rhapsody)
```

2. As a second argument for output is forbidden, we should create an inner model to assume Excel transformation.

```
var exc : excel = context.getWorkbench()  
                        .getMetamodelManager()  
                        .getMetamodel("excel").createModel();
```

3. we need class `com.sodius.mdw.metamodel.rhapsody.proxy.ProxyPreloader` when we want to use instruction `getInstances("Class")` in a deployed ruleset (See [tutorial](#)):

```
var myProxyPreloader : ProxyPreloader = ProxyPreloader  
                        .getNewProxyPreloader(rhp);  
myProxyPreloader.preload();
```

4. We need to prepare main structure of Excel workbook:

```
var workbook : excel.Workbook = exc.create("Workbook");  
var sheet : excel.Sheet = exc.create("Sheet");  
workbook.sheets.add(sheet);
```

5. For each row we create only cell to insert class name:

```

var row : excel.Row = exc.create("Row");
var cell : excel.Cell = exc.create("Cell");
cell.value = class.name;
row.cells.add(cell);
sheet.rows.add(row);

```

6. Finally, we can save model in a XLS file:

```

var app : rhapsody.Application =
rhp.getInstances("Application").first();
var path : String = app.activeProject
                    .activeConfiguration.getPath();
exc.write("Excel Workbook", path + "\\output.xls");

```

## Run this ruleset

1. Click **Run > Run Configurations...**
2. Expand folder **Rulescomposer** in the **Run Configurations** window.
3. Choose **Rhp Class 2 XLS List** launch configuration in this folder and press **Run**.
4. You will find a compliant Excel file **output.xls** in the code generation folder of the Rational Rhapsody active configuration.

## Deploy this ruleset

1. Click **File > Export...**
2. Expand folder **Rulescomposer** in the **Export** window.
3. Choose **Deployable Rulescomposer configuration** and press **Next**.
4. Choose **Rhp Class 2 XLS List** launch configuration and press **Next**.
5. Select **Deploy JAR file**.
6. Fill **Export directory** with `c:\tmp` and press **Finish**.
7. Return in **IBM Rational Rhapsody** application.
8. Click **Tools > Customize...**
9. Create a new customized tool **Rhapsody to Excel** (See details page 41):  
 With Command =  
`<RhplInstallDir>\Sodius\RulesComposer\bin\rhp_generate.bat`  
 With Arguments =  
`"C:\tmp\Rhp Class 2 XLS List.launch"`
10. You can now launch command **Tools > Rhapsody to Excel**.