Generate rule artifacts with IBM ODM APIs

Bring external rules into IBM Operational Decision Manager

Vasudev Mayenkar
Pierre Berlandier

August 17, 2016

When the initial representation of business rules exists outside of IBM Operational Decision Manager (ODM), you need to consider importing an external representation of the rules so you can generate IBM ODM rule artifacts. To import to rules and turn them into IBM ODM artifacts, you can rely on a subset of the IBM ODM API for creating rule artifacts programmatically, either in the Rule Designer rule projects or in the Decision Center repository.

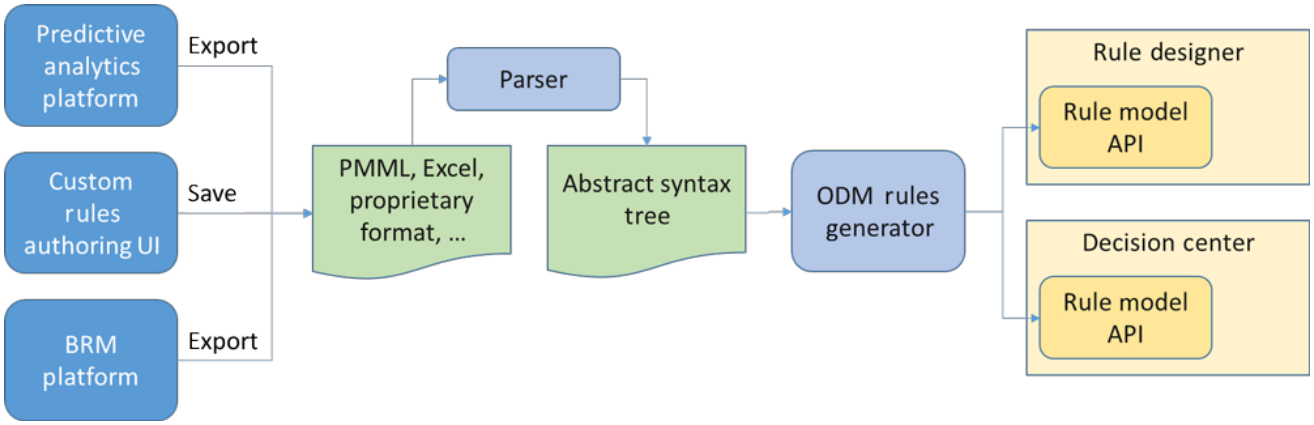
IBM® Operational Decision Manager (ODM) provides an extensive set of components to design, manage, test, and run sets of business rules, which are then organized together as services to render business decisions. For most applications, the life cycle of a rule starts in the Business console of the Decision Center, where the business users create, test, and update the rules.

However, there are times when the initial representation of the rules exists as an artifact outside of IBM ODM, generated by an external system with an arbitrary syntax. Consider the following common examples of rules that begin externally:

- Rules are generated from a predictive analytics model created by a platform such as the IBM SPSS Modeler. These rules can take the form of XML files using the Predictive Model Markup Language (PMML) format, for example. The requirement might be that rules are imported into IBM ODM (usually in the form of decision tables or decision trees) each time the model changes, and are folded into a larger set of rules that composes the decision service.
- Rules are exported from another business rules management platform, for the purpose of a migration. In this case, the rules from the external platform are likely imported once into IBM ODM, and subsequently managed and maintained with IBM ODM components.
- Rules are coming from a custom rule authoring and management component, with its own custom user interface (which could be as simple as a Microsoft Excel spreadsheet). In this case, the IBM ODM platform is only used for its rule execution capabilities, and the requirement is that the rules are imported each time they need to be deployed to the Rule Execution Server.

In all these examples, the common process is to import an external representation of the rules to generate IBM ODM rule artifacts. This import operation relies on a subset of the IBM ODM API,

which allows creating rule artifacts programmatically, either in the Rule Designer rule projects or in the Decision Center repository, as shown in the following illustration.



This tutorial shows how you can use the IBM ODM API to create rule artifacts programmatically. It provides a blueprint that you can use to model your rule importing efforts.

A word of caution is in order: Just because it is possible to create a rule artifact programmatically, it does not mean that you should use this technique in all situations. Instead, exercise proper judgment, based on the design of your application. For example, it is tempting to programmatically convert a lookup table in the database consisting of thousands of rows into a decision table using IBM ODM APIs. But that conversion is an example of poor design.

The examples in this tutorial are based on the `loanvalidation-rules` sample rule project that you can install with IBM ODM. See Setting up the Custom report generation sample. The code that is demonstrated is based on IBM ODM version 8.7.1.

This tutorial illustrates how to use IBM ODM APIs to create rule artifacts (action rules and decision tables) programmatically. You learn from examples how to create rule artifacts in both Rule Designer and Decision Center, which gives you flexibility to choose the approach that works best for your needs.

It is possible to use APIs to create rule projects and other rule project artifacts such as packages and variable sets. However, for the sake of clarity and brevity, this tutorial assumes that the required project and package structure is already in place.

A quick glance at the Eclipse Modeling Framework

Both Rule Designer and Decision Center components in IBM ODM provide an API to create and manage rule projects and the rule project artifacts that they contain. They each use a rule model that leverages the Eclipse Modeling Framework (EMF) and the Ecore meta-model.

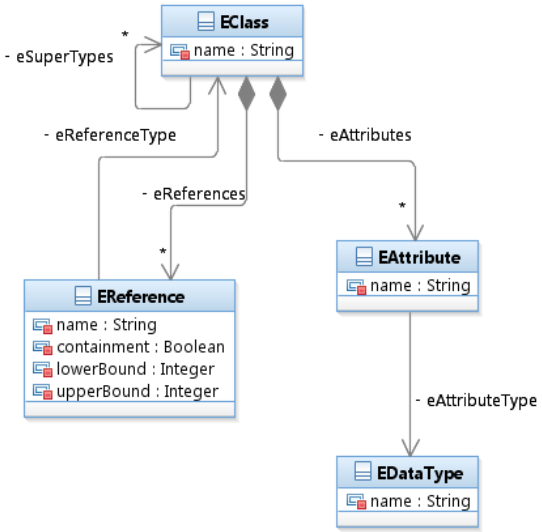
Although you do not need to know the details of EMF and Ecore to create and manipulate rule artifacts through the IBM ODM API, it helps to remember that the operations on the rule artifact instances are sometimes performed through a meta-model protocol.

EMF is an abstract language and framework for specifying, constructing and managing technology neutral models. Models are defined using the Ecore meta-model.

The following entities are key to using Ecore:

- The `EObject` interface, which is the root of all modeled objects in EMF and provides support for the behaviors and features common to all modelled objects.
- The `EClass` interface, which is used to models classes that are identified by name and can contain a number of features, for example attributes (`EAttribute`), operations (`EOperation`), references (`Ereference`).
- The `EFactory` interface, which models factories for creating instance objects.

The following figure shows the organization of key class-modeling elements of the Ecore meta-model:



For any given model, all the parts (including its classes, attributes, references) are grouped in a package of the `Epackage` type.

The rule model API for Rule Designer

The rule model API for Rule Designer makes it possible to access and manipulate rule model elements such as rules packages, action rules, technical rules, and decision tables, as well as manage their file persistence in the Eclipse workspace.

The Rule Designer API is defined by a set of classes and interfaces that are found in the `ilog.rules.studio.model` package and its sub-packages.

In addition to the data model, the API also provides notifications and services over the rule artifacts (such as search and queries, for example).

To start using the key elements of the Rule Designer API, understand the following classes and interfaces:

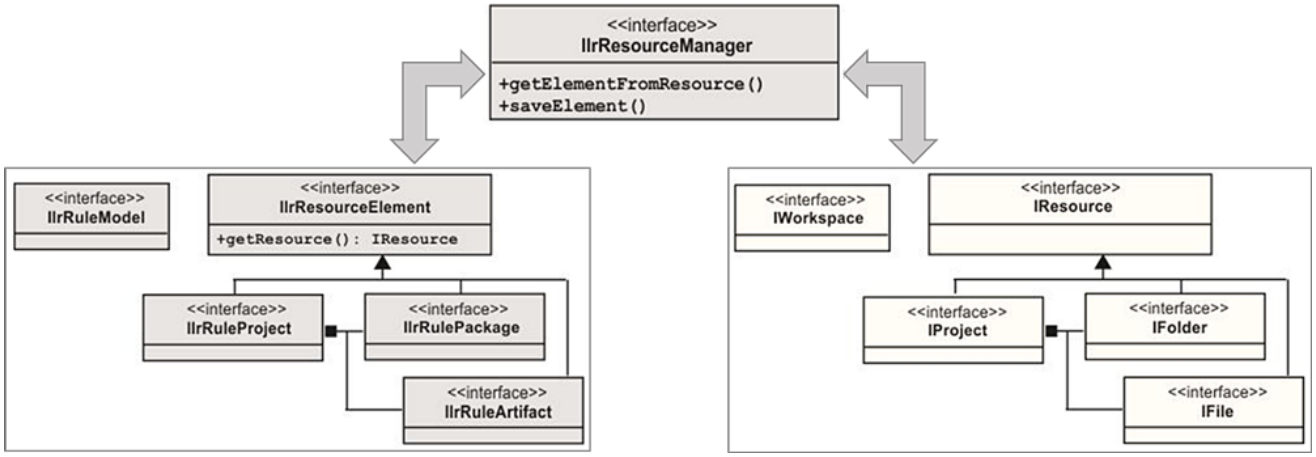
- `IlrStudioModelPlugin`: This class is the main entry point class when working with the Rule Designer API. It provides the following set of static methods to access the main model objects:
  - The Eclipse workspace singleton instance (`getWorkspace`)
  - The rule model singleton instance (`getRuleModel`)
  - The resource manager singleton instance (`getResourceManager`)
- `IlrRuleModel`: This interface, implemented by a rule model singleton class, allows selecting and retrieving a rule project to work on, using the following `getRuleProject` method:

```
// Retrieve the target rule project where the rules will be created
IlrRuleProject ruleProject =
 IlrStudioModelPlugin.getRuleModel().getRuleProject(projectName);
```

- `IlrResourceManager`: This interface manages the mapping between the rule model and the Eclipse workspace. In this interface, you use the `getElementFromResource` method to map a workspace resource to a rule model element, and the `saveElement` method to save a rule model element as a workspace resource:

```
// Persist the action rule to the workspace
IlrResourceManager resourceManager = IlrStudioModelPlugin.getResourceManager();
resourceManager.saveElement(actionRule);
```

The following illustration, based on the IBM ODM product documentation, shows how the key elements of the rule model and their associated Eclipse resources are mapped:



Rule project services

Services built on top of the Rule Designer API, such as `IlrDTService` and `IlrBRLService`, are useful for building rules. They provide functionality dedicated to management of an artifact in Rule Designer. You can obtain a handle to the service by passing a service ID to the `getService` method of `IlrRuleProject`:

```
// Get handle to IlrDTService
IlrDTService dtService =
   (IlrDTService) ruleProject.getService(IlrDTService.SERVICE_ID);
```

Rule factories

The factory classes, such as `IlrBrlFactory`, `IlrDTFactory`, and `IlrIrlFactory`, provide a create method for each non-abstract class of the rule model. The rule factory classes implement a

singleton pattern. Use the static instance named `eINSTANCE` to obtain a handle to the factory, as shown in the following example:

```
/ Create a factory for Action Rules
IlrBrlFactory factory = IlrBrlFactory.eINSTANCE;

// Create a factory for decision table or decision tree
IlrDtFactory factory = IlrDtFactory.eINSTANCE;

// Create a factory for Technical Rules
IlrIrlFactory factory = IlrIrlFactory.eINSTANCE;
```

Example: Create action rules with the Rule Designer API

Most IBM ODM decision service implementations use a mix of action rules and decision tables, so this tutorial shows examples of how to create both with the Rule Designer API.

In Rule Designer, you create an instance of an IBM ODM action rule using the `IlrBrlFactory` interface, as shown in the following example:

```
// Create a factory for Action Rules
IlrBrlFactory factory = IlrBrlFactory.eINSTANCE;

// Create an instance of action rule using the factory
IlrActionRule actionRule = factory.createActionRule();
```

After the action rule instance is available, you populate the different attributes of the rule model, such as the rule name, the rule project and the rule package to which it belongs.

Use the definition attribute to specify the body of the action rule, with a string in the form `if <conditions> then <actions>`, similar to the expression used to define the content of the Business Action Language (BAL) of action rule in the Rule Designer rule editor, as shown in the following example:

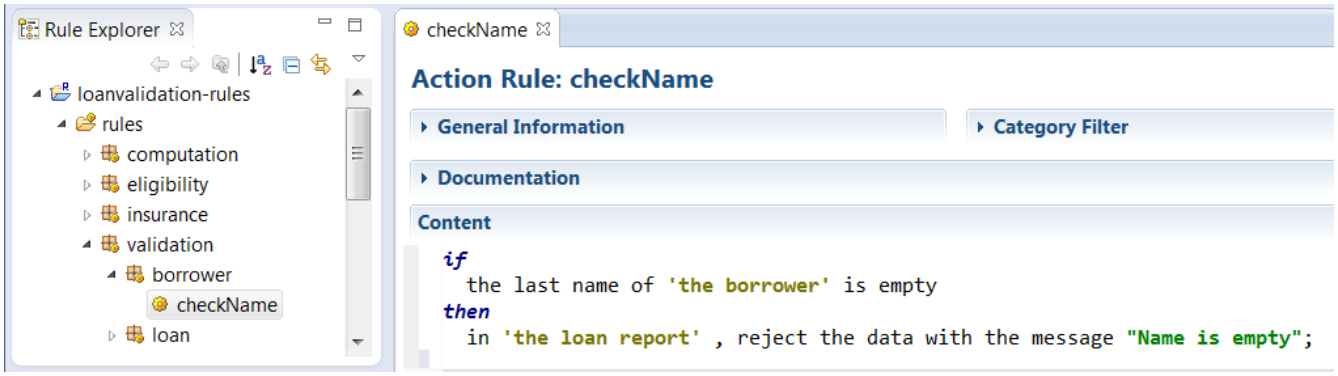
```
// Set the attributes of the model
actionRule.setName(ruleName);
actionRule.setLocale(Locale.US);
actionRule.setDefinition(ruleDefinition);
```

The following example shows the combination of the different steps to create and persist a `checkName` action rule in the `loanvalidation-rules` project, under the `validation.borrower` rule package.

```
IlrRuleModel model = IlrStudioModelPlugin.getRuleModel();
IlrRuleProject ruleProject = model.getRuleProject("loanvalidation-rules");
IlrRulePackage rulePackage = ruleProject.getRulePackage("validation.borrower");

// Create the rule instance
IlrActionRule actionRule = IlrBrlFactory.eINSTANCE.createActionRule();

// Set the rule properties.
actionRule.setName("checkName");
actionRule.setLocale(Locale.US);
actionRule.setPriority(Integer.toString(IlrPriorityValues.maximum));

// Set the rule definition.
```

```
String body =
 "if \n" +
 "  the last name of 'the borrower' is empty \n" +
 "then \n" +
 "  in 'the loan report' , reject the data with the message \"Name is empty\";";
actionRule.setDefinition(body);

// Add the rule to the rule package.
rulePackage.getFolderElements().add(actionRule);

// Persist the rule in the workspace.
IlrResourceManager resourceManager =
 IlrStudioModelPlugin.getResourceManager();
resourceManager.saveElement(actionRule);
```

The following screen capture of the Rule Designer shows the outcome of running the `checkName` action rule example code:



Example: Create decision tables with the Rule Designer API

Similar to action rules, you can create an instance of a decision table using the `IlrDtFactory` interface of the Rule Designer API, and then set the values for the basic artifact properties such as name and locale, as shown in the following example:

```
// Create the decision table object using IlrDtFactory.
// Use the createDecisionTree method to create a decision tree.
IlrDecisionTable dt = IlrDtFactory.eINSTANCE.createDecisionTable();

// Set the attributes of the model.
dt.setName(ruleName);
dt.setLocale(Locale.US);
```

So far, there are no surprises, and constructing decision tables is very similar to constructing action rules. The main difference and complexity for decision tables is the construction of the table definition using the IBM ODM decision table model.

Note that most of the Rule Designer API operations that are applicable to decision table artifacts can also be used to manipulate decision trees. However, decision trees are deprecated in IBM ODM as of version 8.8.1, so this tutorial does not focus on them.

The decision table model: IlrDTModel

To create the definition of the decision table instance, you use an instance of an `IlrDTController`. This controller is obtained through the `IlrDTService` rule project service. It controls the decision table model represented by an instance of `IlrDTModel` interface.

The `IlrDTModel` instance defines a decision table model, and provides APIs to create the structure of decision table and edit its contents. The following example shows how to obtain the `IlrDTModel` instance that is associated with an existing decision table instance.

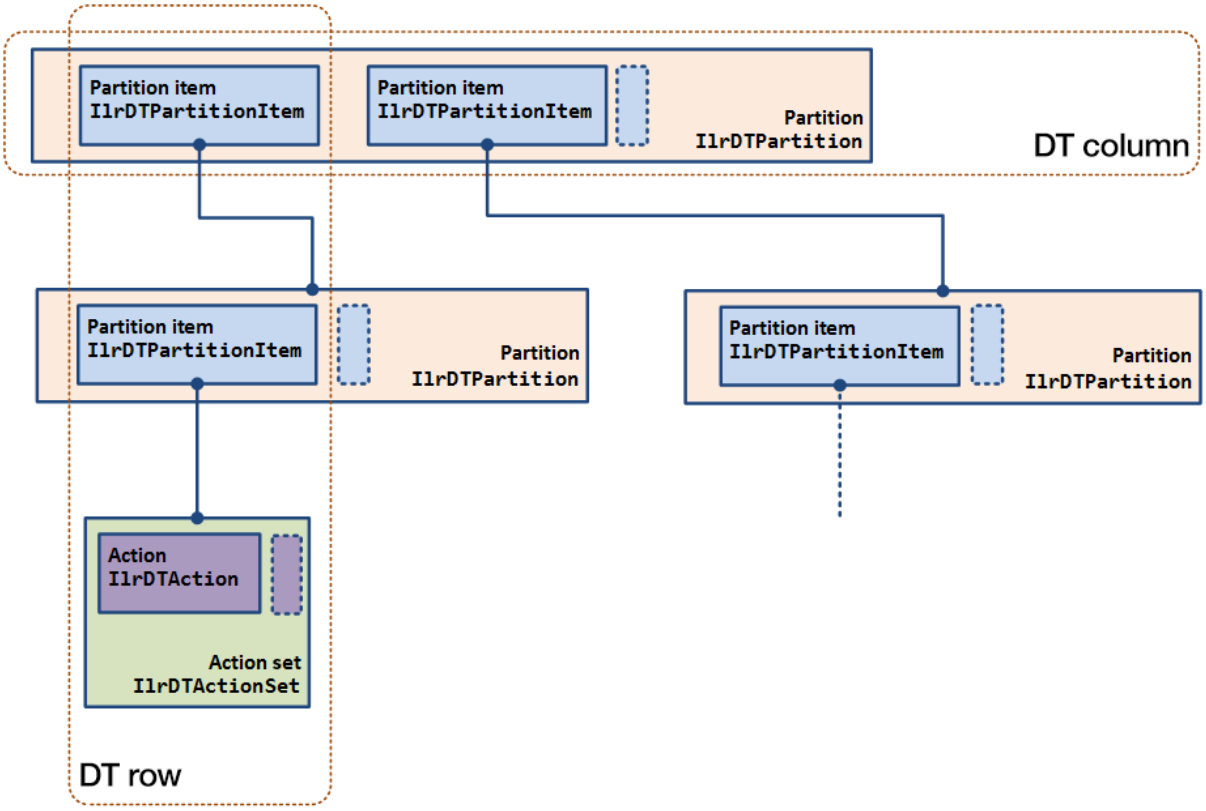
```
IlrDTService dtService =
    (IlrDTService) ruleProject.getService(IlrDTService.SERVICE_ID);
IlrDTController dtController = IlrDTHelper.createDTController(dt, dtService);
IlrDTModel dtModel = dtController.getDTModel();
```

The `IlrDTModel` interface is common between Rule Designer and Decision Center, and is used to represent a decision table. It organizes the content of the decision table as a tree of partition items. Each branch in this tree represents a row of the decision table, for example, a set of conditions, with the leaf node representing a set of actions to run if the conjunction of conditions is satisfied.

Now examine in more detail how the tree is organized for the decision table. A table representation of an `IlrDTModel` involves the following elements:

- *Partition items* (`IlrDTPartitionItem`), which represent the cells in a condition column of the table. Partition items are grouped in *partitions* (`IlrDTPartition`). All partitions have a parent partition item, which represents the cell in the preceding condition column.
- Actions are grouped in *action sets* (`IlrDTActionSet`), which hold as many *actions* (`IlrDTAction`) as the count of action columns. All action sets have a parent partition item, which represents the cell in the last condition column.
- *Partition definitions* (`IlrDTPartitionDefinition`) capture the base expression for a condition column in a decision table. The expression (a BAL statement) is held in an *expression definition* object (`IlrDTExpressionDefinition`). Expression definitions are created through an *expression manager* (`IlrDTExpressionManager`) that is obtained from the decision table model.
- Similarly, *action definitions* (`IlrDTActionDefinitions`) capture the base expression for an action column. The expression definitions for actions are also created through an expression manager.

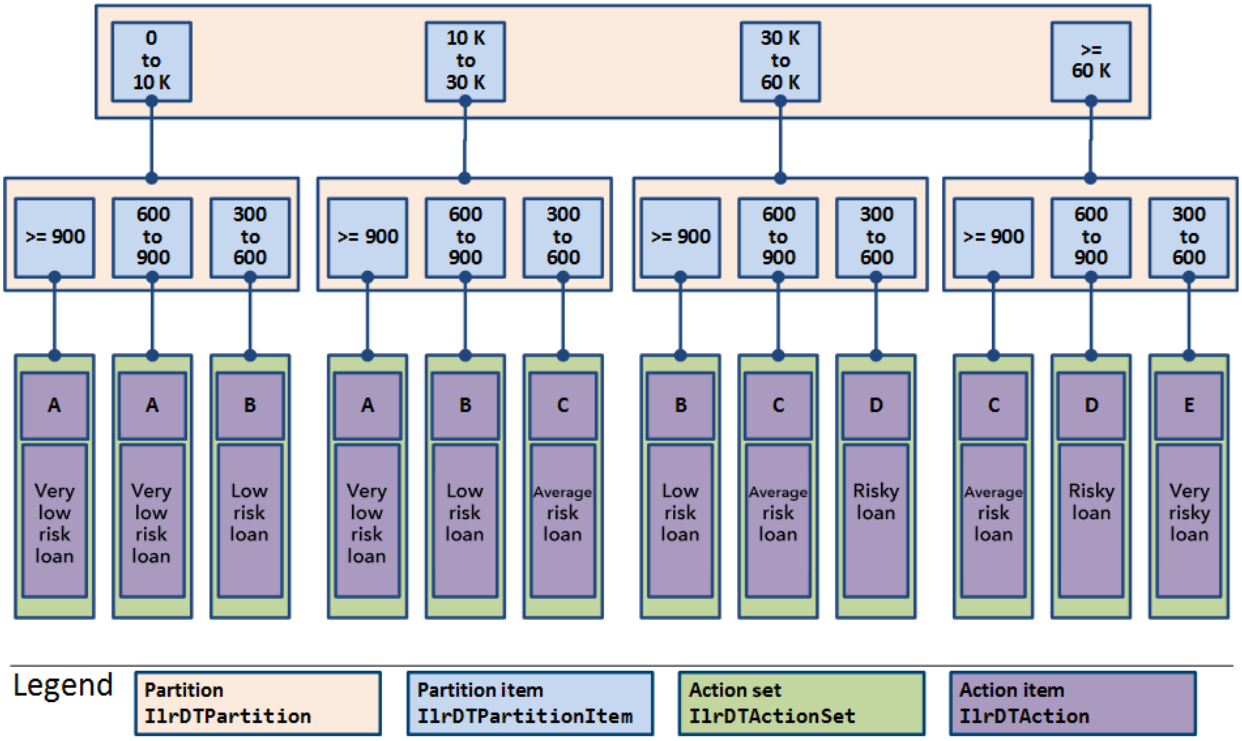
The following illustration shows the organization of the different elements that compose the `IlrDTModel` decision table model, as well as the mapping to the intuitive concept of decision table rows and columns.



Consider the organization of the model elements with the example of the Grade decision table from the `loanvalidation-rules` sample rule project. The structure and content of the Grade decision table is shown in the following screen capture:

	Yearly repayment		Corporate score		Grade	Message
	Min	Max	Min	Max		
1	0	10,000	≥ 900		A	Very low risk loan
2			600	900	A	Very low risk loan
3			300	600	B	Low risk loan
4	10,000	30,000	≥ 900		A	Very low risk loan
5			600	900	B	Low risk loan
6			300	600	C	Average risk loan
7	30,000	60,000	≥ 900		B	Low risk loan
8			600	900	C	Average risk loan
9			300	600	D	Risky loan
10	≥ 60,000		≥ 900		C	Average risk loan
11			600	900	D	Risky loan
12			300	600	E	Very risky loan

The following illustration shows the corresponding tree organization of the elements used to model this decision table:



Create decision tables with the decision table model

This section describes the steps for creating the model for a table, using the `IlrDTController` and associated classes.

When you create an `IlrDTController` for a decision table, the corresponding initial model is created with a root partition, a partition definition, a single partition item, and an action set with an empty action. The first condition column is defined simply by creating an expression definition for the column, associating it to the existing partition definition, and giving the column a title, as shown in the following example:

```
// Create the base expression for the “Yearly repayment” condition column.
String repaymentDefinitionText =
    "the yearly repayment of 'the loan' is at least <min> and less than <max>";
IlrDTExpressionDefinition repaymentExpression =
    dtModel.getExpressionManager().newExpressionDefinition(repaymentDefinitionText);

// The first partition and partition definition already exist, so they are just
// retrieved from the model, not created.
IlrDTPartitionDefinition repaymentPartitionDefinition = dtModel.getPartitionDefinition(0);

// Associate the expression definition with the partition definition
repaymentPartitionDefinition.setExpression(repaymentExpression);

// Set a title for the column (set on the partition definition)
IlrDTPropertyHelper.setDefinitionTitle(repaymentPartitionDefinition, "Yearly repayment");
```

The condition columns that come after the first one do not exist initially in the model. Create them using `newPartitionDefinition` method of the `IlrDTModel`, as shown in the following example:

```
// Create the base expression for the “Corporate score” condition column.
String scoreDefinitionText =
    "the corporate score in 'the loan report' is at least <min> and less than <max>";
IlrDTExpressionDefinition scoreExpression =
    dtModel.getExpressionManager().newExpressionDefinition(scoreDefinitionText);

// Create a new partition definition associated with the expression definition
IlrDTPartitionDefinition scorePartitionDefinition =
    dtModel.newPartitionDefinition(scoreExpression);

// Set a column title on the partition definition
IlrDTPropertyHelper.setDefinitionTitle(scorePartitionDefinition, "Corporate score");

// Add the new partition definition to the model
dtModel.addPartitionDefinition(1, scorePartitionDefinition);
```

Action columns in a decision table are represented as `IlrDTActionDefinition`. As with condition columns (partition definitions), action columns take an expression for their definition, and are assigned titles in the same way. Add the action columns to the `IlrDTModel` using the `addActionDefinition` method, as shown in the following example:

```
// Create the base expression for the “Grade” action column.
String gradeDefinitionText =
    "set the grade of 'the loan report' to a <string>";
IlrDTExpressionDefinition gradeExpression =
    dtModel.getExpressionManager().newExpressionDefinition(gradeDefinitionText);

// Create a new action definition associated with the expression definition
IlrDTActionDefinition gradeActionDefinition =
    dtModel.newActionDefinition(gradeExpression);

// Set a column title on the action definition
IlrDTPropertyHelper.setDefinitionTitle(gradeActionDefinition, "Grade");

// Add the new action definition to the model
dtModel.addActionDefinition(0, gradeActionDefinition);
```

After you define the condition and action columns, create the rows for the decision table through partition items. The partition items for the first column are stored in the *root* partition.

Each partition item has an expression instance (`IlrDTExpressionInstance`) associated with the item's column expression definition, and the expression instance contains the parameter values for that row. Create the expression instance using the `newExpressionInstance` method from `IlrDTExpressionManager`, as shown in the following example:

```
IlrDTPartition repaymentPartition = dtModel.getRoot();

// The root node of a newly created model contains a single partition item
IlrDTPartitionItem itemZeroTo10K = repaymentPartition.getPartitionItem(0);

// Prepare the parameter values for the partition item
List<String> parameters = Arrays.asList(new String[]{"0", "10000"});

// The values along with the definition are passed to the Expression
// Manager to generate the Expression Instance for this Partition Item
IlrDTExpressionInstance repaymentExpressionInstance =
    dtModel.getExpressionManager().newExpressionInstance(repaymentExpression, parameters);
itemZeroTo10K.setExpression(repaymentExpressionInstance);
```

In some cases, the expression instance should use a different statement from the one defined by the partition definition. For example, in the Grade table, the following statement from the partition definition is using two parameter values: `the yearly repayment of 'the loan' is at least <min> and less than <max>`

However, for the `">= 60,000"` cell, use the following statement instead: `the yearly repayment of 'the loan' is at least <a number>`

For these cases, when a different operator and different number of parameter values are used for a given partition item, create the expression instance with the `newOverridenExpressionInstance` method instead of `newExpressionInstance`:

```
// Prepare the parameter values for the partition item

List<String> parameters = Arrays.asList(new String[]{"60000"});

// The values along with the definition are passed to the Expression
// Manager to generate the Expression Instance for this Partition Item
IlrDTExpressionInstance repaymentExpressionInstance =
    dtModel.getExpressionManager().
        newOverriddenExpressionInstance("<a number> is at least <a number>", parameters,
 repaymentExpression);
```

To create additional partitions items, use the `addPartitionItem` method of `IlrDTModel`, as shown in the following example:

```
// Create a second partition item, after the itemZeroTo10K (i.e. position 1)
parameters = Arrays.asList(new String[]{"10000", "30000"});
repaymentExpressionInstance =
    dtModel.getExpressionManager().newExpressionInstance(repaymentExpression, parameters);

dtModel.addPartitionItem(repaymentPartition, 1, repaymentExpressionInstance);
```

As mentioned earlier, the decision table model is a tree structure, so there is only a single partition, called the root partition, for the first column or node. However, for each subsequent column, you must create a partition for each partition item from the previous column. You add partitions to each of the partition items fetched from the previous columns or node using the `addPartition` method of the `IlrDTModel`.

For example, to populate the `Corporate score` column, you need to add a new partition to each partition items of the `Yearly repayment` column. The following code snippet shows an example of adding a partition for the `Corporate score` column to the first partition item of the `Yearly repayment` column:

```
// Add a partition for Corporate score as a child of the first
// partition item of the root partition
IlrDTPartition firstScorePartition =
    dtModel.addPartition(itemZeroTo10K, scorePartitionDefinition, null);

parameters = Arrays.asList(new String[]{"900"});
IlrDTExpressionInstance scoreExpressionInstance =
    dtModel.getExpressionManager().
        newOverriddenExpressionInstance("<a number> is at least <a number>", parameters, scoreExpression);

dtModel.addPartitionItem(firstScorePartition, 0, scoreExpressionInstance);
```

Finally, to define actions, you retrieve the partition items for the condition column immediately before the first action column and assign actions to each partition item. Each cell in the action column is represented by an instance of `IlrDTActionSet`.

The `getStatement` method returns the next branch in the tree, which is the next condition after this partition item, if it exists. If there are no further partitions, it returns the action set. Populate the action set with actions (`IlrDTAction`) that correspond to each action column in the decision table, as shown in the following example:

```
parameters = Arrays.asList(new String[]{"\"A\""});
IlrDTExpressionInstance gradeExpressionInstance =
    dtModel.getExpressionManager().newExpressionInstance(gradeExpression, parameters);

IlrDTActionSet actionSet =
    (IlrDTActionSet) firstScorePartition.getPartitionItem(0).getStatement();
actionSet.addAction(0, gradeActionDefinition, gradeExpressionInstance);
```

After you populate all columns in the decision table model, persist the decision table in the workspace using a `IlrResourceManager` instance, in the same way you did for action rules:

```
// Persist the decision table in the workspace
IlrResourceManager resourceManager = IlrStudioModelPlugin.getResourceManager();
resourceManager.saveElement(dt);
```

The rule model API for Decision Center

The IBM ODM Decision Center application is a rule management server and repository integrated into a Java EE application server. You use it to author, manage, validate, and deploy business rules. The Decision Center API allows you to connect to a Decision Center instance using credentials of a registered user, to manage projects and baselines, to create packages and rules, to search the repository, and to lock, unlock, and commit project elements.

To create rule artifacts through the Decision Center API, get acquainted with the following three key interfaces:

- `IlrSession`, which represents a connection to the Decision Center instance,
- `IlrSessionHelper`, which provides a set of helper methods to create and manipulate rule artifacts,
- `IlrBrmPackage`, which defines the collection of all the parts that make up the rules model.

The Decision Center session

Before you can start any interaction with Decision Center, you need to establish a remote session to it, by creating an instance of `IlrSession`. You use an `IlrSessionFactory` instance to connect to the Decision Center and get a handle to an `IlrSession` instance. `IlrSession` objects are stateful and are associated with the user whose credentials are supplied to the connect method, as shown in the following example:

```
String serverUrl = "http://localhost:9080/teamserver";
String datasource = "jdbc/ilogDataSource";
String login = "rtsAdmin";
String password = "rtsAdmin";

// IlrRemoteSessionFactory is an implementation of IlrSessionFactory
// that can be used to connect to Decision Center through a client.
IlrSessionFactory factory = new IlrRemoteSessionFactory();
factory.connect(login, password, serverUrl, datasource);

// Use the IlrSessionFactory instance to connect to Decision Center
// and get handle to IlrSession instance
IlrSession session = factory.getSession();
```

Decision Center session helper methods

You can use the `IlrSession` instance to create rule artifacts. However, the helper class `IlrSessionHelper` provides a set of static methods that make it easier to create and manipulate rule project artifacts such as rule packages, action rules, decision tables, and variable sets.

The session helper is also useful to access projects and project baselines from the Decision Center repository. In particular, after you successfully create a connection to the Decision Center, the next step is usually to set up the project and the baseline you work on throughout the session, as shown in the following code:

```
// Retrieve the rule project by name
IlrRuleProject ruleProject =
    (IlrRuleProject) IlrSessionHelper.getProjectNamed(session, "loanvalidation-rules");

// Set the sessions’s working baseline to the current baseline
IlrBaseline currentBaseline =
    IlrSessionHelper.getCurrentBaseline(session, ruleProject);
session.setWorkingBaseline(currentBaseline);
```

The rule model package

The `IlrBrmPackage` interface is the EMF package for the Decision Center business rules model. It contains accessors for meta-objects to represent each class, feature, enumerations and data types that support the definition of business rules artifacts. You frequently use this package to retrieve the following information, for example:

- The EMF `EClass` of the rule artifact that you want to create. For example, the `getActionRule` method of `IlrBrmPackage` returns the `EClass` instance associated with action rule.
- The EMF `EAttribute` of a rule artifact class that you want to get or set. For example, the `getRule_Priority` method of `IlrBrmPackage` returns the `EAttribute` instance associated with the priority property of rules.

Example: Create action rules with the Decision Center API

Most IBM ODM decision service implementations use a mix of action rules and decision tables, so this tutorial shows examples of how to create both with the Decision Center API with the Decision Center API.

You can create a rule artifact and commit it to the working baseline with the `createRule` method of the session helper. You specify the type of the rule artifact by providing its `EClass`, which is obtained from the `IlrBrmPackage`.

The following example shows how to create an action rule by selecting the action rule `EClass` from the BRM package using the `getActionRule` method. If you want to create a technical rule instead, use the `getTechnicalRule` method to look up and use the `Eclass` technical rule in the package:

```
// Get handle to the package for the Decision Center rule model
IlrBrmPackage model = session.getBrmPackage();

// Create an action rule instance
IlrActionRule actionRule =
    (IlrActionRule)IlrSessionHelper.createRule(session, model.getActionRule(), rulePackage, ruleName,
 ruleDefinition);
```

Note that the helper class also provides a dedicated `createActionRule` method that you can use to directly create an action rule.

To set the value of structural features (`EAttribute` or `EReference`) for a rule artifact, you use the `setRawValue` method of the `IlrElementDetails` interface.

If the target feature is part of the base rule model, you can obtain the feature from the BRM package. For example, the following code snippet shows how to set the value of the *priority* feature of an action rule:

```
IlrBrmPackage model = session.getBrmPackage();
String priorityValue = Integer.toString(IlrPriorityValues.maximum);
actionRule.setRawValue(model.getRule_Priority(), priorityValue);
```

If the target feature is part of the rule model extension (custom properties), you can obtain the feature through the `IlrModelInfo` instance, which contains information about the overall Decision Center model, particularly the rule model extension.

The following example shows how to set the value of a property that is part of a rule model extension:

```
IlrModelInfo modelInfo = session.getModelInfo();
EAttribute e = (EAttribute) modelInfo.getElementFromFQN(propertyName);
actionRule.setRawValue(e, propertyValue);
```

The fully qualified name of a rule property is usually prefixed by `brm.BusinessRule`. For example, the fully qualified name of the effective date property is `brm.BusinessRule.effectiveDate`.

Example: Create decision tables with the Decision Center API

The process of creating decision tables with the Decision Center API is similar to creating action rules: use the `createRule` method, with the appropriate `EClass` obtained from the BRM package (using either `getDecisionTable` or `getDecisionTree` method), as shown in the following example. If no definition is provided (`null` value), a default empty table or tree is generated.

```
// Get handle to the package for the Decision Center rule model
IlrBrmPackage model = session.getBrmPackage();

// Create a decision table instance
IlrDecisionTable dTable =
   (IlrDecisionTable) IlrSessionHelper.createRule(session, model.getDecisionTable(), rulePackage, ruleName,
 null);
```

After the table is created, populate it in the same way as you did for the Rule Designer API, using the `IlrDTModel` and `IlrDTController` API. (As mentioned earlier, the `IlrDTModel` is common between Rule Designer and Decision Center models.) Obtain the controller and associated model, as shown in the following example:

```
IlrDTController dtController = IlrSessionHelper.getDTController(session, dTable, Locale.US);
IlrDTModel dtModel = dtController.getDTModel();
```

Note that the default decision table generated using the session helper contains empty rows, condition columns, and action columns. (Similarly, the default decision tree contains empty nodes.) You need to remove these columns and rows before populating the decision table model.

After the definition of the decision table model is complete, assign it to a decision table using the `setDefinition` method of the session helper:

```
String dtBody = IlrSessionHelper.dtControllerToStorableString(session, dtController);
IlrSessionHelper.setDefinition(session, dTable, dtBody);
```

Finally, you commit the decision table to the Decision Center repository using the `commit` method from the `IlrSession` class:

```
session.commit(dTable);
```

Importing external rule artifacts

Because the focus of this tutorial is using IBM ODM APIs, the examples intentionally keep the source of imported rules very simple. If the source rules to be imported are expressed using an arbitrary syntax, first convert them into an intermediate model using a parser, and then generate IBM ODM rule artifacts (using the Rule Designer API or the Decision Center API described in this tutorial) from the intermediate model. The intermediate model tends to differ from project to project. Therefore, the examples avoid using an intermediate model.

To convert a set of source rules expressed in an arbitrary syntax into an intermediate model, you can build a parser, for example, with the ANTLR framework. ANTLR is a powerful parser generator for reading, processing, running, or translating structured text. This framework allows

you to generate a parser based on a grammar for the parsed entities. From the grammar, ANTLR produces Java Lexer and Parser classes, and a listener class to listen to the traversal of the entities as each grammar rule is used.

Label the rules with the `#` operator (such as `#DecisionTable`). Generate two methods in the listener class: one for entering the grammar rule, and one for exiting the grammar rule:

```
public void enterDecisionTable(SourceRulesParser.DecisionTableContext ctx)
public void exitDecisionTable(SourceRulesParser.DecisionTableContext ctx)
```

To build instances of an intermediate model that represents an abstract syntax tree of the artifacts that are parsed, implement a subclass of the base listener class. This way, you can use the ANTLR framework to convert rules that are defined with an arbitrary syntax into an intermediate model.

Finally, generate IBM ODM rule artifacts from the intermediate model, using the APIs described in this tutorial.

Conclusion

This tutorial described in brief the meta-model for IBM ODM rules that is based on Eclipse Modeling Framework. It covered rule model APIs for both Rule Designer and Decision Center and illustrated how to use the APIs to generate IBM ODM rule artifacts programmatically. The tutorial also suggested an approach for importing rule artifacts with an arbitrary syntax into IBM ODM as rule artifacts through an intermediate model. It introduced ANTLR as a framework to use to generate the intermediate model.

You can download the sample code used in the examples in this tutorial from GitHub at vmayenkar/ruleimporterworkspace and try it out yourself. If your organization has a use case for rules artifact generation, you can use the knowledge you gained in this tutorial, with further help from the IBM ODM product documentation, to programmatically generate rule artifacts for your applications.

Acknowledgements

The authors would like to thank Matt Voss and Uzma Siddiqui for their careful review of this tutorial and their insightful suggestions.

Related topics

- Eclipse Modeling Framework (EMF)
- ANTLR framework
- Sample: Create a decision table using APIs