



OPENRULES® DECISION MANAGER

User Manual for Business Analysts

How to Create, Test, and Deploy Operational Business Decision Models

OpenRules, Inc.

www.OpenRulesDecisionManager.com

November-2020

Table of Contents

Introduction	5
Installation	5
Sample Decision Models	6
Introductory Decision Model "Vacation Days"	7
Business Logic	7
Glossary	9
Environment	11
Test Cases	12
Building and Testing Decision Model	13
File "project.properties"	13
File "test.bat"	14
Testing Results	14
Explanations	15
Decision Modeling Approach	16
What is Decision Model	16
Goal-Oriented Decision Modeling	17
Decision Tables	18
Decision Table Structure	18
Execution Logic	20
Single-Hit Decision Tables	21
Multi-Hit Decision Tables	22
Sequential Decision Tables	24
Conditions	25
Comparing Strings	25
Comparing Numbers	26
Using Natural Language Inside Decision Tables	28

Comparing Dates	29
Comparing Boolean Values	30
Other Condition Types	31
Conditions on Collections	32
Conclusions	34
Simple Conclusions/Actions	34
Conclusions on Collections	36
Displaying Messages	37
Expressions	38
Formulas.....	38
Java Snippets	41
Dealing with Dates	43
Iteration over Collections of Objects	45
Sorting Collections of Objects	47
Glossary	49
Standard Glossary	49
Multiple Glossaries.....	50
Optional Glossary Columns	51
Decision Model Testing	52
Building Test Cases.....	52
Data Tables	52
Test Cases.....	53
More Complex Test Cases	54
Building and Running Decision Model	56
Configuration File “project.properties”	56
Build and Run	57
Error Reporting.....	57

Testing Decision Model.....	58
Debugging Decision Model	59
Decision Model Deployment	63
Decision Model Execution Using Java API.....	63
Decision Model as an AWS Lambda Function.....	64
Decision Model as a RESTful Web Service	66
Packaging Decision Models as a Docker Image.....	68
Rules-based Service Orchestration.....	70
Technical Support	73

INTRODUCTION

OpenRules® Decision Manager allows business analysts to develop, test, deploy, and continue to maintain operational business decision models. Using only familiar tools (MS Excel® or Google Sheets® and a file manager like Windows File Explorer), business analysts can:

- **Create business decision models** in Excel files using decision tables and other standard decisioning constructs to represent sophisticated business decision logic
- **Test/Debug/Execute decision models** and **Analyze** the produced results
- **Deploy decision models** as decision microservices on-cloud or on-premise.

This guide explains how you, as a business analyst, can do it yourself before involving software developers for the integration of your tested decision models with IT systems.

INSTALLATION

We will assume that you've already installed Java Development Kit, version 1.8 (or higher) and MS Excel (or Google Sheets). After you [download](#) OpenRules® Decision Manager, you will have one zip-file “OpenRulesDecisionManager_8.x.y.zip”. Unzip this file to your hard drive, and you will see the folder “**openrules.install**”. We call this folder a workspace as it contains everything you need to work with OpenRules Decision Manager.

To complete the installation, open the sub-folder “openrules.config” and double-click on “install.bat” (or run “install” if you use Unix or Mac). Make sure that you have an internet connection during this installation process and you haven't received any “red” messages during the installation. The execution protocol (a black screen) will show many installation commands and should end up with “BUILD SUCCESS” like on the following screen:

```
[INFO] --- maven-install-plugin:2.4:install (default-install) @ openrules-config ---
[INFO] Installing C:\_GitHub\openrules.dm.models\openrules.config\pom.xml to C:\Users\jacob\.m2
\repository\com\openrules\openrules-config\8.0.3-SNAPSHOT\openrules-config-8.0.3-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 5.154 s
[INFO] Finished at: 2019-12-08T13:37:50-05:00
[INFO] -----
Press any key to continue . . .
```

The workspace “openrules.install” contains several samples of decision projects in separate folders - you may explore them with the standard File Manager or any IDE such as [Eclipse](#).

Many decision projects are similar to the very simple project “Hello”. It keeps business rules in the file “Rules.xls” and test-cases in the file “Test.xls”. You may use “test.bat” to build and execute these decision models.

SAMPLE DECISION MODELS

You also may [download](#) the orkspace “openrules.samples” that includes many decision models, such as “[Hello](#)“, “[VacationDays](#)“, “[UpSellRules](#)“, “[PatientTherapy](#)“, and others, which are ready to be built, tested, and deployed. Let’s look inside a typical decision model such as “[VacationDays](#)”.

While it contains many sub-folders and files, you will be interested only in the sub-folder “**rules**” that we usually call “Rules Repository”. It contains the following Excel files:

- **DecisionModel.xls** with the Environment table that refers to all Excel files that compose this decision model;
- **Glossary.xls** with the table Glossary that describes all decision variables used by this decision model;
- **Rules.xls** with decision tables that implement business logic;
- **Test.xls** with tables that describe test cases.

If you double-click on the file “*VacationDays/run.bat*” it will build and execute this decision model. Let’s look at how this decision model is organized.

INTRODUCTORY DECISION MODEL "VACATION DAYS"

This decision model specifies decision logic for assigning vacation days to an employee based on his/her age and years of service. Here are the business rules:

- *Every employee receives at least **22** vacation days. Additional days are provided depending on age and years of service:*
- *Only employees younger than 18 or at least 60 years, or employees with at least 30 years of service will receive an **extra 5 days**;*
- *Employees with at least 30 years of service and also employees of age 60 or more receive an **extra 3 days**, on top of possible additional days already given;*
- *If an employee has at least 15 but less than 30 years of service, an **extra 2 days** are given. These 2 days are also provided for employees of age 45 or more. These **extra 2 days cannot be combined with an extra 5 days**.*

Business Logic

The ultimate goal of this decision model is to determine the value of the decision variable "**Vacation Days**" that we will refer to as our main goal. If you open the file "*VacationDays/rules/Rules.xls*" in Excel, you will see that the business logic for this goal has been presented in the following decision table "CalculateVacationDays":

DecisionTableMultiHit CalculateVacationDays				
If	If	If	Conclusion	
Eligible for Extra 5 Days	Eligible for Extra 3 Days	Eligible for Extra 2 Days	Vacation Days	
			=	22
TRUE			+	5
	TRUE		+	3
FALSE		TRUE	+	2

This is the first example of a decision table presented in Excel using the standard OpenRules format. The first row (with a black background and white foreground) is called a signature row. Every OpenRules decision table in the left top corner contains a keyword such as DecisionTable, Glossary, DecisionTest, etc. This table starts with the

keyword **“DecisionTableMultiHit”**. It tells us that this is a so-called “multi-hit decision table” that executes all (!) satisfied rules (while the default “single-hit” table stops after executing the first hit rule).

The second word in the signature row “CalculateVacationDays” specifies the name of the decision table that should be unique, start with a letter, and not include whitespaces.

The second row specifies different conditions and actions (conclusions). This table contains 3 conditions (specified by the keyword “If”) and one conclusion (the keyword “Conclusion”). The third row contains the names of decision variables used by conditions and actions. The variable name can use spaces and should clearly explain the business meaning of these variables.

The very first rule in the 4th row assigns 22 days to the variable “Vacation Days” (unconditionally). Empty cells indicate that the proper condition is not applicable and you can use hyphen “-” instead of leaving the cells empty. The rules in rows 5, 6, and 7 may add (or not) extra 5, 3, or 2 days when an employee is eligible for them. Looking at the last rule, you will see how this decision table takes care of the rule *“An extra 2 days cannot be combined with an extra 5 days.”* Hopefully, this decision table is intuitive enough to represent our top-level decision logic.

Note that all columns in the first (signature) row be merged to indicate the end of the table. Usually, all decision tables should be surrounded by empty cells, especially there should be an empty row at the end of any decision table.

Now let’s look at how the eligibility logic for extra days is defined. The same Excel workbook “*VacationDays/rules/Rules.xls*” contains another decision table (in the second worksheet) that specifies decision logic for the decision variable (sub-goal) **“Eligible for Extra 5 Days”**:

DecisionTable SetEligibleForExtra5Days		
If	If	Then
Age in Years	Years of Service	Eligible for Extra 5 Days
< 18		TRUE
>= 60		TRUE
	>= 30	TRUE
		FALSE

This is a regular, **single-hit** decision table specified by the keyword “**DecisionTable**”. It executes rules in the top-down order and stops when one rule is satisfied. You may use the keyword “**DecisionTableSingleHit**” as well. As you may guess, the first rule sets “Eligible for Extra 5 Days” to TRUE when Employee’s “Age in Years” (another decision variable) is less than 18. If not, the second rule will do the same for employees age 60 or older. The third rule sets “Eligible for Extra 5 Days” to TRUE when Employee’s “Years of Service” (another decision variable) is more or equal to 30. If all first 3 rules fail, then the last rule (so-called “default” rule) will set “Eligible for Extra 5 Days” to FALSE.

Similarly, the following decision table specifies decision logic for the sub-goal “**Eligible for Extra 3 Days**”:

DecisionTable SetEligibleForExtra3Days		
If	If	Then
Age in Years	Years of Service	Eligible for Extra 3 Days
	>= 30	TRUE
>= 60		TRUE
		FALSE

And finally, the following decision table specifies decision logic for the sub-goal “**Eligible for Extra 2 Days**”:

DecisionTable SetEligibleForExtra2Days		
If	If	Then
Age in Years	Years of Service	Eligible for Extra 2 Days
	[15..30)	TRUE
>= 45		TRUE
		FALSE

This completes the representation of the business logic for this decision model.

Glossary

Every decision model requires that all used decision variables (goals and input variables used in the above tables) are described in the special table called “**Glossary**”. Here is an example of a glossary described in the file “*VacationDays/rules/Glossary.xls*”:

Glossary glossary			
Variable Name	Business Concept	Attribute	Type
Name	Employee	name	String
Vacation Days		vacationDays	int
Eligible for Extra 5 Days		eligibleForExtra5Days	boolean
Eligible for Extra 3 Days		eligibleForExtra3Days	boolean
Eligible for Extra 2 Days		eligibleForExtra2Days	boolean
Age in Years		age	int
Years of Service		service	int

The first column “**Variable Name**” contains the names of decision variables exactly how they were used inside the decision tables.

The second column “**Business Concept**” contains the name of a business concept to which these variables belong. There could be several business concepts, but this model contains only one concept “Employee”. The name of the business concept should be unique, start with a letter, and not include whitespaces. Note that merging cells inside the second column “Employee” indicates that all variables on the left belong to this concept.

The third column “**Attribute**” provides technical names for all decision variables – they will be used for the IT integration. These names should start with a small letter and not include whitespaces.

The fourth column “**Type**” describes the expected type of each decision variable such as “String” for text variables, “int” for integer variables, “double” or “float” for real variables, “Boolean” for logical variables, “String[]” for an array of text variables, etc. Actually, the types are the valid Java types but as a business analyst, you don’t have to even know this fact and just memorize the most frequently used keywords such as String, int, double, Boolean, Date.

Please note that in the classic version of OpenRules instead of the column “Type” you needed to define a special table “Datatype” such as below:

Datatype Employee	
String	id
int	vacationDays
boolean	eligibleForExtra5Days
boolean	eligibleForExtra3Days
boolean	eligibleForExtra2Days
int	age
int	service

You still may use this table in the Decision Manager.

A glossary may contain optional columns such as:

- “Description” with a plain English explanation of the term
- “UsedAs” with possible values Input, Output, InOut, Temp, Const
- “Domain” lists possible values of the variable, e.g. 1-120 for Age, Single, Married for Gender.

These columns could be very helpful to understand the decision model.

You may notice that some decision variables (goals and sub-goals) are hyperlinked to point to the decision tables (worksheets) that specify these goals. A click on the variable inside the glossary will immediately open the xls-file and the table that specifies this variable. It’s easy to do using Excel Hyperlinks and is very convenient for the future maintenance of your decision models when you want to find out “what is defined where”.

Environment

There is one more important file “*VacationDays/rules/DecisionModel.xls*” that describes the structure of the decision mode in the table “Environment”:

Environment	
include	Glossary.xls
	Rules.xls

This table states that our decision model includes files “Glossary.xls” and “Rules.xls”. Your model can use multiple xls- and xlsx-files located in different folders, and you can define them all in the Environment table relative to the file “DecisionModel.xls”. If your entire decision model is described in one Excel file, you don’t need to define the Environment table at all.

The Environment table usually also specifies various properties used to build, test, deploy this decision model:

Environment	
include	Glossary.xls
	Rules.xls
model.name	VacationDays
model.goal	Vacation Days
model.package	vacation.days
model.precision	0.001

The property “**model.name**” specifies the name of the decision model as it will be known

to the external world. This name should start with a letter and not contain whitespaces. The property “**model.goal**” specifies the name of the main goal from the glossary that your decision model should determine.

The property “**package.name**” specifies the name of the internal Java package in which OpenRules will put generated Java files. It could be any name similar to “com.company.problem” but it should start with a letter and contain no whitespaces.

The property “**model.precision**” specifies the precision of real numbers used to compare the expected and actual produced results.

Note that these properties could be overridden in the file “project.properties”.

Test Cases

The file “*VacationDays/rules/Test.xls*” describes test cases for this decision model:

DecisionTest testCases			
#	ActionDefine	ActionDefine	ActionExpect
Test ID	Age in Years	Years of Service	Vacation Days
Test A	17	1	27
Test B	25	5	22
Test C	49	30	30
Test D	49	29	24
Test E	57	32	30
Test F	64	42	30

This table describes 6 test cases where columns “ActionDefine” specify input decision variable and the columns “ActionExpect” specify the expected values. The first column “#” defines the name of the test.

When a decision model contains many decision variables, it can be more convenient to use an alternative way to specify test-cases using Data tables. For example, the table

Data Employee employees		
name	age	service
Name	Age in Years	Years of Service
A	17	1
B	25	5
C	49	30
D	49	29
E	57	32
F	64	42

describes an array of 6 test-employees. The first row specifies the table type using the keyword “**Data**”. Then after space, it contains the word “Employee” that is the same name we used as a business concept in the above Glossary to describe the type of our test-objects. And then after space, it contains the word “employees” that is the name of this array of employees.

The second row contains the names of attributes “name”, “age”, and “service” used as input for our decision model that should be the same as in the third column of the glossary.

The next 6 rows describe employees with specific values of these attributes.

Test cases with expected results defined in this table of the type “**DecisionTest**”:

DecisionTest testCases		
#	ActionUseObject	ActionExpect
Test ID	Employee	Vacation Days
Test A	employees[0]	27
Test B	employees[1]	22
Test C	employees[2]	30
Test D	employees[3]	24
Test E	employees[4]	30
Test F	employees[5]	30

Here the second column “ActionUseObject” defines the business objects associated with the business concepts defined in the glossary, in this case, “Employee”. And the third column “ActionExpect” specifies the expected values of the decision variable “Vacation Days”.

Building and Testing Decision Model

OpenRules provides a **decision engine** capable to build, test, and deploy business decision models on-premise or on-cloud. There are several bat-files in every project folder such as “*VacationDays*” that help a business user to execute OpenRules decision engine to build/test decision models.

File “project.properties”

The file “*VacationDays/project.properties*” specifies various properties of our project used to build, test, deploy this decision model. Here are the required properties:

```
model.file="rules/DecisionModel.xls"
test.file="rules/Test.xls"
```

The property “**model.file**” specifies the name of the main file that defines the structure of the decision model.

The property “**test.file**” specifies the name of the xls-file that defines test cases.

This file may also contain properties described above in the Environment table, and if they are defined here they will have a preference.

File “test.bat”

The file “*VacationDays/test.bat*” is used to build and test your decision model. This file is the same for all standard decision models and you don’t even have to look inside this file. When you double-click on this file, it will do the following:

- If the model hasn’t been built yet or some files were changed, it will execute these steps:
 - Analyze all files included in your decision model and check the model for possible errors;
 - If there are errors, it will show the errors pointing to the reasons and the proper place in xls-files;
 - If there are no errors, it will generate Java classes (in the folder “target”) needed internally to execute this decision model;
 - The generated Java classes will be compiled preparing the decision model for execution.
- After a successful build, the decision model will be executed against test cases described in the “test.file”.

You will also find the file “*VacationDays/build.bat*” that can be used to build the decision model as well, but it will execute the model only after rebuild.

Note. If you use Mac or Linux, instead of “test.bat” you can use the provided shell-files “test” or “build”.

Testing Results

During the execution, you will see the execution protocol similar to this one shown below for the test case D. To see the execution protocol you need to define the property trace=On in the file “project.properties”. This execution protocol shows all executed rules

with references to their locations in Excel, e.g. Rule #4 (B8:D8). The highlighted lines show the old and new values of decision variables modified by the current rule.

```
Execute 'VacationDays'
  SetEligibleForExtra5Days #4 (B8:D8)
    THEN 'Eligible for Extra 5 Days' = false
    Variables:
      Eligible for Extra 5 Days: false

  SetEligibleForExtra3Days #3 (B7:D7)
    THEN 'Eligible for Extra 3 Days' = false
    Variables:
      Eligible for Extra 3 Days: false

  SetEligibleForExtra2Days #1 (B5:D5)
    IF 'Years of Service' Is [15..30]
    THEN 'Eligible for Extra 2 Days' = true
    Variables:
      Eligible for Extra 2 Days: false --> true
      Years of Service: 29

  CalculateVacationDays #1 (B5:F5)
    THEN 'Vacation Days' = 22
    Variables:
      Vacation Days: 0 --> 22

  CalculateVacationDays #4 (B8:F8)
    IF 'Eligible for Extra 5 Days' Is false
    AND 'Eligible for Extra 2 Days' Is true
    THEN 'Vacation Days' + 2
    Variables:
      Eligible for Extra 2 Days: true
      Eligible for Extra 5 Days: false
      Vacation Days: 22 --> 24

Test 'Test D' completed OK. Elapsed time 44.25 ms
```

Explanations

After the execution, you also may look at the generated HTML report that explains which rules were executed and why (assuming the property report=On). The report is generated in a user-friendly HTML format and can be seen with any browser. It contains all rules that were actually executed and the values of all decision variables participated in these rules at the moment they were executed. Here is the report created by OpenRules in the file “report/TestD.html”:

Decision "VacationDays" (Test D)**Executed Decision Tables and Rules (Sat Nov 28 12:01:18 EST 2020)**

Decision Table: Rule# (Cells)	Executed Rule	Variables and Values
SetEligibleForExtra5Days: 4 (B8:D8)	THEN 'Eligible for Extra 5 Days' = false	Eligible for Extra 5 Days=false
SetEligibleForExtra3Days: 3 (B7:D7)	THEN 'Eligible for Extra 3 Days' = false	Eligible for Extra 3 Days=false
SetEligibleForExtra2Days: 1 (B5:D5)	IF 'Years of Service' Is [15..30) THEN 'Eligible for Extra 2 Days' = true	Years of Service=29 Eligible for Extra 2 Days= {old:false, new:true}
CalculateVacationDays: 1 (B5:F5)	THEN 'Vacation Days' = 22	Vacation Days={old:0, new:22}
CalculateVacationDays: 4 (B8:F8)	IF 'Eligible for Extra 5 Days' Is false AND 'Eligible for Extra 2 Days' Is true THEN 'Vacation Days' + 2	Eligible for Extra 5 Days=false Eligible for Extra 2 Days=true Vacation Days={old:22, new:24}

Elapsed time = 16 ms

DECISION MODELING APPROACH

OpenRules provides all the necessary tools to support the modern decision modeling methodology such as the [Goal-Oriented Decision Modeling](#). It allows business analysts to develop and maintain operational business decision models and deploy them as decision microservices. Subject matter experts without help from programmers can create decision models using only familiar MS Excel (or Google Sheets) as an editor and OpenRules Decision Manager as a building and execution environment.

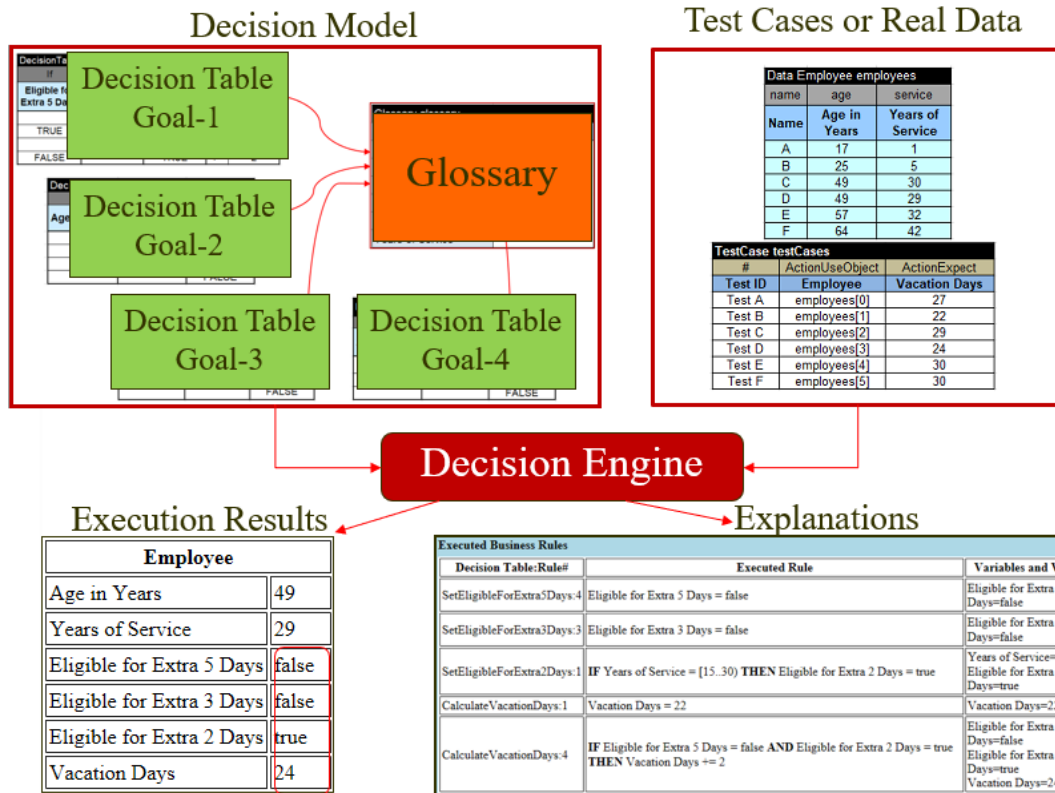
What is Decision Model

From the OpenRules® perspective a decision model consists of:

- Decision variables that can take specific values from domains of values
- Decision rules (frequently expressed as decision tables) that specify relationships

between decision variables.

The above introductory example shows a typical representation of a decision model as a glossary surrounded by decision tables that specify decision logic for different goals and sub-goals as in the following picture:



All decision variables should be described in the special table “Glossary”. Some of these decision variables are known (decision input) and some of them are unknown (decision output) that may represent goals/sub-goals. By executing a decision model OpenRules decision engine finds a decision that assigns values to unknown decision variables following the business logic specified by decision rules.

Goal-Oriented Decision Modeling

OpenRules uses a goal-oriented approach to decision modeling described in this [book](#). It promotes a top-down approach that starts with the definition of the top-level Decision Goal (not with rules or data). You put the top-level goal into a glossary and define its business logic using a decision table that specifies its sub-goals using sub-goals and other decision variables. You continue this process for all sub-goals until business logic

for all goals and sub-goals is defined.

For example, the introductory decision model has the top-level goal called “Vacation Days” – the only decision variable added to the initial glossary. Its decision logic defines in the decision table “CalculateVacationDays” that specifies 3 sub-goals: “Eligible for Extra 5 Days”, “Eligible for Extra 3 Days”, “Eligible for Extra 2 Days”. We added these sub-goals to the glossary and specified their own decision tables. These decision tables identified two input variables “Age in Years” and “Years of Service“, which we also added to the glossary. Then we added test cases and executed the decision model.

The real-world decision models can be much more complex, contain much more rules, but the methodological approach remains the same.

OpenRules allows a business analyst to represent and maintain decision logic directly in Excel. The following sections describe major OpenRules decisioning constructs.

DECISION TABLES

OpenRules use the classical decision tables that were in the heart of OpenRules from its introduction in 2003 and became the major decisioning construct of the DMN standard. OpenRules utilizes MS Excel and/or Google Sheets as the most powerful and commonly known table editors (but doesn’t rely on Excel’s formulas).

Decision Table Structure

OpenRules uses the keyword “**DecisionTable**” for the most frequently used single-hit decision tables. For example, let’s consider a very simple decision table:

DecisionTable DefineSalutation					
Condition		Condition		Conclusion	
Gender		Marital Status		Salutation	
Is	Male			Is	Mr.
Is	Female	Is	Married	Is	Mrs.
Is	Female	Is	Single	Is	Ms.

Its first row contains the keyword “**DecisionTable**” and a unique table’s name such as “DefineSalutation” (no spaces allowed). The second row uses the keywords “**Condition**”

and **“Conclusion”** to specify the types of decision table columns. The third row contains the names of decision variables expressed in plain English (spaces are allowed). The columns of a decision table define conditions and conclusions using different operators and operands appropriate to the decision variable specified in the column headings.

The rows below the decision variable names specify multiple rules. For instance, the second rule can be read as:

“IF Gender is Female AND Marital Status is Married THEN Salutation is Mrs”.

This is an example of the **horizontal** decision table when rules are defined from top to bottom. The same decision table may be presented in the **vertical** format when rules are presented from left to right:

DecisionTable DefineSalutation				
Condition	Gender	Is	Is	Is
		Male	Female	Female
Condition	Marital Status		Is	Is
			Married	Single
Conclusion	Salutation	Is	Is	Is
		Mr.	Mrs.	Ms.

If some cells in the rule conditions are empty, it is assumed that this condition is satisfied. A decision table may have no conditions, but it always should contain at least one conclusion/action.

The conditions in a decision table are always connected by the logical operator “AND” and never by the operator “OR”. Each rule can be read as:

IF Condition-1 AND Condition-2 AND ...

THEN Conclusion-1 AND Conclusion-2 AND ...

When you need to use “OR”, you may add another rule that is an alternative to the previous rule(s). However, some conditions may have a decision variable defined as an array or a list of values. Within such array-conditions “ORs” can be expressed using commas. Consider for example the following decision table from the standard project “UpSelRules”:

DecisionTable DefineUpSellProducts								
Condition		Condition		Condition		Conclusion		Action
Customer Profile		Customer Products		Customer Products		Offered Products		Recommendation
Is One Of	Bronze,Silver	Include	Product 1	Do Not Include	Product 2	Are	Product 2, Product 4, Product 5	Additional Products 2,4,5
Is One Of	Bronze,Silver	Include	Product 1, Product 3	Do Not Include	Product 6, Product 7, Product 8	Are	Product 6, Product 7, Product 8	Additional Products 6,7,8
Is One Of	Bronze,Silver	Include	Product 1, Product 2	Do Not Include	Product 6, Product 7, Product 8	Are	Product 4, Product 5, Product 7, Product 8, Product 9	Additional Products 4,5,7,8,9
Is One Of	Gold	Include	Product 1	Do Not Include	Product 6, Product 7, Product 5	Are	Product 9, Product 7, Product 8, Product 4, Product 5, Product 10	Gold Package
Is One Of	Platinum	Include	Product 1, Product 2	Do Not Include	Product 6, Product 7, Product 5	Are	Product 9, Product 7, Product 8 with no annual fee, Product 4, Product 5 with no charge, Product 10	Platinum Package
						Are	None	Sorry, no products to offer

Here the decision variables “Customer Profile”, “Customer Product”, and “Offered Products” are arrays of strings. In this case, the second rule can be read as:

IF Customer Profile is one of Bronze **or** Silver
 AND Customer Products include Product 1 and Product 3
 AND Customer Products do not include Product 6, Product 7, and Product 8
 THEN Offered Products ARE Product 6, Product 7, and Product 8
 AND Recommendation is Additional Products 6,7,8.

Execution Logic

All rules are executed one-by-one in the order they are placed in the decision table. For the vertical decision tables, all rules (rows) are executed in top-down order. For horizontal decision tables, all rules (columns) are executed in left-to-right order.

The execution logic of one rule is the following:

IF ALL conditions are satisfied THEN execute ALL actions.

If at least one condition is violated (evaluation of the code produces **false**), all other conditions in the same rule are ignored and not evaluated. Actions are executed only if all conditions in the same rule are satisfied. Conditions and actions with empty cells (or hyphens) are ignored.

There is a simple rule that governs rules execution inside a decision table:

The preceding rules are evaluated and executed first!

However, a designer of decision tables may specify different execution logic by using one of the 3 supported types of decision tables:

- Single-hit rules tables
- Multi-hit decision tables
- Sequential decision tables.

Note. OpenRules also provide a constraint-based rule engine to execute decision models in the inferential mode when an order of rules inside decision tables and between tables is not important.

Single-Hit Decision Tables

By default, the decision tables that start with the keyword “**DecisionTable**” are single-hit, meaning they evaluate rules one-by-one and **stop after the first “hit”** when a rule is satisfied. You also can use explicitly use the keyword “**DecisionTableSingleHit**” instead of “DecisionTable”. All 3 tables in the introductory example that specify decision logic for extra vacation days give examples of single-hit decision tables.

Note that for single-hit decision tables the default rule should be the very last as it will be executed only when all preceding rules fail. It’s usually unconditional but you may use conditions to specified different defaults. If your rules do not cover all possible combinations of decision variables, it is a good practice to catch and report an “impossible” situation in the last rule.

Multi-Hit Decision Tables

Multi-hit decision tables are identified by the keyword “**DecisionTableMultiHit**” and they execute the following logic:

- All rules inside the decision table are being evaluated
- Those rules which conditions are satisfied will be marked as “to be executed”
- Then all actions for “to be executed” rules will be executed in the top-down order.

There are two important observations about the behavior of multi-hit decision tables:

- **Rule actions cannot affect the conditions of any other rules** in the decision table – there will be no re-evaluation of any conditions after actions execution
- **Rule overrides are permitted.** The action of any executed rule may override the action of any previously executed rule.

Let’s consider an example of driving eligibility logic: “A person of age 17 or older is eligible to drive. However, in Florida 16-year-olds can also drive”. If we try to present this rule using the default single-hit decision table, it may look as follows:

DecisionTable DefineDrivingEligibility					
Condition		Condition		Conclusion	
Driver's Age		US State		Driving Eligibility	
>=	17			Is	Eligible
=	16	Is Not	FL	Is	Not Eligible
=	16	Is	FL	Is	Eligible
<	16			Is	Not Eligible

Using a multi-hit decision table, we may present the same logic as:

DecisionTableMultiHit DefineDrivingEligibility					
Condition		Condition		Conclusion	
Driver's Age		US State		Driving Eligibility	
				Is	Eligible
<	17			Is	Not Eligible
=	16	Is	FL	Is	Eligible

In the second decision table, the first unconditional rule sets “Driving Eligibility” to

“Eligible” (default!). The second rule may reset it to “Not Eligible” for all people younger than 17. But for 16-year-olds living in Florida, the third rule will again assign the value “Eligible” to Driving Eligibility. Thus, the default values for a multi-hit table usually defined in the very first rule.

Multi-hit decision tables allow you to specify a very natural requirement:

More specific rules should override more generic rules!

The only thing a designer needs to do is to place "more specific" rules after "more generic" rules. For example, Florida’s driving eligibility rules can override the US rules if we define them after the US rules.

It is very convenient to use multi-hit decision tables to accumulate some data. For example, the following decision table accumulates “Application Risk Score” based on 3 different conditions:

DecisionTableMultiHit ApplicationRiskScore				
If	If	If	Conclusion	
Age	Marital Status	Employment Status	Application Risk Score	
			=	0
[18..21]			+	32
[22..25]			+	35
[26..35]			+	40
[36..49]			+	43
>=50			+	48
	S		+	25
	M		+	45
		UNEMPLOYED	+	15
		STUDENT	+	18
		EMPLOYED	+	45
		SELF-EMPLOYED	+	36

The first rule will unconditionally assign value 0. All other rules may increment the score using the operator “+” and the provided value (32, 35, 40, ...).

Sequential Decision Tables

There is one more type of decision table called “Sequential” and defined by the keyword “**DecisionTableSequence**”. These tables are similar to multi-hit decision tables, but they allow the actions of already executed rules to affect the conditions of rules specified after them (more like in programming languages). “DecisionTableSequence” supports the following rules execution logic:

- Rules are evaluated in top-down order and if a rule condition is satisfied, then the rule actions are immediately executed.
- Rule overrides are permitted. The action of any executed rule may override the action of any previously executed rule.

There are two important observations about the behavior of the sequential tables:

- Rule actions can affect the conditions of other rules
- There could be rule overrides when rules defined below already executed rules could override already executed actions.

Consider the following example:

DecisionTableSequence CalculateTaxableIncome	
If	Action
TaxableIncome	TaxableIncome
	AdjustedGrossIncome - DependentAmount
< 0	0

Here the decision variable “TaxableIncome” is present in both the condition and the action. The first (unconditional) rule will calculate and set its value using the proper formula. The second rule will check if the calculated value is less than 0. If it is true, this rule will reset this decision variable to 0. Of course, in this simple case, the same logic could be expressed with a single-hit table such as:

DecisionTable CalculateTaxableIncome		
Condition		Action
AdjustedGrossIncome		TaxableIncome
>	DependentAmount	AdjustedGrossIncome - DependentAmount
<=	DependentAmount	0

P.S. If your decision table contains hundreds or thousands of rules, single-hit is much more efficient than multi-hit.

Conditions

There are two the most used types of conditions specified by the keywords “**Condition**” and “**If**” in the second row of a decision table, e.g.:

Condition	
Gender	
Is	Male

If
Gender
Male

Condition	
Amount	
>	1000

If
Amount
> 1000

The columns of the type “Condition” have two sub-columns: one for an operator like “is” or “>” and another - for a value. The columns of the type “If” don’t have sub-columns and keep an operator and a value together in the same cell. If there are no operators in front of the value in the column “If” then the operator “Is” is assumed.

The condition cells can contain specific values like “1000” or “> 1000” but they also contain names of other decision variables like in the above decision table “CalculateTaxableIncome” where we compare if the decision variable AdjustedGrossIncome is larger (>) or <= than DependentAmount.

Comparing Strings

The following operators can be used for conditions to compare strings:

Is	To compare two strings are the same. Instead of “Is” you also can write “=” or “==” (with an apostrophe in front of them to avoid confusion with Excel’s own formulas)
Is Not	To check if two strings are not the same. Synonyms: !=, isnot, Is Not Equal To, Not, Not Equal., Not Equal To

Is Empty	To check if a string is empty
Contains	To compare if a decision variable contains certain values. For example, “House” contains “use”. The comparison is not case-sensitive. Synonym: Contain
Does Not Contain	To compare if a decision variable does not contain certain values. For example, “House” doesn’t contain “user”. The comparison is not case-sensitive. Synonym: DoesNotContain
Starts With	To compare if a decision variable starts with certain values. For example, “House” starts with “ho”. The comparison is not case-sensitive. Synonym: Start
Match	To check if a decision variable matches standard <u>regular expressions</u> . For example, you can use the expression “\d{3}-\d{3}-\d{4}” to check if the content of the decision variable is a valid US phone number such as 732-993-3131. Synonyms: Matches, Is Like, Like
No Match	To check if a decision variable doesn’t match a <u>regular expression</u> . For example, you can use the expression “[0-9]{5}” to check if the content of the decision variable consists of 5 digits like a valid US zip code. The condition is satisfied if it is not true. Synonyms: NotMatch, Does Not Match, Not Like, Is Not Like, Different, Different From

Comparing Numbers

The following operators can be used for conditions to compare numbers (integer, real, or BigDecimal):

Is	To compare two numbers are the same. Instead of “Is” you also can write “=” or “==” (with an apostrophe in front of them to avoid confusion with Excel’s own formulas)
Is Not	To check if two numbers are not the same. Synonyms: !=, isnot, Is Not Equal To, Not, Not Equal., Not Equal To

>	To check a number represented by the decision variable is strictly larger than the number in the column' cell. Synonyms: Is More, More, Is More Than, Is Greater, Greater, Is Greater Than
>=	To check a number represented by the decision variable is larger than or equal to the number in the column' cell. Synonyms: Is More Or Equal, Is More Or Equal To, Is More Than Or Equal To, Is Greater Or Equal To, Is Greater Than Or Equal To
<=	To check a number represented by the decision variable is smaller than or equal to the number in the column' cell. Synonyms: Is Less Or Equal, Is Less Than Or Equal To, Is Less Than Or Equal To, Is Smaller Or Equal To, Is Smaller Than Or Equal To, Is Smaller Than Or Equal To
<	To check a number represented by the decision variable is strictly smaller than the number in the column' cell. Synonyms: Is Less, Less, Is Less Than, Is Smaller, Smaller, Is Smaller Than
Within	To check if a decision variable is within the provided interval. The interval can be defined as: [0;9], (1;20], 5–10, between 5 and 10, more than 5 and less or equals 10. Synonyms: Inside, Inside Interval, Interval
Outside	To check if a decision variable is outside of the provided interval. The interval can be defined as: [0;9], (1;20], 5–10, between 5 and 10, more than 5 and less or equals 10. Synonyms: Outside Interval

In columns of the type “IF” the operator “Within” is being assumed when an interval is specified. For example,

If
Current Hour
[0..11)

checks if the variable “Current Hour” is within the interval [0..11) assuming that 0 is included and 11 is not included.

Using Natural Language Inside Decision Tables

OpenRules allows a rules designer to use “almost” natural language expressions to represent intervals of numbers inside the columns of the type “IF”. You may define FROM-TO intervals in practically unlimited English using such phrases as: "500-1000", "between 500 and 1000", "Less than 16", "More or equals to 17", "17 and older", "< 50", ">= 10,000", "70+", "from 9 to 17", "[12;14)", etc.

You also may use many other ways to represent an interval of integers by specifying their two bounds or sometimes only one bound. Here are some examples of valid integer intervals:

Cell Expression	Comment
5	equals to 5
[5,10]	contains 5, 6, 7, 8, 9, and 10
5;10	contains 5, 6, 7, 8, 9, and 10
[5,10)	contains 5, 6,7,8, and 9 (but not 10)
[5..10)	The same as [5,10)
5 - 10	contains 5 and 10
5-10	contains 5 and 10
5- 10	contains 5 and 10
-5 - 20	contains -5 and 20
-5 - -20	error: left bound is greater than the right one
-5 - -2	contains -5, -4, -3, -2
from 5 to 20	contains 5 and 20
less 5	does not contain 5
less than 5	does not contain 5
less or equals 5	contains 5
less or equal 5	contains 5
less or equals to 5	contains 5
smaller than 5	does not contain 5
more 10	does not contain 10
more than 10	does not contain 10
10+	more than 10

>10	does not contain 10
>=10	contains 10
between 5 and 10	contains 5 and 10
no less than 10	contains 10
no more than 5	contains 5
equals to 5	equals to 5
greater or equal than 5 and less than 10	contains 5 but not 10
more than 5 less or equal than 10	does not contain 5 and contains 10
more than 5,111,111 and less or equal than 10,222,222	does not contain 5,111,111 and contains 10,222,222
[5'000;10'000'000)	contains 5,000 but not 10,000,000
[5,000;10,000,000)	contains 5,000 but not 10,000,000
(5;100,000,000]	contains 5,000 and 10,000,000

You may represent integer intervals as you usually do in plain English. The only limitation is the following: *lower bound* should always go before *upper bound*!

Along with integer intervals, you may similarly represent intervals of real numbers. The bounds of double intervals could be integer or real numbers such as [2.7; 3.14).

Comparing Dates

OpenRules naturally supports date comparison with the operators =, !=, >, >=, <=, and < like in the following example:

DecisionTable DefineChild		
Condition		Action
Date of Birth		Is Child
<	January 1, 2010	FALSE
>=	January 1, 2010	TRUE

OpenRules assumes that Date variables are presented in the standard format specific for a country (locale). For example, the standard US date formats are "MM/dd/yyyy" and "EEE MMM dd HH:mm:ss zzz yyyy". Additionally, OpenRules always tries to recognize the following formats:

- MM/dd/yy HH:mm
- yyyy-MM-dd

To compare two Date variables, you may do it as in the following decision table:

DecisionTable CompareDates		
Condition		Message
Date 1		Message
<	Date 2	Date 1 < Date 2
>=	Date 2	Date 1 >= Date 2

You may see more examples of how to use new Date operators by analyzing the sample project "HelloWithDates" available in the downloaded workspace "openrules.samples".

By default, OpenRules compares dates ignoring time. If you want to use time components of the Date variables, instead of the operators such as "<" you should use the operator "< **time**", as in the table below:

DecisionTable ComparePassengerFlights						
Condition		Condition		Condition		Action
Flight 1 Is Suitable		Flight 2 Is Suitable		Flight 1 Arrival		Flight 1 Score
Is	TRUE	Is	FALSE			1
Is	FALSE	Is	TRUE			0
Is	TRUE	Is	TRUE	< time	Flight 2 Arrival	1
Is	TRUE	Is	TRUE	> time	Flight 2 Arrival	0
Is	TRUE	Is	TRUE	= time	Flight 2 Arrival	1

This table is a part of the decision model "FlightRebooking" also included in the workspace "openrules.samples".

Comparing Boolean Values

If a decision variable has type "boolean", e.g. "Employee is Veteran", you can check if it's true by using the following conditions:

Condition		or	If	
Employee Is Veteran			Employee Is Veteran	
Is	TRUE			TRUE

You can use the following boolean values:

- True, TRUE, Yes, YES
- False, FALSE, No, NO

You also may compare two Boolean decision variables as below:

Condition	
Company 1 Eligibility	
Is	Company 2 Eligibility

Other Condition Types

There several convenience condition types described in the examples below.

ConditionBetween

ConditionBetween	
Amount	
10	20

This condition of the type “**ConditionBetween**” check if the variable “Amount” is more or equals to 10 and less or equals to 20.

ConditionVarOperValue

When your decision table contains too many columns it may become too large and unmanageable. In practice, large decision tables have many empty cells because not all decision variables participate in all rule conditions even if the proper columns are reserved or all rules. For example, here is a decision table taken from a real-world application that has more than 20 conditions (not all of them shown) with many empty rule cells:

DecisionTable classificationRules									
Condition		Condition		Condition		Conclusion		Conclusion	
C_OTH_EXPNS_AMT		A_ESTATE_TAX_AMT		...		ClassifiedAs		HitRate	
>=	398	>=	10054		Oper	Value	Is	High	= 66
							Is	Low	= 63
>=	53						Is	Low	= 49
							Is	Low	= 86
							Is	Low	= 78
							Is	Other	= 56

To make this decision table more compact, instead of the standard column's structure

with two sub-columns

Condition	
Variable Name	
Oper	Value

we may use another column representation with 3 sub-columns:

ConditionVarOperValue		
Attribute		
Variable Name	Oper	Value

This way the above table will be replaced may with a much more compact table that may look as follows:

DecisionTable classificationRules											
ConditionVarOperValue			ConditionVarOperValue			...	ConditionVarOperValue			Conclusion	
Attribute			Attribute			...	Attribute			ClassifiedAs	
Variable Name	Oper	Value	Variable Name	Oper	Value	...	Variable Name	Oper	Value	Is	HitRate
C_OTH_EXPNS_AMT	>=	398	A_ESTATE_TAX_A	>=	10054					Is	66
E_PRTSCRPT_TOT_LC	<=	-6955	AGI_TPI_RATIO	<=	0.95993					Is	63
C_OTH_EXPNS_AMT	>=	53	ORD_DIVIDENDS	<=	6617					Is	49
TAXABLE_INC_TPI_R	<=	0.810447	TENT_TAX_AMT	>=	301630					Is	86
DIVIDENDS_AND_INT	<=	12348	EXTNSN_PYMNT	<=	30000					Is	78
										Is	56

P.S. Similarly, instead of a column of the type “Conclusion” you may use a column of the type “ConclusionVarOperValue” with 3 sub-columns that represent a variable name, an operator, and a value.

Conditions on Collections

In practice, business rules deal not only with separate decision variables but also with a collection of decision variables such as arrays or lists. OpenRules provides necessary constructs to use collections in conditions and conclusions.

Condition with Collection Operators

For example, this is a fragment of the decision table from the sample project “UpSellRules”:

DecisionTable DefineUpSellProducts					
Condition		Condition		Condition	
Customer Profile		Customer Products		Customer Products	
Is One Of	Bronze,Silver	Include	Product 1	Do Not Include	Product 2
Is One Of	Bronze,Silver	Include	Product 1, Product 3	Do Not Include	Product 6, Product 7, Product 8

Here the variable “Customer Profile” is a regular variable of the type String, and the first condition simply checks if the value of the variable “Customer Profile” is one of two strings “Bronze” or “Silver”. However, the variable “Customer Products” is an array of strings that identifies all products this customer already has. So, the second condition checks if this array includes product “Product 1” (the first rule) or if it includes the products “Product 1” and “Product 2” (the second rule). The third condition checks if this array doesn’t include product “Product 2” (the first rule) or if it doesn’t include the products “Product 6”, “Product 7”, and “Product 8” (the second rule).

The following operators can be used for conditions defined on collections:

Is One Of	For integer and real numbers, and for strings. Checks if a value is among elements of the domain of values listed through a comma. Synonyms: Is One, Is One of Many, Is Among, Among
Is Not One Of	For integer and real numbers, and for strings. Checks if a value is NOT among elements of the domain of values listed through a comma. Synonyms: Is not among, Not among
Include	To compare two arrays. Returns true when the first array (decision variable) include all elements of the second array (value within decision table cell). Synonyms: Include All
Exclude	To compare two arrays. Returns true when the first array (decision variable) does not include all elements of the second array (value within decision table cell). This operator is opposite to the operator “Include”. Synonyms: Exclude One Of, Do Not Include, Does Not Include, Include Not All

Exclude All	To compare two arrays. Returns true when the first array (decision variable) does not include any element of the second array (value within decision table cell). Synonyms: Do Not Include All, Does Not Include All
Intersect	To compare an array with an array. Synonyms: Intersect With, Intersects

If the decision variables do not have an expected type for the specified operator, the proper syntax error will be diagnosed.

Note that the operators **Is One Of**, **Is Not One Of**, **Include**, **Exclude**, and **Does Not Include** work with arrays or lists of values separated by commas. Sometimes a comma could be a part of the value and you may want to use a different separator. In this case, you may simply add your separator character at the end of the operator. For example, if you want to check that your variable “Address” is if one of “*San Jose, CA*” or “*Fort Lauderdale, FL*”, the comma between City and State should not be confused with a separator. In this case, you may use the operator “**Is One Of #**” or “**Is One Of separated by #**” with an array of possible addresses described as “*San Jose, CA#Fort Lauderdale, FL*”. Instead of the character ‘#’ you may use any other character-separator.

ConditionMap

If the decision variable is a map (e.g. an instance of Java class HashMap) the following condition

ConditionMap	
My Map	
key1	value5

will check if the map-variable “My Map” contains a pair (“key1”, “value5”).

Conclusions

Simple Conclusions/Actions

There are two most used types of conclusions specified by the keywords “**Conclusion**” and “**Action**” or “**Then**”, e.g.:

Conclusion		Conclusion		Action		Then	
Eligibility		Amount		Eligibility		Amount	
Is	TRUE	=	20	TRUE		20	

The columns of the type “Conclusion” have two sub-columns: one for an operator like “Is” or “=” and another - for a value. The columns of the type “Action” (or its synonym “Then”) don’t have sub-columns and assumes the operator “Is” or “=”.

The following operators can be used inside decision table conclusions:

Is	Assigns one value to the conclusion decision variable. Synonyms: =, == When you use “=” or “==” inside Excel, you have to put an apostrophe in front of them to avoid confusion with Excel’s formulas.
Assign Plus	Takes the conclusion decision variable, adds to it a value from the rule cell and saves the result in the same decision variable. Synonym: +=
Assign Minus	Takes the conclusion decision variable, subtracts from it a value from the rule cell and saves the result in the same decision variable. Synonym: -=
Assign Multiply	Takes the conclusion decision variable, multiplies it by a value from the rule cell and saves the result in the same decision variable. Synonym: *=
Assign Divide	Takes the conclusion decision variable, divides it by a value from the rule cell and saves the result in the same decision variable. Synonym: /=

The accumulation operators +=, -=, *=, and /= are usually used in scorecards such as then decision table [above](#).

You may assign string using simple conclusion columns like in these decision tables:

DecisionTable DefineGreeting			
If	Then		
Current Hour	Greeting		
[0..11)	Good Morning	or	

DecisionTable DefineHelloStatement	
Conclusion	
Hello Statement	
Is	Greeting + ", " + Salutation + " " + Name + "!"

You may assign numbers (integer, real, BigDecimal) using simple conclusion columns like in these decision tables:

DecisionTable DefinePhaseOfTheMoonRisk			
Condition		Conclusion	
Phase of the Moon		Phase of the Moon Risk	
Is	New Moon	Is	0.01
Is	Half Moon	Is	0.25

DecisionTable CalculateCreatinineClearance	
Action	
Patient Creatinine Clearance	
(140 - Patient Age) * Patient Weight / (Patient Creatinine Level * 72)	

You may use "" (double quotes) or " " in the action cells to assign an empty string or a space character to a String variable.

Conclusions on Collections

When you want to assign some values to decision variables that are collections (such as arrays or lists) you can use the following operators:

Are	Assigns one or more values listed through commas to the conclusion variable that is expected to be an array
Add	Adds one or more values listed through commas to the conclusion variable that is expected to be an array

For example, if the decision variable "Offered Products" is an array (or a list) of strings, you use the following conclusion to assign to 3 products:

Conclusion	
Offered Products	
Are	Product 2, Product 4, Product 5

If after this conclusion you will also apply the following conclusion

Conclusion	
Offered Products	
Add	Product 7, Product 8

then the value of the variable “Offered Products” will become

{ “Product 2”, “Product 4”, “Product 5”, “Product 7”, “Product 8” }

Displaying Messages

There is a special conclusion type for displaying messages that is like

“Action” but used the keyword “Message”. For example, this following action when executed displays the message “Employee is eligible to 27 vacation days”:

Message	
Display	
Employee is eligible to 27 vacation days	

But if you want instead of hard-coded 27 days display the actual number of vacation days already calculated in the decision variable “VacationDays”, you may use this action:

Message	
Display	
Employee is eligible to VacationDays vacation days	

The variable “VacationDays” will be replaced with its current value.

Sometimes, you want your message to refer to the rule that was applied. To do this, you may associate unique names with all rules in the column of the type “#” and then refer

to these names in the Message column using \$RULE_ID like in the following example:

DecisionTable Swap			
#	If	Then	Message
Rule Id	X	X	Message
Rule 1	1	2	Executed rule <\$RULE_ID>
Rule 2	2	1	

Expressions

OpenRules allows you to use expressions (formulas) in the decision table cells.

There are two types of expressions:

- Simple formulas
- Java Snippets.

Formulas

You may use naturally looking formulas that contain the names of your decision variable and traditional operation signs (+, -, *, /, and more) along with brackets to define the order of the operations. Here is a simple example:

Action
TaxableIncome
AdjustedGrossIncome - DependentAmount

That will assign a difference between values of AdjustedGrossIncome and DependentAmount to the variable TaxableIncome. Here is a more complex formula from the example “PatientTherapy” that calculates Patient Creatinine Clearance:

DecisionTable CalculateCreatinineClearance	
Action	
Patient Creatinine Clearance	
(140 - Patient Age) * Patient Weight / (Patient Creatinine Level * 72)	

When your resulting decision variable has type “String” you can use the operator “+” to concatenate different strings (or even numbers). For example, this conclusion

DecisionTable DefineHelloStatement	
Conclusion	
Hello Statement	
Is	Greeting + ", " + Salutation + " " + Name + "!"

will use the values of decision variable “Greeting”, “Salutation”, and “Name” (defined in the Glossary of the standard project Hello) to define a Hello Statement that may look like “Goof Afternoon, Ms. Robinson!”.

Alternatively, you may explicitly use string intrapolation by taking decision variable names the double curly braces, {{ and }}. It will allow you not to use pluses and quotations and simply write:

DecisionTable DefineHelloStatement	
Conclusion	
Hello Statement	
Is	{{Greeting}}, {{Salutation}} {{Name}}!

You also can use some simple functions like min(x,y) and max(x,y) like in the following actions borrowed from the standard project 1040EZ:

Action	Action	Action
LineC	LineD	LineE
max(LineA,LineB)	4750	min(LineC,LineD)

Here is a list of supported operators and functions:

Feature	Syntax	Examples
Numbers	regular integer or real numbers	10, 465.25, -25, 3.14
Add	x + y	3+2
Subtract	x - y	3 - 2
Multiply	x * y	3 * 2
Divide	x / y	3/2
Power: x^y	x**y or x^y or power(x,y)	5**2
Negate	-x	-3

Comparison	$x < y$ $x \leq y$ $x = y$ $x <> y$ or $x \neq y$ $x \geq y$ $x > y$	$2 <> 3$ [produces 1] $2 \neq 2$ [produces 0]
Logical "and"	x and y	1 and 1 [produces 1] 1 and 0 [produces 0] 0 and 0 [produces 0]
Logical "or"	x or y	1 and 1 [produces 1] 1 and 0 [produces 1] 0 and 0 [produces 0]
Absolute value	$\text{abs}(x)$	$\text{abs}(-5)$ [produces 5] $\text{abs}(5)$ [produces 5]
Maximum between two numbers	$\text{max}(x,y)$	$\text{max}(5,6)$ [produces 6]
Minimum between two numbers	$\text{min}(x,y)$	$\text{min}(5,6)$ [produces 5]
Floor	$\text{floor}(x)$	$\text{floor}(3.5)$ [produces 3] $\text{floor}(-3.5)$ [produces -3]
Ceiling	$\text{ceil}(x)$ or $\text{ceiling}(x)$	$\text{ceil}(3.5)$ [produces 4] $\text{ceil}(-3.5)$ [produces -3]
" π "	π	The mathematical constant " π "
e^x	$\text{exp}(x)$	$\text{exp}(1) = 2.7182818284590451$
Rounding	$\text{round}(x)$	$\text{round}(3.5)$ [produces 4] $\text{round}(-3.5)$ [produces -4]
Square root	$\text{sqrt}(x)$	$\text{sqrt}(9)$ [produces 3]

Java Snippets

OpenRules also allows its users to write any arithmetic and logical formulas using Java expressions placed directly in the decision table cells but preceding by a sign “:=”. To simplify the references to decision variables inside such Java expressions we introduced special macros. Java snippets are less friendly to compare with simple expressions. For example, the above conclusion for the “Hello Statement” could be written using the following Java snippet:

DecisionTable DefineHelloStatement	
Conclusion	
Hello Statement	
Is	:= \${Greeting} + ", " + \${Salutation} + " " + \${Name} + "!"

Similarly, the expression that calculates “Patient Creatinine Clearance” could be written using the following Java snippet:

DecisionTable CalculateCreatinineClearance	
Action	
Patient Creatinine Clearance	
	:= (140 - \$I{Patient Age}) * \$R{Patient Weight} / (\$R{Patient Creatinine Level} * 72)

In the first example, the macro \${Greeting} refers to the value of the decision variable “Greeting” that has type String. In the second example, the decision variable “Patient Age” has type “int” and decision variable “Patient Weight” has type “double” (real). So, in the above Java snippet we refer to them using macros \$I{Patient Age} and \$R{Patient Weight} where the letter “I” and “R” after \$ point to the variable types (I – for int, and R for real).

It's possible to hide a Java snippet inside a special table of the type “Method”, e.g.:

Method double CreatinineClearanceFormula(Decision decision)
double pcc = (140 - \$I{Patient Age}) * \$R{Patient Weight} / (\$R{Patient Creatinine Level} * 72); return decimal(pcc,2);

Then we may call this method from this decision table:

DecisionTable CalculateCreatinineClearance
Action
Patient Creatinine Clearance
<code>:= CreatinineClearanceFormula(decision);</code>

Here is the list of all allowed macros:

Macro Format	Variable Type	Expanded As
<code>\${variable name}</code>	String	<code>decision.getString("variable name")</code>
<code>\$I{variable name}</code>	Integer	<code>decision.getInt("variable name")</code>
<code>\$R{variable name}</code>	Real	<code>decision.getReal("variable name")</code>
<code>\$L{variable name}</code>	Long	<code>decision.getLong("variable name")</code>
<code>\$D{variable name}</code>	Date	<code>decision.getDate("variable name")</code>
<code>\$B{variable name}</code>	Boolean	<code>decision.getBool("variable name")</code>
<code>\$G{variable name}</code>	BigDecimal	<code>decision.getBigDecimal("variable name")</code>
<code>\$G{number}</code>	BigDecimal	<code>new BigDecimal(number)</code>
<code>\$O{BusinessConcept}</code>	Object	Refers to a business concept defined in the Glossary

Inside Java snippets you may use regular operators "+", "-", "*", "/", "%" and any other valid Java operators. You may freely use parentheses to define the desired execution order. You also may use any standard or 3rd party Java methods and functions, e.g.

```
:= Math.min( $R{Line A}, $R{Line B} )
```

If the content of the decision table cell contains only a value of a decision variable, say "Customer Location", then along with

```
:= ${Customer Location}
```

you may simply write

```
$Customer Location
```

even without := and {..}.

In the following table

DT DefineWhomToCharge					
Condition		Condition		Conclusion	
Vendor		Provider		Charged Entity	
Is Empty	FALSE			Is	\$Vendor
		Is Empty	TRUE	Is	UNKNOWN
		Is Not One Of	ABC, KLM, XYZ	Is	\$Provider

the conclusion-column contains references \$Vendor and \$Provider to the values of decision variables Vendor and Provider. You may also use similar references inside arrays. For example, to express a condition that a Vendor should not be among providers, you may use the operator “Is Not One Of” with an array “ABC, \$Vendor, XYZ”. By the way, this decision table starts with the keyword “DT” which is a valid abbreviation for “DecisionTable”.

While being more technical, Java snippets remove any limits from the expressive power of OpenRules. They allow using complex Java constructs like loops, functions, recursion, etc. They allow you to use any Java libraries created by your own programmers or by 3rd parties.

Dealing with Dates

Here are examples of columns that assign dates:

Action	Conclusion
Scheduled Date	Selected Date
10/15/2020	= Current Date

When you need to apply arithmetic operations with date variables such as calculating the number of years, months, or days between dates, you still need to use OpenRules Java snippets. For these purposes, you may use static methods of the class "Dates" included in the standard OpenRules library "com.openrule.tools". For example, you may use the following Java snippet inside a condition cell of your decision table:

```
:= Dates.years( $D{Date1}, $D{Date2} ) >= 2
```

It checks that a number of years passed between the variables "Date1" and "Date2" is at

least 2 years. You may calculate the age of the person from its birthday as follows:

DecisionTable DefineAge
Action
Age
::= Dates.yearsToday(\$D{Date of Birth})

Similarly use the following methods:

Dates.months(Date d1, Date2 d2)

Dates.monthsToday(Date date)

Dates.days(Date d1, Date d2)

Dates.daysToday(Date d).

The standard library "com.openrule.tools" also includes methods that produce new dates:

```
addHours(date, hours)
addDays(date,days)
addMonths(date,months)
addYears(date,years)
setYear(date,year)
setMonth(date,month)
setDay(date,day)
today()
newDate(year,month,day)
newDate("yyyy-mm-dd")
```

You also may get integer values of year, month, and day by calling Dates methods `getYear(date)`, `getMonth(date)`, and `getDay(date)`.

All these methods can be used for dates arithmetic like in this example:

DecisionTableAssign DefineDates	
Variable	Value
Age	::= Dates.yearsToday(\$D{Date of Birth})
Date of Birth plus 6 Years	::= Dates.addYears(\$D{Date of Birth}, 6)
Today	::= Dates.today()

You just need to remember to add an "import.java" statement that points to "com.openrules.tools.Dates" to your Environment table.

Iteration over Collections of Objects

OpenRules provides business-friendly capabilities to deal with arrays and lists of objects. They allow a user to define which decision tables to execute against a collection of objects and to calculate values defined on the entire collection.

Standard sample projects "**AggregatedValues**" and "**AggregatedValuesWithLists**" demonstrate how to do it. The business concept Employee is defined in the Java class Employee with different customer attributes such as name, age, gender, maritalStatus, salary, and wealthCategory. Another class Department defined the business concept Department that include employees defined as a collection of employees using an array Employee[] or aa ArrayList<Employee>.

We want to process all employees in each department to calculate such Department's attributes as "minSalary", "totalSalary", "salaries", "richEmployees" "numberOfHighPaidEmployees", and other attributes, which are specified for the entire collection. Each employee within any department can be processed by the following rules:

DecisionTableMultiHit EvaluateOneEmployee															
Condition		Conclusion		Conclusion		Conclusion		Conclusion		Conclusion		Conclusion		Conclusion	
Salary		Total Salary		Max Salary		Min Salary		Number Of Employees		Number Of High-Paid Employees		Salaries		Wealth Category	
		+=	Salary	Max	Salary	Min	Salary	+=	1			Add	Salary		
>=	85000									+=	1			Is	HighPaid
<	85000													Is	Regular

Pay attention that we use here a multi-hit table, so all satisfied rules will be executed.

The first one unconditionally calculates the Total Salary, Maximal and Minimal Salaries, etc. The second rule defines Employee's Wealth Category, increases the Number of High-Paid Employees inside the department using the accumulation operator "+=", and adds this employee to the collection "Rich Employees".

To execute the above decision table for all employees, we utilize the decision table "EvaluateAllEmployees":

DecisionTable EvaluateAllEmployees						
Action	Action	Action	Action	Action	ActionIterate	
Max Salary	Min Salary	Total Salary	Number Of Employees	Number Of High-Paid Employees	Array of Objects	Rules
0	1000000	0	0	0	Employees	EvaluateOneEmployee

The first 5 actions in this table initialize the initial values of the variables we want to calculate. The last action has a special type "**ActionIterate**" with two sub-columns:

- Array of Objects – in this case, it is defined as "Employees" of the considered Department
- Rules that will be applied to each employee in the array/list "Employees" – this table will use the rules defined in the above decision table "EvaluateOneEmployee".

Thus, a combination of the two regular decision tables (similar to the above ones) provides business users with an intuitive way to apply rules over collections of business objects without the necessity to deal with programming constructions such as loops.

There is even a special decision table of the type "**DecisionTableIterate**" that allows you to iterate over arrays or lists of business objects. Here is an example:

DecisionTableIterate IterateOverDriversAndCars	
Array of Objects	Rules
Drivers	DefineDriverEligibilityRating
Drivers	DefineDriverEligibilityScore
Cars	DefineAutoEligibilityRating
Cars	DefineAutoEligibilityScore

This table iterates over arrays of objects, defined in the first column, applying the rules, defined in the second column, to every element of these arrays. This example is

borrowed from the sample project "**InsurancePremium**", which glossary consists of 3 business concepts: Driver, Car, and Client. The Client defines decision variables Drivers and Cars that are arrays of objects which have types Driver and Car correspondingly. Along with arrays of different objects, you also can use lists of objects.

Sorting Collections of Objects

OpenRules allows you to easily sort arrays or lists of your business objects. You can use a special decision table of the type "**DecisionTableSort**" to sort arrays of objects using regular decision tables that compare any two elements of such arrays.

For example, the sample project "**DecisionSortPassengers**" shows how to sort an array of passengers based on their frequent flier status. When the statuses of 2 passengers are the same, the actual numbers of frequent miles serve as a tiebreaker. It uses the following table to sort all passenger:

DecisionTableSort SortPassengers	
Array of Objects	
Passengers	

This table sorts the arrays "Passengers" where each passenger has known frequent flier status and a number of miles. During the sorting process, every two elements of this array are compared using the following decision table:

DecisionTable ComparePassengers							
Condition		Condition		Condition		Action	Action
Passenger 1 Status		Passenger 2 Status		Passenger 1 Miles		Passenger 1 Score	Passenger 2 Score
Is	GOLD	Is One Of	SILVER, BRONZE			1	0
Is		Is	GOLD	>	Passenger 2 Miles	1	0
Is		Is		<	Passenger 2 Miles	0	1
Is	SILVER	Is	GOLD			0	1
Is		Is	BRONZE			1	0
Is		Is	SILVER	>	Passenger 2 Miles	1	0
Is		Is		<	Passenger 2 Miles	0	1
Is	BRONZE	Is One Of	GOLD,SILVER			0	1
Is		Is	BRONZE	>	Passenger 2 Miles	1	0
Is		Is		<	Passenger 2 Miles	0	1

This table assigns scores to each pair of passengers (Passenger 1 and Passenger 2). So the passenger with higher credentials will receive a higher score. The name of this table "ComparePassengers" is composed of the word "Compare" and the name of the array "Passengers" (with omitting spaces if any). If you prefer to explicitly define the comparison rules, you can do it using the following decision table format:

DecisionTableSortUsingRules SortPassengers	
Array of Objects	Comparison Rules
Passengers	ComparePassengers

It's assumed that the elements of the array "Passengers" are instances of a Java class "Passenger" that extends the standard OpenRules class ComparableDecisionVariable. The proper Glossary is defined as follows:

Glossary glossary		
Variable	Business Concept	Attribute
Passengers	Problem	passengers
Passenger Name	Passenger	name
Passenger Status		status
Passenger Score		score
Passenger Miles		miles
Passenger 1 Status	Passenger1	status
Passenger 1 Score		score
Passenger 1 Miles		miles
Passenger 2 Status	Passenger2	status
Passenger 2 Score		score
Passenger 2 Miles		miles

Thus, the array "Passengers" by itself is a decision variable defined by the business concept "Problem". The glossary must include two business concepts "Passenger1" and "Passenger2", which names are formed by adding "1" and "2" to the type of the array elements. These business concepts should at a minimum include all decision variables that are being used by the comparison rules "ComparePassengers".

You may find a more complex example in the sample project "**FlightRebooking**" from the workspace "openrules.samples". Another sample project "**SortProducts**" demonstrates how to sort arrays of objects not inherited from ComparableDecisionVariable. Along with arrays of different objects, you also can use lists of objects.

GLOSSARY

You’ve already seen many examples of the “Glossary” table that is in the center of any decision model.

Standard Glossary

At the very least the table “Glossary” contains 3 columns:

- Decision Variable
- Business Concept
- Attribute.

The first column simply lists all of the decision variables using exactly the same names that were used inside the decision tables. The second column associates different decision variables with the business concepts to which they belong. Usually, you want to keep decision variables that belong to the same business concept together and merge all rows in the column “Business Concept” that share the same concept.

Here is a typical table of the type Glossary from the sample project “PatientTherapy”:

Glossary glossary			
Decision Variable	Business Concept	Attribute	Type
Encounter Diagnosis	DoctorVisit	encounterDiagnosis	String
Recommended Medication		recommendedMedication	String
Recommended Dose		recommendedDose	String
Drug Interaction Warning		warning	String
Patient Therapy		patientTherapy	String
Patient Name	Patient	name	String
Patient Age		age	int
Patient Weight		weight	double
Patient Allergies		allergies	String[]
Patient Creatinine Level		creatinineLevel	double
Patient Creatinine Clearance		creatinineClearance	double
Patient Active Medication		activeMedication	String

Along with standard 3 columns it may include the fourth column “Type” that specifies the types of decision variables. The typical types are:

- **int** - for integer numbers

- **double** – for real numbers
- **String** – for text variables
- **Date** – for dates
- **boolean** – for boolean variables

You may add [] after the basic type, e.g. String[], to say that this is an array of strings. While it's not important for business users to know this, but these types are valid Java types. Actually, any Java types can be used in the column "Type".

P.S. Instead, of the column "Type" the classic OpenRules used special tables of the type "Datatype", e.g.

Datatype Patient			Datatype DoctorVisit	
String	name		String	encounterDiagnosis
int	age		String	recommendedMedication
double	creatinineLevel		String	recommendedDose
double	creatinineClearance		String	warning
String[]	allergies		String	patientTherapy
double	weight			
String	activeMedication			

You still may use these tables with OpenRules Decision Manager.

Multiple Glossaries

Usually, a business model has one glossary. But if it's too big, you may split it into several tables of the type "Glossary". For example, the sample project "InsurancePremium" contains 3 files "GlossaryClient.xls", "GlossaryDriver.xls", and "GlossaryCar.xls" with separate Glossary tables for glossaryClient, glossaryDriver, and glossaryCar. In the classic OpenRules to use such multiple glossaries you needed to add a special method

```
Method void glossary(Decision decision)
{
    glossaryCar(decision);
    glossaryDriver(decision);
    glossaryClient(decision);
}
```

OpenRules Decision Manager doesn't need this method and will combine all tables of the type "Glossary" automatically.

Optional Glossary Columns

You can add optional columns to the Glossary. The optional columns are:

- **Description** – provides a plain English description of the decision variable. It's always a good practice to have the column "Description" in your glossary
- **Domain** – describes acceptable values (domain) of the decision variable
- **UsedAs** -describes how this decision variable is used: IN, OUT, INOUT, TEMP, CONST.

Here is a typical table Glossary used in the sample project "LoanPreQualification":

Glossary glossary				
Variable	Business Concept	Attribute	Type	Domain
Customer Name	Customer	fullName	String	
Monthly Income		monthlyIncome	double	0-5000000
Monthly Debt		monthlyDebt	double	0-100000
Mortgage Holder		mortgageHolder	boolean	Yes,No
Outside Credit Score		outsideCreditScore	int	0-999
Loan Holder		loanHolder	String	Yes,No
Credit Card Balance		creditCardBalance	double	-1000000 - 100000000
Education Loan Balance		educationLoanBalance	double	-1000000 - 100000000
Internal Credit Rating		internalCreditRating	String	A,B,C,D,F
Internal Analyst Opinion		internalAnalystOpinion	String	High,Mid,Low
Amount	LoanRequest	amount	int	100-10000000
Term		term	int	36,48,60,72
Purpose		purpose	String	
Total Income		totalIncome	double	0-500000
Total Debt		totalDebt	double	0-500000
Income Validation Result		incomeValidationResult	String	SUFFICIENT,UNSUFFICIENT,?
Debt Research Result		debtResearchResult	String	High,Mid,Low,?
Loan Qualification Result		loanQualificationResult	String	QUALIFIED, NOT QUALIFIED, ?

DECISION MODEL TESTING

OpenRules provides all necessary tools to build, test, and debug your business decision models. The same people (subject matter experts) who created decision models can create test cases for these models using simple Excel tables or objects coming from the outside world (from Java, XML, or JSON). You’ve already seen test cases in the introductory [example](#). Now we will explain how to create and use test cases.

Building Test Cases

You can use predefined OpenRules tables of the types “Data” and “DecisionTest” to create executable test cases for your decision models.

Data Tables

First, you need to create test-instances for every business concept used in your Glossary. For example, the sample project “LoanPreQualification” uses the [above glossary](#) with two business concepts Customer and LoanRequest. We can use a table of the type “Data” to create an array of test-customers:

Data Customer customers									
fullName	monthlyIncome	monthlyDebt	mortgageHolder	outsideCreditScore	loanHolder	creditCardBalance	educationLoanBalance	internalCreditRating	internalAnalystOpinion
Borrower Full Name	Monthly Income	Monthly Debt	Mortgage Holder	Outside Credit Score	Loan Holder	Credit Card Balance	Education Loan Balance	Internal Credit Rating	Internal Analyst Opinion
Peter N. Johnson	5000	2300	Yes	720	No	2500.78	0	Mid	Low
Mary K. Brown	4300	2800	No	620	No	5654.33	23800	Mid	Low
Robert Cooper Jr.	6400	2800	Yes	735	Yes	1200	0	Hi	Mid

The first (signature) row contains the keyword “Data” following by the name of the business concept “Customer” and the name of this array “customers”. All 3 words should be separated by spaces. The second row contains the names of attributes included in this business concept, and the 3rd column contains the names of the corresponding decision variables. The order of columns doesn’t have to follow the order of the decision variables in the Glossary. Then you may have as many test-instances as you wish with different test values for all attributes. In this case, we have 3 different customers.

Similarly, the next Data table defined an array of loanRequests:

Data LoanRequest loanRequests							
amount	purpose	term	incomeValidationResult	debtResearchResult	loanQualificationResult	totalIncome	totalDebt
Loan Amount	Loan Purpose	Loan Term	Income Validation Result	Debt Research Result	Loan Qualification Result	Total Income	Total Debt
30000	Home Improvement	72	?	?	?	0	0
15000	Education	36	?	?	?	0	0
55000	Education	24	?	?	?	0	0

In situations when your business concept has too many attributes, you may use Excel Copy + Paste Special + Transpose to switch to the proper horizontal format of the same Data table:

Data LoanRequest loanRequests				
amount	Loan Amount	30000	15000	55000
purpose	Loan Purpose	Home Improvement	Education	Education
term	Loan Term	72	36	24
incomeValidationResult	Income Validation Result	?	?	?
debtResearchResult	Debt Research Result	?	?	?
loanQualificationResult	Loan Qualification Result	?	?	?
totalIncome	Total Income	0	0	0
totalDebt	Total Debt	0	0	0

Test Cases

You may build and test your decision model using only Data tables. However, when you want OpenRules to automatically compare the execution results with the results you expected for every test case, you need to define one more table that has type “DecisionTest”. Here is an example of such a table for the same project “LoanPreQualification”:

DecisionTest testCases					
#	ActionUseObject	ActionUseObject	ActionExpect	ActionExpect	ActionExpect
Test ID	Customer	LoanRequest	Income Validation Result	Debt Research Result	Loan Qualification Result
Test 1	customers[0]	loanRequests[0]	SUFFICIENT	Low	NOT QUALIFIED
Test 2	customers[1]	loanRequests[1]	SUFFICIENT	Low	NOT QUALIFIED
Test 3	customers[2]	loanRequests[2]	SUFFICIENT	Mid	QUALIFIED

The columns of the type “**ActionUseObject**” refer to the test-instances and the columns of the type “**ActionExpect**” refer to the expected results. In the above table “testCases” the first column of the type “#” simply identifies different test-case – these IDs will be used in the execution protocol. The first test case “Test 1” in the second column contains “customers[0]” that refers to the very first element of the above array “customers”. The second column of this test-case contains “loanRequests[0]” that refers to the very first element of the above array “loanRequests”. This column has type “ActionExpect” and specifies an expected value “SUFFICIENT” for the business concepts “LoanRequest” (defined in the third row). You may guess how other test-cases and columns are specified. If during the decision model execution the actually calculated value is different from the expected value the error will be reported.

More Complex Test Cases

OpenRules Data tables allow you to create more complex test cases with inter-object references when some test instances refer to other test-instances. For example, the Data table for the type “Department” may refer to specific employees defined in the data table for the type “Employee” – see how it’s done in the sample project “AggregatedValues”.

Here we will consider a more complex decision model “InsurancePremium” that calculates insurance premiums for different clients which may include multiple drivers and cars. The glossary for the business concept “Client” refers to arrays of drivers of the type Driver[] and an array of cars of the type Car[].

Here is the fragment of the array “drivers”:

Data Driver drivers							
name	Name	Sara Klaus	Spenser Klaus	Mark Houston	Angie Houston	Ray Meno	Shane Meno
gender	Driver Gender	Female	Male	Male	Female	Male	Male
maritalStatus	Marital Status	Single	Single	Single	Single	Married	Single
state	State	AR	AR	AR	AR	CA	CA
age	Driver Age	38	17	38	17	45	21
ageCategory	Driver Age Category	?	?	?	?	45	?

And here is the fragment of the array “cars”:

Data Car cars				
name	Name	Honda Odyssey	Toyota Camry	VW Bug
year	Year	2005	2002	1965
price	Auto Price	39000	12000	1500
convertible	Auto is Convertible	FALSE	FALSE	TRUE
newCar	Car is New	FALSE	FALSE	FALSE

Out Data table for the business concept “Client” should refer to specific drivers and cars defined in the arrays “drivers” and “cars”. Here is how it can be done:

Data Client clients								
name		Name	Client 1		Client 2		Client 3	
drivers	>drivers	Drivers	Sara Klaus	Spenser Klaus	Mark Houston	Angie Houston	Ray Meno	Shane Meno
cars	>cars	Cars	Honda Odyssey	Toyota Camry	Honda Odyssey	Toyota Camry	Honda Odyssey	VW Bug
autoPremium		Client Auto Premium	0	0	0	0	0	0
driverPremium		Client Driver Premium	0	0	0	0	0	0
basePremium		Client Base Premium	0	0	0	0	0	0
totalPremium		Client Total Premium	0	0	0	0	0	0
eligibilityScore		Client Eligibility Score	0	0	0	0	0	0
eligibilityRating		Client Eligibility Rating	?	?	?	?	?	?
longTermClient		Long Term Client	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
segment		Client Segment	Preferred	Preferred	Regular	Regular	Elite	Elite

You may notice that this Data table contains an additional column between columns for decision variable attributes and their names. This column contains “>drivers” for the decision variable “Drivers”. It tells OpenRules that the names such as “Sara Klaus” and “Spenser Klaus” in the columns for the “Client 1” are actually references to the array “drivers”. Similarly the reference “>cars” for the decision variable “Cars” allows OpenRules to interpret the names “Honda Odyssey” and “Toyota Camry” as references to objects in the array “cars”. The only important limitation is that the refereed names inside arrays should be specified in the very first columns and to have type String.

Note. Instead of defining test data in Excel, you may read data from relational databases by using special tables of the type “DataSQL” – see <http://RuleDB.com>.

Building and Running Decision Model

Configuration File “project.properties”

After you complete the design of your decision model and its test cases, you need to adjust the standard file “project.properties”. An example of such a file was provided for the introductory model as follows:

```
model.file="rules/DecisionModel.xls"  
test.file="rules/Test.xls"
```

Usually, you need only two properties:

- **model.file** – it is usually the file “DecisionModel.xls” that describes the structure of your model in the Environment table
- **test.file** – the name of the file that contains your test cases (it could be omitted if using test your model directly from Java)

There could be several optional properties:

- **run.class** – the name of a Java class that will be used instead of the standard OpenRules class; see an example in the project “HelloJava”;
- **trace=On/Off** – to show/hide all executed rules in the execution protocol;

- **report=On/Off** – to generate or not the HTML-reports that show all executed rules (and only them) with explanations why they were executed;
- **debug=On/Off** – to turn on/off the debug mode.

More properties could be added for different deployment options.

Build and Run

To build and run your decision model, you need to double-click on the standard file “**test.bat**”. If you run it for the first time or made any changes in your decision mode, first it will build your model. During the “build” OpenRules will do the following:

- analyze the decision model for errors and consistency
- if everything is OK, it will automatically generate Java code for your model used for testing and execution
- if OpenRules finds errors in your design, it will show them in red in the execution protocol.
- Runs the generated code against your test cases.

Error Reporting

OpenRules is trying to find as many errors as possible in your decision model and report them in friendly business terms. For example, let’s get back to the introductory decision model “VacationDays” and make a mistake in the decision table

“CalculateVacationDays” by omitting a space in the name of the decision variable

“Eligiblefor Extra 5 Days”. OpenRules will catch the error and will show it as follows:

```

Output folder = C:\_GitHub\openrules.samples\VacationDays\target\generated-sources
package.name  = vacation.days
model.name    = DecisionModelVacationDays
goal.name     = Vacation Days
model.file    = rules/DecisionModel.xls
test.file     = rules/Test.xls
FEEL         = On
aws.lambda    = On
precision    = 0.0000000001
report.folder = reports

[ERROR] Failed to execute goal com.openrules:openrules-plugin:8.1.0-SNAPSHOT:generate
(default-generate) on project VacationDays: Decision Model build failed: [If] condit
ion variable 'Eligiblefor Extra 5 Days' not found at file:/C:/_GitHub/openrules.samp
les/VacationDays/rules/Rules.xls?sheet=Vacation%20Days&cell=B4 -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the
following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
Press any key to continue . . . _

```

We highlighted the error message. As you can see OpenRules reports that the variable “Eligiblefor Extra 5 Days” not found and points to the exact place in your Excel files where the error occurred. It is a very important feature of OpenRules, as the generated Java code keeps track of the original Excel tables, and produces all messages and explanations in the business terms used by the decision model author in Excel.

P.S. The generated Java code will be used internally to deploy and execute your model. As a business analyst, you even don’t have to look at them or to know where they are located. Nobody ever should modify the generated files as they will be automatically re-generated when you modify your decision model.

Testing Decision Model

You can fix the error in the file Rules.xls by adding a space between the words “Eligible” and “for” and double-click on “**run.bat**” again. It will re-build your decision model and execute it against all test-cases. Here is an execution protocol (for the first test case only):

```

Execute 'VacationDays'
  SetEligibleForExtra5Days #4 (B8:D8)
    THEN 'Eligible for Extra 5 Days' = false
    Variables:
      Eligible for Extra 5 Days: false

  SetEligibleForExtra3Days #3 (B7:D7)
    THEN 'Eligible for Extra 3 Days' = false
    Variables:
      Eligible for Extra 3 Days: false

  SetEligibleForExtra2Days #1 (B5:D5)
    IF 'Years of Service' Is [15..30)
    THEN 'Eligible for Extra 2 Days' = true
    Variables:
      Eligible for Extra 2 Days: false --> true
      Years of Service: 29

  CalculateVacationDays #1 (B5:F5)
    THEN 'Vacation Days' = 22
    Variables:
      Vacation Days: 0 --> 22

  CalculateVacationDays #4 (B8:F8)
    IF 'Eligible for Extra 5 Days' Is false
    AND 'Eligible for Extra 2 Days' Is true
    THEN 'Vacation Days' + 2
    Variables:
      Eligible for Extra 2 Days: true
      Eligible for Extra 5 Days: false
      Vacation Days: 22 --> 24

Test 'Test D' completed OK. Elapsed time 44.25 ms

```

The protocol shows all executed actions and their results. Along with the execution protocol, “test.bat” also produces the explanation reports in the folder “**report**” using a friendly HTML format. It shows all executed rules and values of the involved decision variables in the moment of execution – see the above [example](#).

Debugging Decision Model

You may debug your decision models while they are being executed. To do that, you need to add the property “**debug=On**” to your file “project.properties” and double-click on “run.bat”. The debugger will stop execution after executing the first selected rule and you can analyze the current content of all decision variables to understand why certain rules were executed or skipped. After you push “Enter”, the next selected rule will be executed. You may continue to push “Enter” until all satisfied rules are executed one-by-one.

Alternatively, you may enter

```
>table <name>
```

```
>go
```

and all rules will be executed until debugger reaches the table “name”. You may analyze all variables before and after table/rule execution.

Let’s consider how you can debug the introductory decision model “Vacation Days”.

After a double-click on “**run.bat**” in the folder “VacationDays”, you will see this protocol at your console:

```
OpenRules Decision Manager, Release 8.1.0 (build of 2020-01-07) - DecisionTest

model.name    = DecisionModelVacationDays
goal.name     = Vacation Days
package.name  = vacation.days
test.file     = file:rules/Test.xls
debug         = On
=====
OpenRules Decision Manager v.8.1.0
Licensed to "OpenRules"
Evaluation period expires on February 7, 2020
Copyright (C) 2004-2020 OpenRules, Inc.
=====

*** Decision DecisionModelVacationDays initialized ***
Read all tests cases in 7 ms

RUN TEST: Test A 2020-01-09T15:24:23.24
Decision DecisionModelVacationDays started
>
```

The debugger will wait for you typing something after “>”. If you just push “**Enter**” or type “**next**”, the next rule will be executed and will show:

- the rule and table names
- the rule text
- all variables involved in the rule and their values after the rules execution.

Here is an example:

```
>
Executed rule# 1 from table SetEligibleForExtra5Days cells file:/C:/_GitHub/openrules.samples/VacationDays/rules/Rules.xls?sheet=Extra%205%20Days&range=B5:D5
  IF Age in Years < 18 THEN Eligible for Extra 5 Days true
  Age in Years=17
  Eligible for Extra 5 Days=true
>_
```

If you enter “*” or “vars” you will see the values of all decision variables as below:

```

>
Executed rule# 1 from table SetEligibleForExtra5Days cells file:/C:/_GitHub/openrules.samples/VacationDays/rules/Rules.xls?sheet=Extra%205%20Days&range=B5:D5
  IF Age in Years < 18 THEN Eligible for Extra 5 Days true
  Age in Years=17
  Eligible for Extra 5 Days=true
>*
  Age in Years=17
  Eligible for Extra 2 Days=false
  Eligible for Extra 3 Days=false
  Eligible for Extra 5 Days=true
  Id=A
  Vacation Days=0
  Years of Service=1
>_

```

You may continue to push “Enter”, to see and analyze other executed rules. You may enter “>go” or “>end” to run all rules to the very end.

To interrupt the execution process at any moment you may enter the command “>q” or “>quit”.

Let’s start again by double-click on “run.bat”. This time you want to debug only the table “CalculateVacationDays”. To do this, you may enter the commands:

>table CalculateVacationDays

>go

The debugger will show all executed rules (without stopping at them) and will stop only before evaluating the table “CalculateVacationDays”. You may display all variables at this moment by typing “>vars”. Here is what you will see:

```

Decision DecisionModelVacationDays started
>table CalculateVacationDays
>go
Executed rule# 1 from table SetEligibleForExtra5Days cells file:/C:/_GitHub/openrules.samples/VacationDays/rules/Rules.xls?sheet=Extra%205%20Days&range=B5:D5
  IF Age in Years < 18 THEN Eligible for Extra 5 Days true
  Age in Years=17
  Eligible for Extra 5 Days=true
Executed rule# 3 from table SetEligibleForExtra3Days cells file:/C:/_GitHub/openrules.samples/VacationDays/rules/Rules.xls?sheet=Extra%203%20Days&range=B7:D7
  THEN Eligible for Extra 3 Days false
  Eligible for Extra 3 Days=false
Executed rule# 3 from table SetEligibleForExtra2Days cells file:/C:/_GitHub/openrules.samples/VacationDays/rules/Rules.xls?sheet=Extra%202%20Days&range=B7:D7
  THEN Eligible for Extra 2 Days false
  Eligible for Extra 2 Days=false
Evaluating table CalculateVacationDays
>

```

As you can see, several rules were already executed and the variables “Eligible for Extra 5 Days”, “Eligible for Extra 3 Days”, and “Eligible for Extra 2 Days” received values true, false, and false correspondingly. It already can prompt you to how the rules inside

this multi-hit decision table “CalculateVacationDays” will be evaluated.

Let’s assume that you want to stop before some rule inside this table being evaluated, for example before the second rule described in cells “B6:F6”. You can enter the command “>rule 2” or “>rule B6:F6” or even simply “>rule B6“. After you enter “>go“, you will see:

```
>rule 2
>go
Evaluating rule 2 from table CalculateVacationDays
  IF Eligible for Extra 5 Days false AND Eligible for Extra 2 Days true THEN Vacation
Days += 2
  Eligible for Extra 2 Days=false
  Eligible for Extra 5 Days=true
  Vacation Days=0
>
```

Now you may continue to push “Enter” and see all executed rules. Here is the final view for this test case:

```
Evaluating table CalculateVacationDays
>rule 2
>go
Evaluating rule 2 from table CalculateVacationDays
  IF Eligible for Extra 5 Days false AND Eligible for Extra 2 Days true THEN Vacati
on Days += 2
  Eligible for Extra 2 Days=false
  Eligible for Extra 5 Days=true
  Vacation Days=0
>
Executed rule# 1 from table CalculateVacationDays cells file:/C:/_GitHub/openrules.
samples/VacationDays/rules/Rules.xls?sheet=Vacation%20Days&range=B5:F5
  THEN Vacation Days = 22
  Vacation Days=22
>
Executed rule# 2 from table CalculateVacationDays cells file:/C:/_GitHub/openrules.
samples/VacationDays/rules/Rules.xls?sheet=Vacation%20Days&range=B6:F6
  IF Eligible for Extra 5 Days true THEN Vacation Days += 5
  Eligible for Extra 5 Days=true
  Vacation Days=27
>
Decision DecisionModelVacationDays completed
Validating results for the test <Test A>
Test A was successful
Executed test Test A in 501318 ms

RUN TEST: Test B 2020-01-09T15:41:59.712
Decision DecisionModelVacationDays started
>
```

Then you may continue to debug the second test case or just enter “>go” or “>end“ to finish debugging.

At any moment you may enter the command “>help” you will see a summary of all available commands:


```
>help
Debugger commands:
<Enter> or next - execute the next rule and pause
table <name> - stop at the beginning of this table
rule <row> - stop before evaluating rule at this row, e.g. >rule 85 or >rule 2
* or vars - show all variables and their current values
go or end - run without stops until a selected table or end
q - quit
h or help - show help commands
```

Thus, using these commands you may successfully debug your business decision model.

DECISION MODEL DEPLOYMENT

OpenRules provides all the necessary facilities to simplify the integration of business decision models with modern enterprise-level applications. Tested decision models may be easily deployed on-premise or on-cloud.

Decision Model Execution Using Java API

After you build and test your decision model, it is ready to be incorporated in any Java-based application using a simple Java API internally generated for this decision model. When you talk to your software developers, you may point them to a simple Java launcher for the introductory decision model “VacationDays” that is located in the file VacationDays/src/test/java/vacation.days/Main.java. This launcher can execute your business decision model by passing to it a Java object customer that can come from a database, JSON files, or any other source. Here is the source code for the launcher:

```
public static void main(String[] args) {
    .... DecisionModel model = new DecisionModelVacationDays();
    .... Goal goal = model.createGoal();
    ....
    .... Employee employee = new Employee ();
    .... employee.setId("Mary Grant");
    .... employee.setAge(46);
    .... employee.setService(18);
    .... goal.use("Employee", employee);
    .... goal.execute();
    .... System.out.println("Vacation Days = " + employee.getVacationDays());
}
```

The classes `com.openrules.core.DecisionModel` and `com.openrules.core.Goal` are the standard OpenRules classes included in the automatically installed OpenRules jar-files. The class `vacation.days.DecisionModelVacationDays` was automatically generated when you built your decision model. Three underlying methods

- `model.createGoal(goal-name)`
- `goal.use(object)`
- `goal.execute()`

provide a simple and intuitive invocation API for any business decision model. The developers can use the provided file “jar.bat” to pack your decision model in a jar-file and call it from any Java application similar to the above launcher.

Decision Model as an AWS Lambda Function

A business analyst may deploy a tested decision model on Amazon Web Services (AWS) cloud with one click without involving software developers. To do AWS deployment, you should already have an active [AWS account](#) and define your security [credentials](#) (access key ID and secret access key) which should not be shared with anybody.

Let’s consider how to deploy the introductory decision model “VacationDays” as [AWS Lambda function](#). Look at the standard project “**VacationDaysLambda**” to learn how to do it:

- Add additional properties to the configuration file “project.properties”:

```
model.file="../VacationDays/rules/DecisionModel.xls"
test.file="../VacationDays/rules/Test.xls"
trace=On
report=On
model.package=vacation.days.lambda

#deployment properties
aws.lambda=On
aws.lambda.bucket=openrules-demo-lambda-bucket
aws.lambda.region=us-east-1
aws.api.stage=test
```

Note that this project uses the same rules repository that was created in the project “./VacationDays”. The property “*aws.lambda.bucket*” defines a new or existing AWS S3 bucket name.

- Double-click on the provided file “**deployLambda.bat**”.

The decision model will be deployed, and the console log will show the invoke URL for the deployed decision service (highlighted in the following example):


```
[INFO] --- openrules-plugin:8.1.0-SNAPSHOT:deployLambda (default-cli) @ VacationDaysLambda ---
DecisionModel Lambda upload started. File C:\_GitHub\openrules.samples\VacationDaysLambda\target\Vaca
tionDaysLambda-1.0-SNAPSHOT.lambda.zip
DecisionModel Lambda upload complete
Updated lambda DecisionModelVacationDays
Created API Gateway deployment DecisionModelVacationDays for stage [test]
Decision Service DecisionModelVacationDays successfully deployed.

Invoke URL: https://ab2gxc0cbh.execute-api.us-east-1.amazonaws.com/test/vacation-days

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 13.727 s
[INFO] Finished at: 2020-01-18T16:54:49-05:00
[INFO] -----
```

If you previously deployed your model as a Lambda function, it is possible that sometimes you will receive the following error:

“AWS lambda deployment failure: The statement id (gw-lambda) provided already exists. Please provide a new statement id, or remove the existing statement.”

In this case, you need to run the provided “undeployLambda.bat” and try again.

After successful AWS Lambda deployment, OpenRules generates the file “**testLambda.bat**” that already includes the above invoke URL. When you click on this file, it remotely executes all test cases defined in the Excel file “Test.xls” against just deployed AWS Lambda! You will see that a pure execution time on the AWS cloud for any test is less than 1 millisecond, while based on your connection speed the round-trip time with sending a request and receiving a response takes on average around 40 milliseconds.

Optionally, you may use additional settings to specify more details about your AWS deployment preferences. For example, by default, we use the word “test” (inside the invoke URL) as your deployment stage. But you may redefine it as “dev”, “prod”, or any similar word using this setting:

```
aws.api.stage=test
```

By default, your deployed service will be publicly accessible, but it is possible to make it private. Usually, people keep their AWS Credentials (access key ID and secret access key) in the default file `~/.aws/credentials`. However, alternatively, you may define them directly in the file “project.properties”:

```
aws.access.key.id=<aws-access-key-id>
aws.secret.access.key=<aws-secret-access-key>
```

If you decide to undeploy your decision service and clean up your AWS, just click on the file “**undeployLambda.bat**”.

Decision Model as a RESTful Web Service

The sample project “**VacationDaysSpringBoot**” demonstrates how to deploy your business decision model as a RESTful web service using [SpringBoot](#). It converts our business decision model “VacationDays” into a Web Service that can accept HTTP requests at <http://localhost:8080/vacations-days> and will respond with proper responses in the JSON format.

This project extends the project “VacationDays” and uses the same rules repository “rules”. First, we modified the configuration file “project.properties” as follows:

```
model.name=DecisionModelVacationDays
goal.name="Vacation Days"
model.file=../VacationDays/rules/DecisionModel.xls
test.file=../VacationDays/rules/Test.xls
package.name=vacation.days.spring

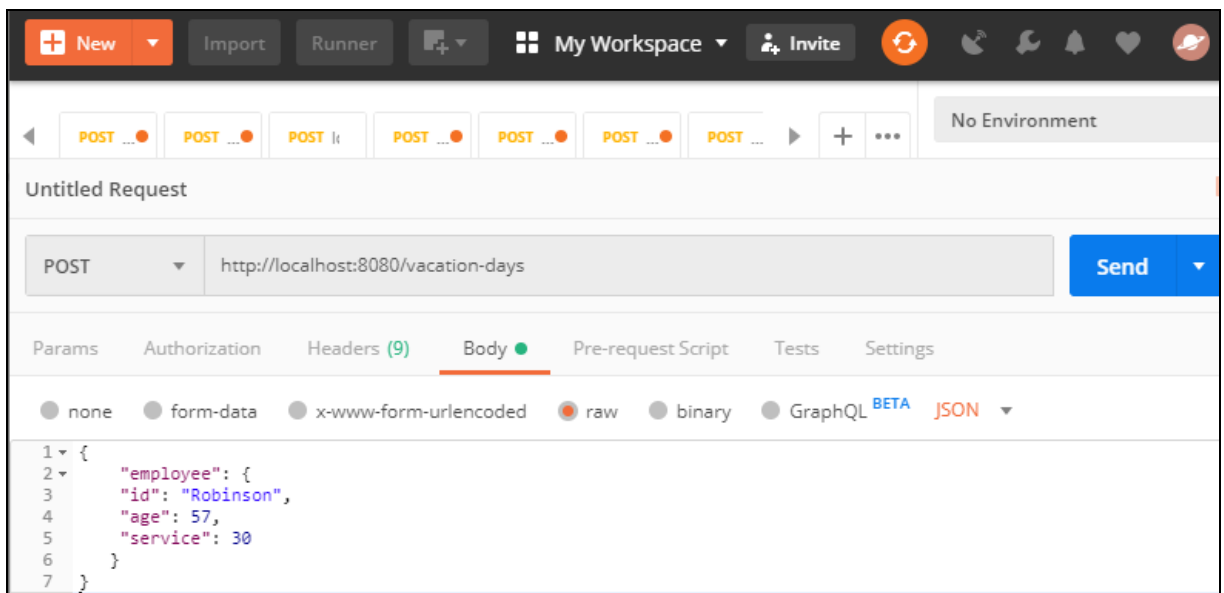
#deployment properties
spring.boot=On
```

Note that we refer to the model.file and test.file with the prefix “../VacationDays”. The property “*spring.boot=On*” specifies SpringBoot-based deployment.

The project includes one additional file “**runLocalServer.bat**”. When you double-click on it for the first time, it will install all necessary jar files (such as SpringBoot jars). Then it will build our model in such a way that it can be deployed as a RESTful web service on the local server. After a successful start of the sever you should see a console screen that looks like this one:

[illegible]

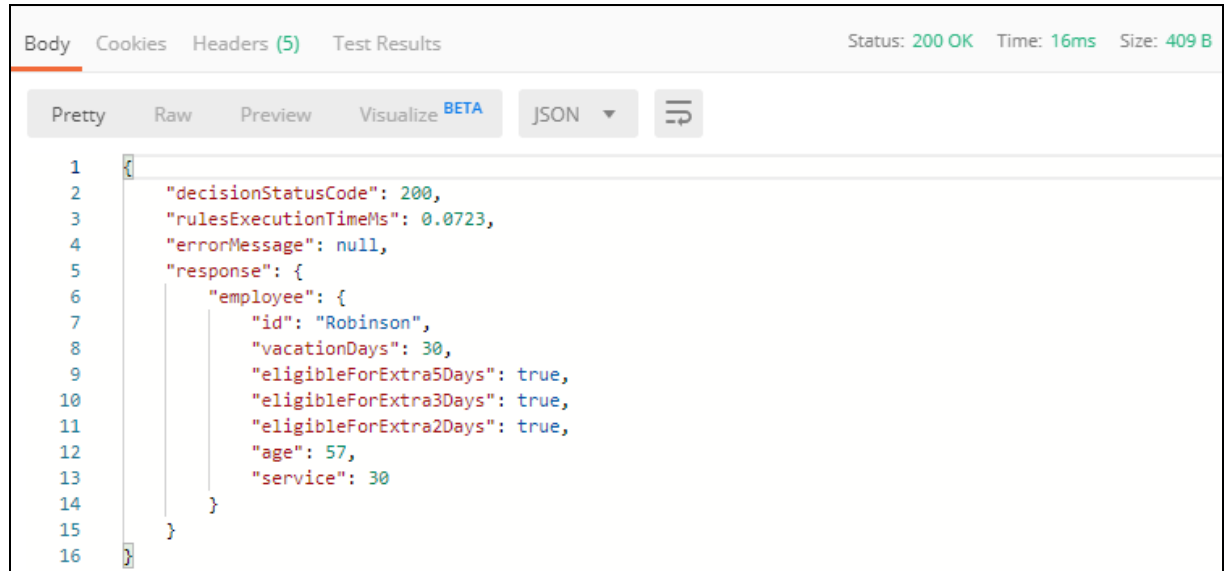
To test this web service you may use the commonly used POSTMAN, that can be downloaded for free from <https://www.getpostman.com/>. After installation and start, you may fill out this POSTMAN's form:



It will post HTTP requests to the service located at <http://localhost:8080/vacations-days>. In the section “Body” you can enter

```
{
  "employee": {
    "id": "Robinson",
    "age": 57,
    "service": 30
  }
}
```

and click on the button “Send”. This request will be sent to our web service “vacation-days” and it will respond with a JSON response



Packaging Decision Models as a Docker Image

We can use the same sample project “**VacationDaysSpringBoot**” to demonstrate how to package our RESTful web service as a [Docker](#) image. This project contains the batch file “**buildDocker.bat**” that does exactly this. Internally it utilizes Google Container Tool “[Jib](#)” that is a [Maven](#) plugin for building Docker images for Java applications. When you run “buildDocker.bat” it will automatically download install all necessary files and will create a Docker image using locally installed [Docker](#).

After that you may switch to a command line and enter

>docker images

It will show all docker images that may look as below:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
vacation-days	latest	1696a40938c7	41 seconds ago	124MB

To start start this Docker container to serve requests on the local port 8081, you need to

enter the following command

```
>docker run -ti -p 8081:8080 vacation-days
```

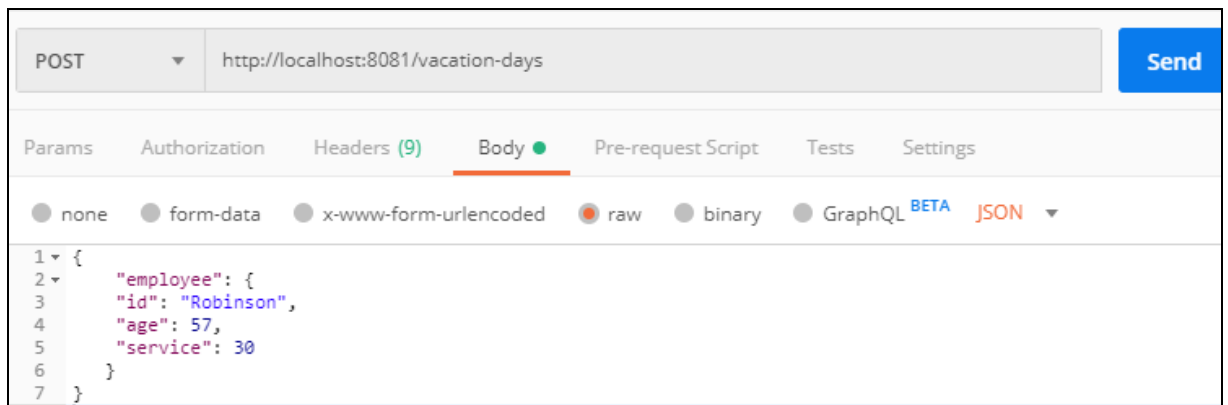
The start of the container will look as follows:

```
2020-01-23 23:30:30,823 main INFO Log4j appears to be running in a Servlet environment, but there's no log4j-web module available. If you want better web container support, please add the log4j-web JAR to your web archive or server lib directory.

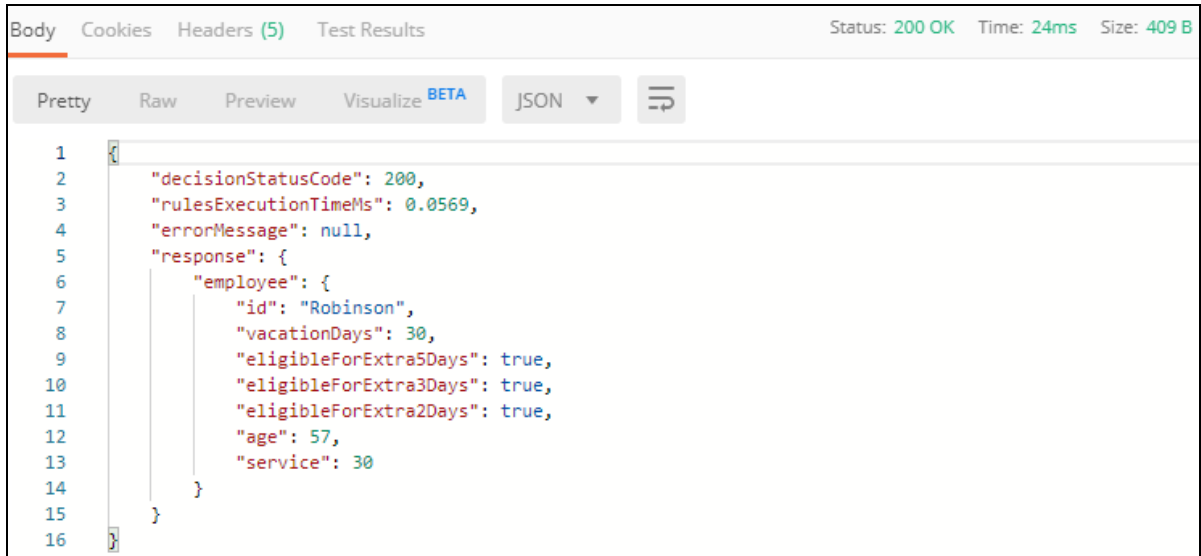
:: Spring Boot :: (v2.2.2.RELEASE)

Jan 23, 2020 11:30:32 PM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-nio-8080"]
Jan 23, 2020 11:30:32 PM org.apache.catalina.core.StandardService startInternal
INFO: Starting service [Tomcat]
Jan 23, 2020 11:30:32 PM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet engine: [Apache Tomcat/9.0.29]
Jan 23, 2020 11:30:32 PM org.apache.catalina.core.ApplicationContext log
INFO: Initializing Spring embedded WebApplicationContext
Jan 23, 2020 11:30:32 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-nio-8080"]
```

This container will wait for requests on the port 8081. Now we can use the same POSTMAN to send a test request:



After you push “Send”, you will receive the expected results:



```

1  {
2    "decisionStatusCode": 200,
3    "rulesExecutionTimeMs": 0.0569,
4    "errorMessage": null,
5    "response": {
6      "employee": {
7        "id": "Robinson",
8        "vacationDays": 30,
9        "eligibleForExtra5Days": true,
10       "eligibleForExtra3Days": true,
11       "eligibleForExtra2Days": true,
12       "age": 57,
13       "service": 30
14     }
15   }
16 }

```

On the console you will see that our decision model was actually executed as a Docker container:

```

Jan 23, 2020 11:31:50 PM org.apache.catalina.core.ApplicationContext
INFO: Initializing Spring DispatcherServlet 'dispatcherServlet'
=====
OpenRules Decision Manager v.8.1.0
Licensed to "OpenRules"
Evaluation period expires on February 19, 2020
Copyright (C) 2004-2020 OpenRules, Inc.
=====
*** Decision DecisionModelVacationDays initialized ***

```

So, this example demonstrates how OpenRules-based decision model can be deployed as a Docker container and be executed locally. Now it is ready to be deployed to any of the following container registries:

- Google Container Registry (GCR)
- Amazon Elastic Container Registry (ECR)
- Docker Hub Registry
- Azure Container Registry (ACR).

It can be done by your software developers following this [manual](#).

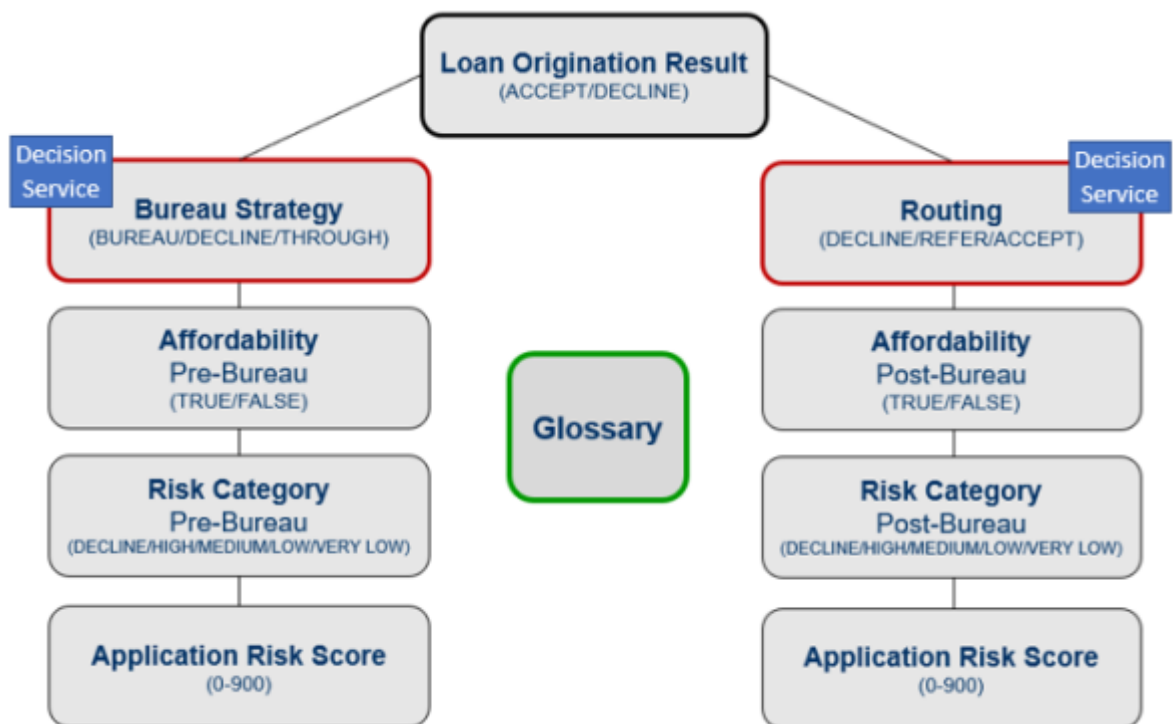
RULES-BASED SERVICE ORCHESTRATION

OpenRules provides business users with abilities to build and deploy operational decision microservices. It empowers business users with an ability to assemble new decision services by orchestrating existing decision services independently of how they were built

and deployed. The service orchestration logic is a business logic too, so it's only natural to apply the decision modeling approach to orchestration. To orchestrate different services you may create a special **orchestration decision model** that describes under which conditions such services should be invoked and how to react to their execution results.

OpenRules decision tables have special action-columns of the type “**ActionExecute**” that is usually used to execute different services upon certain conditions without worrying how they were implemented and deployed. To describe such external services OpenRules added a special new table “**DecisionService**“. You may [download a special workspace “openrules.loan”](#) that implements a library of decision services described in the [Loan Origination example](#) from the [DMN](#) Section 11.

The workspace “openrules.loan” contains several decision models with two main goals “BureauStrategy” and “Routing” deployed as external decision services:



The high-level goal “Loan Origination Result” is an example of the orchestration decision models. The orchestration logic here is relatively simple:

Execute decision service “BureauStrategy” that should determine the goal “Bureau Strategy”. If Bureau Strategy is DECLINE, then set Loan Origination Result to DECLINE, and stop. If Bureau Strategy is not DECLINE, then execute decision service “Routing” that will determine the goal “Routing”. If Routing is DECLINE, then set Loan Origination Result to

DECLINE. If Routing is REFER, then set Loan Origination Result to REFER. If Routing is ACCEPT, then set Loan Origination Result to ACCEPT.

This logic can be naturally presented in the following table:

Decision LoanOriginationResult					
Condition		Condition		ActionExecute	Action
Bureau Strategy		Routing		Execute	Loan Origination Result
				BureauStrategyService	
Is	DECLINE				DECLINE
Is Not	DECLINE			RoutingService	
Is Not		Is	DECLINE		DECLINE
Is Not		Is	REFER		REFER
Is Not		Is	ACCEPT		ACCEPT

Here the third column “ActionExecute” may execute two decision services: “BureauStrategyService” and “RoutingService”. The actual implementation of these services is described in the following table:

DecisionService decisionServices		
Service Name	Service Type	Service Endpoint
BureauStrategyService	REST	https://bfsu86u7u6.execute-api.us-east-1.amazonaws.com/test/bureau-strategy
RoutingService	REST	https://f7b53vrel.execute-api.us-east-1.amazonaws.com/test/routing

The column “Service Type” defines these services as REST web services and provides their endpoints – in this particular case both services were deployed as AWS Lambda functions, the default OpenRules deployment destination (it was done with an instant click). The table “DecisionService” may have the 4th (optional) column

Business Objects
Applicant,Application,RequestedProduct
Applicant,Application,RequestedProduct,BureauData

that describes the parameters of each service that correspond to the business concepts defined in the common Glossary. If the column “Business Objects” is omitted (like in the above table), all business objects will be passed to all decision services even if “BureauStrategyService” doesn’t need BureauData.

Along with REST web services, OpenRules supports other types of services. For example, you may get essentially faster execution by taking advantage of the fact that your services are deployed as AWS Lambdas by using their ARN addresses as endpoints:

DecisionService decisionServices		
Service Name	Service Type	Service Endpoint
BureauStrategyService	AWS Lambda	arn:aws:lambda:us-east-1:395608014566:function:BureauStrategy
RoutingService	AWS Lambda	arn:aws:lambda:us-east-1:395608014566:function:Routing

If your decision services are deployed as AWS Lambda functions you even don't have to provide the complete ARN addresses, and can simply write their names as in the following table:

DecisionService decisionServices		
Service Name	Service Type	Service Endpoint
BureauStrategyService	AWS Lambda	BureauStrategy
RoutingService	AWS Lambda	Routing

OpenRules will automatically expand the name like “BureauStrategy” to “arn:aws:lambda:us-east-1:395608014566:function:BureauStrategy”.

Another supported type of services is regular Java classes automatically generated by OpenRules from decision models:

DecisionService decisionServices		
Service Name	Service Type	Service Endpoint
BureauStrategyService	DecisionModel	loan.origination.bureaustrategy.BureauStrategy
RoutingService	DecisionModel	loan.origination.routing.Routing

You also may invoke any static Java method, e.g. use Service Type “JavaMethod” and Service Endpoint “loan.origination.EmailService:send” to send an automatically generated email to the Applicant.

TECHNICAL SUPPORT

Direct all your technical questions to support@openrules.com or to this [Discussion Group](#). Read more at <http://openrules.com/services.htm>.