OpenRules®

Open Source Business Decision Management System
Release 6.1

User Manual

OpenRules, Inc.

www.openrules.com

September-2011

Table of Contents

INTRODUCTION

OpenRules® was developed in 2003 by OpenRules, Inc. as an open source Business Rules Management System (BRMS) and since then has become one of the most popular BRMS on the market. Today OpenRules® is a Business Decision Management System (BDMS) with proven records of delivering and maintaining areliable decision support software. OpenRules® is a winner of several impressive software awards for innovation and is used worldwide by multi-billion dollar corporations, major banks, insurers, health care providers, government agencies, online stores, universities, and many other institutions.

Brief History

From the very beginning, OpenRules® was oriented to subject matter experts (business analysts). allowing them to work in concert with software developers to create, maintain, and efficiently execute business rules housed in enterprise-class rules repositories. OpenRules® avoided the introduction of yet another "rule language" as well as another proprietary rules management GUI. Instead, OpenRules® relied on commonly used tools such as MS Excel, Google Docs and Eclipse. This approach enabled OpenRules® users to create and maintain inter-related decision tables directly in Excel. Each rule table included several additional rows called a "technical view" where a software developer could use Java snippets to specify the exact semantics of rule conditions and actions.

In March of 2008, OpenRules® Release 5 introduced Rule Templates. Templates allowed a business analyst to create hundreds and thousands of business rules based on a small number of templates supported by software developers. Rule templates minimized the use of Java snippets and hid them from business users. Rule templates was a significant step in minimizing rule repositories and clearly separating the roles of business analysts and software specialists in maintaining the rules.

In March of 2011 OpenRules® introduced Release 6, which finally allowed business users to eliminate the need for IT involvement in creating and maintaining rules repositories. OpenRules® 6 implements a highly successful decision management approach described in the book "The Decision Model" by Barbara von Halle and Larry Goldberg (2009) without IT involvement. This new release effectively removes any Java coding from rules representation and allows business analysts themselves to specify their decision models and supporting rule families directly and completely in Excel. Business users can also create business glossaries and test cases in Excel tables. They may then test the accuracy of execute their decision models without the need for any coding at all. Once a decision model has been tested it can be easily incorporated into any Java or .NET environment. This process may involve IT specialists but only to integrate the business glossary with a specific business object model. The business logic remains the complete prerogative of subject matter experts.

OpenRules® Components

OpenRules® offers the following decision management components:

- Rule Repository for management of enterprise-level decision models
- Rule Engine for execution of decision models and different business rules
- Rule Dialog for building rules-based Web questionnaires
- Rule Learner for rules discovery and predictive analytics
- Rule Solver for solving constraint satisfaction and optimization problems
- Finite State Machines for event processing and "connecting the dots".

Integration of these components with executable decision models has effectively converted OpenRules® from a BRMS to a BDMS, Business Decision Management System, oriented to "decision-centric" application development.

OpenRules, Inc. is a professional open source company that provides product documentation and technical support that is highly praised by our customers. You may start learning about product with the document "Getting Started" which describes how to install OpenRules® and includes simple examples. Then you may look at a more complex example in the tutorial "Calculating A Tax Return Using the Decision Model". This user manual covers the core OpenRules® concepts in greater depth. Additional OpenRules® components are described in separate user manuals: see Rule Learner, Rule Solver, and Rule Dialog.

Document Conventions

The regular Century Schoolbook font is used for descriptive information.

The italic Century Schoolbook font is used for notes and fragments clarifying the text.

```
The Courier New font is used for code examples.
```

CORE CONCEPTS

OpenRules® is a BDMS, Business Decision Management System, oriented to "decision-centric" application development. OpenRules® utilizes the well-established spreadsheet concepts of workbooks, worksheets, and tables to build enterprise-level rule repositories. Each OpenRules® workbook is comprised of one or more worksheets that can be used to separate information by types or categories.

To create and edit rules and other tables presented in Excel-files you can use any standard spreadsheet editor such as:

- MS Excel™
- OpenOffice™
- Google Spreadsheets™

Google Spreadsheets are especially useful for collaborative rules management.

OpenRules® supports different types of spreadsheets that are defined by their keywords. Here is the list of OpenRules® tables along with brief description of each:

Table Type (Keyword)	Comment
Decision	Defines a decision that may consist of multiple sub-decisions associated with different rule families
RuleFamily	This is a single-hit decision table that uses multiple conditions on different fact types to reach a single conclusion about the decision fact type
Glossary	For each fact type used in the rule families the glossary defines related business concepts, as well as related implementation attributes and their possible domain
DecisionObject	Associates business concepts specified in the glossary with concrete objects defined outside the decision model (i.e.

	as Java objects or Excel Data tables)
Rules	Defines a decision table that includes Java snippets that specify custom logic for conditions and actions. Read more. Some Rules tables may refer to templates that hide those Java snippets.
Datatype	Defines a new data type directly in Excel that can be used for testing
Data	Creates an array of test objects
Variable	Creates one test object
Environment	This table defines the structure of a rules repository by listing all included workbooks, XML files, and Java packages
Method	Defines expressions using snippets of Java code and known fact types and objects
RuleFamily1	A multi-hit rule family that allows rule overrides
RuleFamily2	A rule sequence that executes all rules in top-down order when results of the execution of previous rules may affect the conditions of rules that follow
Layout	A special table type used by OpenRules® Forms and OpenRules® Dialog

The following section will provide a detailed description of these concepts.

SPREADSHEET ORGANIZATION AND MANAGEMENT

OpenRules® uses Excel spreadsheets to represent and maintain business rules, web forms, and other information that can be organized using a tabular format. Excel is the best tool to handle different tables and is a popular and widely used tool among business analysts.

Workbooks, Worksheets, and Tables

OpenRules® utilizes commonly used concepts of workbooks and worksheets. These can be represented and maintained in multiple Excel files. Each

OpenRules® workbook is comprised of one or more worksheets that can be used to separate information by categories. Each worksheet, in turn, is comprised of one or more tables. Decision tables are the most typical OpenRules® tables and are used to represent business rules. Workbooks can include tables of different types, each of which can support a different underlying logic.

How OpenRules® Tables Are Recognized

OpenRules® recognizes the tables inside Excel files using the following parsing algorithm.

1. The OpenRules® parser splits spreadsheets into “parsed tables”. Each logical table should be separated by at least one empty row or column at the start of the table. Table parsing is performed from left to right and from top to bottom. The first non-empty cell (i.e. cell with some text in it) that does not belong to a previously parsed table becomes the top-left corner of a new parsed table.
2. The parser determines the width/height of the table using non-empty cells as it’s clues. Merged cells are important and are considered as one cell. If the top-left cell of a table starts with a predefined keyword (see the table below), then such a table is parsed into an OpenRules® table.
3. All other "tables," i.e. those that do not begin with a keyword are ignored and may contain any information.

The list of all keywords was described above. OpenRules® can be extended with more table types, each with their own keyword.

While not reflected in the table recognition algorithm, it is good practice to use a black background with a white foreground for the very first row. All cells in this row should be ***merged***, so that the first row explicitly specifies the table width. We call this row the "***table signature***". The text inside this row (consisting of one or more merged cells) is the table signature that starts with a keyword. The information after the keyword usually contains a unique table name and additional information that depends on the table type.

If you want to put a table title before the signature row, use an empty row between the title and the first row of the actual table. Do not forget to put an empty row after the last table row. Here are examples of some typical tables recognized by OpenRules®.

OpenRules® table with 3 columns and 2 rows:

Keyword <some text>		
Something	Something	Something
Something	Something	Something

OpenRules® table with 3 columns and still 2 rows:

Keyword	Something	Something
Something	Something	Something
Something	Something	Something

OpenRules® table with 3 columns and 3 rows (empty initial cells are acceptable):

Keyword <some text>		
Something	Something	
	Something	Something
		Something

OpenRules® table with 3 columns and 2 rows (the empty 3rd row ends the table):

Keyword <some text>		
Something	Something	Something
Something	Something	Something

Something	Something	Something

OpenRules® table with 2 columns and 2 rows (the empty cell in the 3rd column of the title row results in the 4th columns being ignored. This also points out the importance of merging cells in the title row):

Keyword	Something		Something
Something	Something	Something	Something
Something	Something	Something	Something

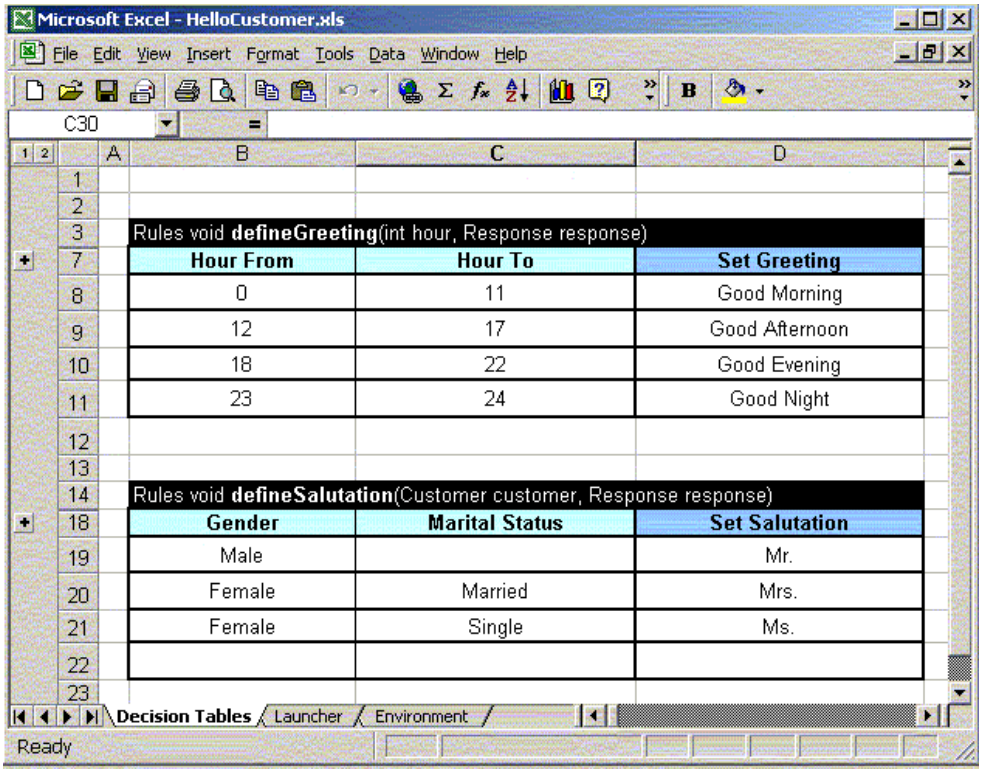
OpenRules® will not recognize this table (there is no empty row before the signature row):

Table Title		
Keyword <some text>		
Something	Something	
	Something	Something
		Something

Fonts and coloring schema are a matter of the table designer's taste. The designer has at his/her disposal the entire presentation power of Excel (including comments) to make the OpenRules® tables more self-explanatory.

OpenRules® Table Example

Here is an example of a worksheet with two rules tables:



This workbook is comprised of three worksheets:

1. Worksheet "Decision Tables" - includes rule tables
2. Worksheet "Launcher" - includes a method that defines an order and conditions under which rules will be executed
3. Worksheet "Environment" - defines the structure of a rules repository by listing all included workbooks, XML files, and Java packages (if any).

The worksheet "Decision Tables" is comprised of two rule tables "defineGreeting" and "defineSalutation". Rule tables are a traditional way to represent business decision tables. Rule tables are decision tables that usually describe combinations of conditions and actions that should be taken when all of the conditions have been satisfied. In the table "defineGreeting", the action "Set Greeting" will be executed when an "hour," passed to this table as a parameter, is between "Hour From" and "Hour To". In the table "defineSalutation", an action "Set Salutation" will be executed when a customer's Gender *and* Marital Status correspond to the proper row.

These tables start with signature rows that are determined by a keyword in the first cell of the table. A table signature in general has the following format:

```
Keyword return-type table-name(type1 par1, type2 par2,..)
```

where `table-name` is a one-word function name and `return-type`, `type1`, and `type 2` are types defined in the current OpenRules® configuration. For example, type may be any basic Java type such as `int`, `double`, `Date`, or `String`.

The rule tables above are recognized by the keyword "`Rules`". All of the columns have been merged into a single cell in the signature rows. Merging cells B3, C3, and D3 specifies that table "defineGreeting" has 3 columns. A table includes all those rows under its signature that contain non empty cells: in the example above, an empty row 12 indicates the end of the table "defineGreeting".

Limitation. Avoid merging rule rows in the very first column (or in the very first row for horizontal tables) - it may lead to invalid logic.

Business and Technical Views

OpenRules® tables such as “Rules” may have two views:

[1] Business View

[2] Technical View

These two views are implemented using Excel's outline buttons `[1]` and `[2]` at the top left corner of every worksheet - see the figure below. This figure represents a business view - no technical details about the implementation are provided. For example, from this view it is hard to tell for sure what greeting will be generated at 11 o'clock: "Good Morning" or "Good Afternoon"? If you push the Technical View button `[2]` (or the button "+" on the left), you will see the hidden rows with the technical details of this rules table:

	A	B	C	D
2				
3		Rules void **defineGreeting**(int hour, Response response)		
4		C1	C2	A1
5		min <= hour	hour <= max	response.map.put("greeting", greeting);
6		int min	int max	String greeting
7		**Hour From**	**Hour To**	**Set Greeting**
8		0	11	Good Morning
9		12	17	Good Afternoon
10		18	22	Good Evening
11		23	24	Good Night

The technical view opens hidden rows 4-6 that contain the implementation details. In particular, you can see that both "Hour From" and "Hour To" are included in the definition of the time intervals. Different types of tables have different technical views.

Note. Using Rules Templates you may completely split business and technical information between different Excel tables. The decision models do not use technical views at all because they do not require any coding.

DECISION MODELING AND EXECUTION

OpenRules® implements a highly popular methodological approach described in the book "The Decision Model". You may become familiar with the major decision modeling concepts from simple examples provided in the document "Getting Started" and several associated tutorials. First we will consider the standard implementation options for The Decision Model, and later on we will describe OpenRules® concepts that go beyond the Decision Model.

Starting with Decision

The Decision Model applies a top-down approach to decision modeling. This means that we usually start with the definition of a Decision and not with rules or data. Only then we will define rule families, a glossary, and then data. Here is an example of a Decision:

Decision DeterminePatientTherapy	
Decisions	**Execute Rule Families**
Define Medication	:= DefineMedication()
Define Dosing	:= DefineDosing()

Here the decision "DeterminePatientTherapy" consists of two sub-decisions:

- "Define Medication" that is implemented using a rule family "DefineMedication"
- "Define Dosing" that is implemented using a rule family "DefineDosing".

The table "Decision" has two columns "Decisions" and "Execute Rule Family". The first column contains the names of all our sub-decisions - here we can use any combinations of words as decision names. The second column contains exact names of rule families that implement these sub-decisions. The names cannot contain spaces or special characters (except for "underscore") and they should always be preceded by ":=", which indicates the rule families will actually be executed by the OpenRules® engine.

OpenRules® allows you to use multiple "Decision" tables to define more complex decisions. For example, a top-level decision that defines a main fact may be defined through several sub-decisions about related facts:

DecisionModel DecisionMain	
Decisions	**Execute**
Define Fact 1	:= RuleFamilyFact1()
Define Fact 2	:= RuleFamilyFact21()
Define Fact 2	:= RuleFamilyFact22()
Define Fact 3	:= DecisionFact3()
Define Fact 4	:= RuleFamilyFact4()

In order to Define Fact 2 it is necessary to execute two rule families. Some decisions, like "Define Fact 3", may require their own separate decision models such as described in the following table:

DecisionModel DecisionFact3	
Decisions	**Execute**
Define Fact 3.1	:= RuleFamilyFact31()
Define Fact 3.2	:= RuleFamilyFact32()
Define Fact 3.3	:= RuleFamilyFact33()

These tables can be kept in different files and can be considered as building blocks for your decision models. This top-down approach with Rule Families and dependencies between them allows you to represent your decision logic in an intuitive, easy to understand way.

Some decisions may have a more complex structure than the just described sequence of sub-decisions. You can even use conditions inside decision tables. For example, consider a situation when the first sub-decision validates your data and a second sub-decision executes complex calculations but only if the preceding validation was successful. Here is an example of such a decision from the tax calculation tutorial:

Decision Apply1040EZ			
Condition		ActionPrint	ActionExecute
1040EZ Eligible		**Decisions**	**Execute**
		Validate	:= ValidateTaxReturn(decision)
Is	TRUE	Calculate	:= DetermineTaxReturn(decision)
Is	FALSE	Do Not Calculate	

Since this Decision table uses an optional column “Condition”, we have to add a second row. The keywords “Condition”, “ActionPrint”, and “ActionExecute” are defined in the standard OpenRules® template “DecisionTemplate” – see the configuration file “DecisionTemplates.xls” in the folder “openrules.config”. This table uses a fact type “1040EZ Eligible” that is defined by the first (unconditional) sub-decision “Validate”. We assume that the decision “ValidateTaxReturn” should set this fact type to TRUE or FALSE. Then the second sub-decision “Calculate” will be executed only when “1040EZ Eligible” is TRUE. When it is FALSE, this decision, “Apply1040EZ”, will simply print “Do Not Calculate”. In our example the reason will be printed by the rule family “ValidateTaxReturn”.

Note. You may use many conditions of the type "Condition" defined on different fact types. Similarly, you may use an optional condition "ConditionAny" which instead of fact types can use any formulas defined on any known objects. It is also possible to add custom actions using an optional action "ActionAny" – see "DecisionTemplates.xls" in the folder "openrules.config".

When you have completed defining all decision and sub-decisions, you may define rule families.

Defining Rule Families

At the heart of the Decision Model approach is the concept of the "Rule Family". Rule families specify business logic using conditions and conclusions defined on fact types. For example, let's consider a very simple decision "DetermineCustomerGreeting":

Decision DetermineCustomerGreeting	
Decisions	**Execute Rule Families**
Define Greeting Word	:= DefineGreeting()
Define Salutation Word	:= DefineSalutation()

It refers to two rule families. Here is an example of the first rule family:

RuleFamily DefineGreeting					
Condition		Condition		Conclusion	
Current Hour		**Current Hour**		**Greeting**	
>=	0	<=	11	Is	Good Morning
>=	11	<=	17	Is	Good Afternoon
>=	17	<=	22	Is	Good Evening
>=	22	<=	24	Is	Good Night

Its first row contains a keyword "RuleFamily" and a unique name (no spaces allowed). The second row uses keywords "Condition" and "Conclusion" to specify the types of the rule family columns. The third row contains fact types expressed in plain English (spaces are allowed but the fact names should be unique).

The columns of a rule family define conditions and conclusions using different operators and operands appropriate to the fact type specified in the column headings. The rows of a rule family specify multiple rules. For instance, in the above rule family "DefineGreeting" the second rule can be read as:

"IF Current Hour is more than or equal to 11 AND Current Hour is less than or equal to 17 THEN Greeting is Good Afternoon".

Similarly, we may define the second rule family "DefineSalutation" that determines a salutation word:

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

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

Rule Family Execution Logic

OpenRules® executes all rules within a rule family in a top-down order. When all conditions inside one rule (row) are satisfied the proper conclusion(s) from the same row will be executed, and all other rules will be ignored.

However, the Decision Model approach requires that the order of rules inside a rule family should not matter. It means that to stay compliant with the Decision Model principle, you should not rely on the top-down rules execution order and design your rule families in a such way that all rules are mutually exclusive and cover all possible combinations of conditions. The advantage of this approach is that when you decide to add new rules to your rule family you may place them in any rows without jeopardizing the execution logic. In some cases, however, a problem may require a more complex and precise rule family organization. See more options below.

AND/OR Conditions

The conditions in a rule family are always connected by a logical operator "AND". When you need to use "OR", you may add another rule (row) that is an alternative to the previous rule(s). However, some conditions may have a fact type defined as an array, and within such array-conditions "ORs" are allowed. Consider for example the following, more complex rule family:

RuleFamily DefineUpSellProducts								
Condition		Condition		Condition		Conclusion		Message
Customer Profile		**Customer Products**		**Customer Products**		**Offered Products**		**Set Comment**
Is One Of	New,Bronze,Silver	Include	Checking Account	Do Not Include	Saving Account	Are	Saving Account, Debit/ATM Card, Web Banking	
Is One Of	New,Bronze,Silver	Include	Checking Account, Overdraft Protection	Do Not Include	CD with 25 basis point increase, Money Market Mutual Fund, Credit Card	Are	CD with 25 basis point increase, Money Market Mutual Fund, Credit Card	
Is One Of	New,Bronze,Silver	Include	Checking Account, Saving Account	Do Not Include	CD with 25 basis point increase, Money Market Mutual Fund, Credit Card	Are	CD with 50 basis point increase, Money Market Mutual Fund, Credit Card, Debit/ATM Card, Web Banking	
Is One Of	Gold	Include	Checking Account	Do Not Include	CD with 25 basis point increase, Money Market Mutual Fund, Web Banking	Are	CD with 50 basis point increase, Money Market Mutual Fund, Credit Card, Debit/ATM Card, Web Banking, Brokerage Account	Gold Package
Is One Of	Platinum	Include	Checking Account, Saving Account	Do Not Include	CD with 25 basis point increase, Money Market Mutual Fund, Web Banking	Are	CD with 50 basis point increase, Money Market Mutual Fund, Credit Card with no annual fee, Debit/ATM Card, Web Banking with no charge, Brokerage Account	Platinum Package
						Are	None	Sorry

Here the fact types "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 New or Bronze or Silver
AND Customer Products Include Checking Account and
Overdraft Protection
AND Customer Products Do Not Include CD with 25 basis point
increase, Money Market Mutual Fund, and Credit Card
THEN Offered Products ARE CD with 25 basis point increase,
Money Market Mutual Fund, and Credit Card
```

Rule Family Operators

OpenRules® supports multiple ways to define operators within rule family conditions and conclusions. When you use a text form of operators you can freely use upper and lower cases and spaces. The following operators can be used inside rule family conditions:

Operator	Synonyms	Comment
Is	=, ==	When you use "=" or "==" inside Excel write"'=" or"'==" to avoid confusion with Excel's own formulas

Is Not	!=, isnot, Is Not Equal To, Not, Not Equal., Not Equal To	Defines an inequality operator
>	Is More, More, Is More Than, Is Greater, Greater, Is Greater Than	For integers and real numbers, and Dates
>=	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	For integers and real numbers, and Dates
<=	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,	For integers and real numbers, and Dates
<	Is Less, Less, Is Less Than, Is Smaller, Smaller, Is Smaller Than	For integers and real numbers, and Dates
Is True		For booleans
Is False		For booleans
Contains	Contain	For strings only, e.g. "House" contains "use". The comparison is not case-sensitive
Starts With	Start with, Start	For strings only, e.g. "House" starts with "ho". The comparison is not case-sensitive
Within	Inside, Inside Interval, Interval	For integers and real numbers. The interval can be defined as: [0;9], (1;20], 5–10, between 5 and 10, more than 5 and less or equals 10 – see more

Is One Of	Is One, Is One of Many, Is Among, Among	For integer and real numbers, and for strings. Checks if a value is among elements of the domain of values listed through comma
Include	Include All	To compare an array or value with an array
Exclude	Do Not Include, Exclude One Of	To compare an array or value with an array
Intersect	Intersect With, Intersects	To compare an array with an array

If the fact types do not have an expected type for a particular operator, the proper syntax error will be diagnosed.

The following operators can be used inside rule family conclusions:

Operator	Synonyms	Comment
Is	=, ==	Assigns one value to the conclusion fact type. When you use "=" or "==" inside Excel write'"=" or'"==" to avoid confusion with Excel's own formulas.
Are		Assigns one or more values listed through commas to the conclusion fact that is expected to be an array
Add		Adds one or more values listed through commas to the conclusion fact that is expected to be an array

Conditions and Conclusions without Operators

Sometimes the creation of special columns for operators seems unnecessary, especially for the operators "Is" and "Within". OpenRules® allows you to use a simpler format as in this rule family:

RuleFamily DefineGreeting	
If	Then
Current Hour	**Greeting**
0-11	Good Morning

11-17	Good Afternoon
17-22	Good Evening
22-24	Good Night

As you can see, instead of keywords "Condition" and "Conclusion" we use the keywords "If" and "Then" respectively. While this rule family looks much simpler in comparison with the functionally identical rule family defined above, we need to make an implicit assumption that the lower and upper bounds for the intervals "0-11", "11-17", etc. are included.

Using Formulas inside Rule Families

OpenRules® allows you to use formulas in the rule cells instead of constants. The formulas usually have the following format:

```
::= ( expression )
```

where an "expression" can be written using standard Java expressions. Here is an example:

RuleFamily **CalculateAdjustedGrossIncome**	
Conclusion	
Adjusted Gross Income	
Is	::= (getReal("Wages") + getReal("Taxable Interest") + getReal("Unemployment Compensation"))

This rule family simply calculates a value for the fact type "Adjusted Gross Income" as a sum of values for the fact types "Wages", "Taxable Interest", and "Unemployment Compensation". This example also demonstrates how to gain access to different fact types – you may simply write getReal("FACT_TYPE_NAME") for real fact types. Similarly, you may use methods `getInt(…)`, `getBool(…)`, `getDate(…)`, and `getString(…)`.

You may also put your formula in a specially defined Method and then refer to this method from the rule family – observe how it is done in the following example:

Method double taxableIncome()
return getReal("Adjusted Gross Income") - getReal("Dependent Amount");

RuleFamily CalculateTaxableIncome			
ConditionAny		Conclusion	
Condition		**Taxable Income**	
Is True	:= (taxableIncome() > 0)	Is	::= taxableIncome()
Is False	:= (taxableIncome() > 0)	Is	0

Here we defined a new method "taxableIncome()" that returns a real value using the standard Java type "double". Then we used this method inside both conditions and one conclusion of this rule family.

Note. Actually the formula format `::= ( expression )` is a shortcut for a more standard OpenRules® formula format `:= "" +( expression )` that also can be used inside rule families.

Defining Business Glossary

While defining rules families, we freely introduced different fact types assuming that they are somehow defined. The business glossary is a special OpenRules® table that actually defines all fact types. The Glossary table has the following structure:

Glossary glossary			
Fact Name	**Business Concept**	**Attribute**	**Domain**

The first column will simply list all of the fact types using exactly the same names that were used inside the rule families. The second column associates different fact types with the business concepts to which they belong. Usually you want to keep fact types that belong to the same business concept together and merge all rows in the column "Business Concept" that share the same concept. Here is an example of a glossary from the standard OpenRules® example "DecisionLoan":

Glossary glossary			
Fact Type	**Object**	**Attribute**	**Domain**
Monthly Income	Customer	monthlyIncome	0-5000000
Mortgage Holder		mortgageHolder	Yes,No
Outside Credit Score		outsideCreditScore	0-999
Loan Holder		loanHolder	Yes,No
Credit Card Balance		creditCardBalance	-1000000 - 100000000
Education Loan Balance		educationLoanBalance	-1000000 - 100000000
Internal Credit Rating		internalCreditRating	A,B,C,D,F
Internal Analyst Opinion		internalAnalystOpinion	High,Mid,Low
Income Validation Result	Request	incomeValidationResult	SUFFICIENT,UNSUFFICIENT,?
Debt Research Result		debtResearchResult	High,Mid,Low,?
Loan Qualification Result		loanQualificationResult	QUALIFIED, NOT QUALIFIED, ?
Total Income	Internal	totalIncome	0-500000
Total Debt		totalDebt	0-500000

All rows for the concepts such as “Customer” and “Request” are merged.

The third column “Attribute” contains “technical” names of the fact types – these names will be used to connect our fact types with attributes of objects used by the actual applications, for which a decision model has been defined. The application objects could be defined in Java, in Excel tables, in XML, etc. The decision model does not have to know about it: the only requirement is that the attribute names should follow the usual naming convention for identifiers in languages like Java: it basically means no spaces allowed. The last column “Domain”, is optional, but it can be useful to specify which values are allowed to be used for different fact types. Fact type domains can be specified using the naming convention for the intervals and domains described below. The above glossary provides a few intuitive examples of such domains. These domains can be used during the validation of a decision model.

Defining Test Data

OpenRules® provides a convenient way to define test data for decision models directly in Excel without the necessity of writing any Java code. A non-technical user can define all business concepts in the Glossary table using Datatype tables.

For example, here is a Datatype table for the business concept "Customer" defined above:

Datatype Customer	
String	fullName
String	SSN
int	monthlyIncome
int	monthlyDebt
String	mortgageHolder
int	outsideCreditScore
String	loanHolder
int	creditCardBalance
int	educationLoanBalance
String	internalCreditRating
String	internalAnalystOpinion

The first column defines the type of the attribute using standard Java types such as "`int`", "`double`", "`Boolean`", "`String`", or "`Date`". The second column contains the same attribute names that were defined in the Glossary. To create an array of objects of the type "`Customer`" we may use a special "`Data`" table like the one below:

Data Customer customers										
fullName	SSN	monthlyIncome	monthlyDebt	mortgageHolder	outsideCreditScore	loanHolder	creditCardBalance	educationLoanBalance	internalCreditRating	internalAnalystOpinion
Borrower Full Name	**Borrower SSN**	**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	157-82-5344	5000	2300	Yes	720	No	2500	0	A	Low
Mary K. Brown	056-45-8233	4300	2800	No	620	No	5654	23800	B	Low
Robert Cooper Jr.	241-58-9082	6400	2800	Yes	735	Yes	1200	0	C	Mid

This table is too wide (and difficult to read), so we could actually transpose it to a more convenient but equivalent format:

Data Customer customers				
fullName	**Borrower Full Name**	Peter N. Johnson	Mary K. Brown	Robert Cooper Jr.
SSN	**Borrower SSN**	157-82-5344	056-45-8233	241-56-9082
monthlyIncome	**Monthly Income**	5000	4300	6400
monthlyDebt	**Monthly Debt**	2300	2800	2800
mortgageHolder	**Mortgage Holder**	Yes	No	Yes
outsideCreditScore	**Outside Credit Score**	720	620	735
loanHolder	**Loan Holder**	No	No	Yes
creditCardBalance	**Credit Card Balance**	2500	5654	1200
educationLoanBalance	**Education Loan Balance**	0	23800	0
internalCreditRating	**Internal Credit Rating**	A	B	C
internalAnalystOpinion	**Internal Analyst Opinion**	Low	Low	Mid

Now, whenever we need to reference the first customer we can refer to him as `customers[0]`. Similarly, if you want to define a doubled monthly income for the second custromer, "Mary K. Brown", you may simply write

```
::= (customers[1].monthlyIncome * 2)
```

You can find many additional details about data modeling in this section.

Connecting the Decision Model with Business Objects

To tell OpenRules® that we want to associate the object `customers[0]` with our business concept "Customer" defined in the Glossary, we need to use a special table "DecisionObject" that may look as follows:

DecisionObject decisionObjects	
Business Concept	**Business Object**
Customer	:= customers[0]
Request	:= loanRequests[0]
Internal	:= internal

Here we also associate other business concepts namely Request and Internal with the proper business objects – see how they are defined in the standard example "DecisionLoan".

The above table connects a decision model with test data defined by business users directly in Excel. This allows the model to be tested. However, after the model is tested, it will be integrated into a real application that may use objects defined in Java, in XML, or in a database, etc. For example, if there are

instances of Java classes Customer and LoanRequest, they may be put in the object "decision" that is used to execute the decision model. In this case, the proper table "decisionObjects" may look like:

DecisionObject decisionObjects	
Business Concept	**Business Object**
Customer	:= decision.get("customer")
Request	:= decision.get("loanRequests")
Internal	:= internal

It is important that a decision model does not "know" about a particular object implementation: the only requirement is that the attribute inside these objects should have the same names as in the glossary.

Note. You cannot use the predefined function "decision()" within the table "decisionObjects" because its content is be not defined yet. You need to use the internal variable "decision" directly.

Executing Decision Models

OpenRules® provides a template for Java launchers that may be used to execute different decision models. There are OpenRules® API classes OpenRulesEngine and Decision. Here is an example of a decision model launcher for the sample project "DecisionLoan":

```
import com.openrules.ruleengine.Decision;

public class Main {
    public static void main(String[] args) {
        String fileName = "file:rules/main/Decision.xls";
        Decision decision = new Decision("DetermineLoanPreQualificationResults",fileName);
        decision.execute();
    }
}
```

Actually, it just creates an instance of the class Decision. It has only two parameters:

1) a path to the main Excel file "Decision.xls"
2) a name of the main Decision inside this Excel file.

When you execute this Java launcher using the provided batch file "run.bat" or execute it from your Eclipse IDE, it will produce output that may look like the following:

```
*** Decision DetermineLoanPreQualificationResults ***
Decision has been initialized
```

```
Decision DetermineLoanPreQualificationResults: Calculate Internal
Variables
Conclusion: Total Debt Is 165600.0
Conclusion: Total Income Is 360000.0
Decision DetermineLoanPreQualificationResults: Validate Income
Conclusion: Income Validation Result Is SUFFICIENT
Decision DetermineLoanPreQualificationResults: Debt Research
Conclusion: Debt Research Result Is Low
Decision DetermineLoanPreQualificationResults: Summarize
Conclusion: Loan Qualification Result Is NOT QUALIFIED
ADDITIONAL DEBT RESEARCH IS NEEDED from DetermineLoanQualificationResult
*** OpenRules made a decision ***
```

This output shows all sub-decisions and conclusion results for the corresponding rule families.

Controlling Decision Model Output

OpenRules® relies on the standard Java logging facilities for the decision model output. They can be controlled by the standard file "log4j.properties".

You may control how "talkative" your decision model is by setting decision's parameter "Trace". For example, if you add the following setting to the above Java launcher

```
decision.put("trace", "Off");
```

just before calling `decision.execute()`, then your output will be much more compact:

```
*** Decision DetermineLoanPreQualificationResults ***
Decision DetermineLoanPreQualificationResults: Calculate Internal
Variables
Decision DetermineLoanPreQualificationResults: Validate Income
Decision DetermineLoanPreQualificationResults: Summarize
ADDITIONAL DEBT RESEARCH IS NEEDED from DetermineLoanQualificationResult
*** OpenRules made a decision ***
```

You may also change output by modifying the tracing details inside the proper decision templates in the configuration files "DecisionTemplates.xls" and "RuleFamilyExecuteTemplates.xls".

Validating Decision Model

OpenRules® allows you to validate your decision model by checking that:

- there are no syntax error in the organization of all decision model tables
- values inside rule family fact type cells correspond to the associated domains defined in the glossary.

To validate a decision model you can use write `decision.validate()` instead of `decision.execute()` in the proper Java launcher. The validation template is described in the standard file "RuleFamilyValidateTemplates.xls".

Note. The next release of OpenRules® will include new validation facilities that will check whether rule families have conflicts such as overlapping or under-coverage. It can also be used make sure that the major principles of The Decision Model approach are satisfied.

BEYOND THE DECISION MODEL

OpenRules® does not prevent you from creating executable decision models in accordance with the principles of The Decision Model approach. However, OpenRules® also allows a user to go beyond this approach when necessary. In real-world situations OpenRules® customers found that sometimes they needed more complex representations of rule families and the relationships between them. For example, OpenRules® allows you to arrive at multiple conclusions inside the same rules family (while the standard approach allows only a single conclusion).

Specialized Conditions and Conclusions

The standard columns of the types "**Condition**" and "**Conclusion**" always have two sub-columns: one for operators and another for values. OpenRules® allows you to specify columns of the types "**If**" and "**Then**" that do not require sub-columns. Instead, they allow you to use operators or even natural language expressions together with values to represent different intervals and domains of values. Read about different ways to represent intervals and domains in this section below.

Sometimes your conditions or actions are not related to a particular fact type and can be calculated using formulas. For example, a condition can be defined based on combination of several fact types, and you would not want to artificially add an intermediate fact type to your glossary in order to accommodate each needed combination of existing fact types. In such a case, you may use a special type "**ConditionAny**" like in the example below:

RuleFamily CalculateTaxableIncome			
ConditionAny		Conclusion	
Condition		**Taxable Income**	
Is True	:= (getReal("Adjusted Gross Income") > getReal("Dependent Amount"))	Is	::= (getReal("Adjusted Gross Income") - getReal("Dependent Amount"))
Is False	:= (getReal("Adjusted Gross Income") > getReal("Dependent Amount"))	Is	0

Here the word "Condition" does not represent any fact type and instead you may insert any text, i.e. "Compare Adjusted Gross Income with Dependent Amount". When your conclusion, does not set a value for a single fact type but rather does something that is expressed in the formulas within the cells of this column, you should use a column of type "**ActionAny**". It does not have sub-columns because there is no need for an operator.

Note. There is also a column of type "**Action**" that is equivalent to type "Then".

Specialized Rule Families

Sometimes the standard behavior of a RuleFamily is not sufficient. OpenRules® provide two non-standard types of rule families RuleFamily1 and RuleFamily2. While we recommend avoiding these types of rule families, in certain situations they provide a convenient way around the limitations imposed by the standard RuleFamily.

RuleFamily1

Contrary to the standard RuleFamily that is implemented as a single-hit decision table, rule families of type "RuleFamily1" are implemented as multi-hit decision tables. "RuleFamily1" supports the following rules execution logic:

1. All rules are evaluated and if their conditions are satisfied, they will be marked as "to be executed"
2. All actions columns (of the types "Conclusion", "Then", "Action", "ActionAny", or "Message") for the "to be executed" rules will be executed in top-down order.

Thus, we can make two important observations about the behavior of the "RuleFamily1":

- Rule actions cannot affect the conditions of any other rules in the rule family – there will be no re-evaluation of any conditions
- 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 a rule that states: "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 standard RuleFamily, it may look as follows:

RuleFamily ValidateDrivingEligibility					
Condition		Condition		Conclusion	
Driver's Age		US State		Driving Eligibility	
>=	17			Is	Eligible
Is	16	Is Not	Florida	Is	Not Eligible
Is	16	Is	Florida	Is	Eligible
<	16			Is	Not Eligible

Using a non-standard RuleFamily1 we may present the same rule as:

RuleFamily1 ValidateDrivingEligibility					
Condition		Condition		Conclusion	
Driver's Age		US State		Driving Eligibility	
				Is	Eligible
<	17			Is	Not Eligible
>=	16	Is	Florida	Is	Eligible

Here the first unconditional rule will set "Driving Eligibility" to "Eligible". The second rule will reset it to "Not Eligible" for all people younger than 17. But for 16 year olds living in Florida, the third rule will override the fact again to "Eligible".

RuleFamily2

There is one more type of rule family, "RulesFamily2," that is similar to "RulesFamily1" but allows the actions of already executed rules to affect the conditions of rules specified below them. "RuleFamily2" supports the following rules execution logic:

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

Thus, we can make two important observations about the behavior of the "RuleFamily2":

- 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.

Let's consider the following example:

RuleFamily2 CalculateTaxableIncome			
Condition		Conclusion	
Taxable Income		**Taxable Income**	
		Is	::= (getReal("Adjusted Gross Income") - getReal("Dependent Amount"))
Is Less	0	Is	0

Here the first (unconditional) rule will calculate and set the value of the fact type "Taxable Income". The second rule will check if the calculated value is less than 0. If it is true, this rule will reset this fact type to 0.

DECISION TABLES

OpenRules® supports several ways to represent business rules inside Excel tables. A decision table is the most popular way to present sets of related business rules. Decision tables are used to describe and analyze decision situations, where the state of a number of conditions determines the execution of a set of actions. OpenRules® allows a user to configure different types of decision tables directly in Excel. The user also may use simple IF-THEN-ELSE statements to describe rules logic inside tables of type "Method".

The Decision Model described above with inter-connected rule families became available only in March 20011. A key advantage is its complete orientation to business users. It means that a business specialist (not a programmer) can design and test a decision model yet no coding is required. However, OpenRules® has offered very powerful decision tables presented in Excel since its beginning in 2003, and these decision tables are successfully used by major corporations in real-world decision support applications. Actually, a rule family is a special case

of an OpenRules® single-hit decision table that is based on a predefined template (see below). This chapter describes different decision tables that go beyond the Decision Model approach.

Simple Rules Table

Let's consider a simple set of HelloWorld rules that can be used to generate a string like "Good Morning, World!" based on the actual time of the day. How one understands such concepts as "morning", "afternoon", "evening", and "night" is defined in this simple decision table:

Rules void helloWorld(int hour)		
Hour From	**Hour To**	**Greeting**
0	11	Good Morning
12	17	Good Afternoon
18	22	Good Evening
23	24	Good Night

Hopefully, this decision table is self-explanatory. It states that if the current hour is between 0 and 11, the greeting should be "Good Morning", etc. You may change Hour From or Hour To if you want to customize the definition of "morning" or "evening". This decision table is oriented to a business user. However, its first row already includes some technical information (a table signature):

```
Rules void helloWorld(int hour)
```

Here "Rules" is the OpenRules® keyword to recognize a table type as a decision table; "helloWorld" is the name of this particular decision table. It tells an external program or other rules how to launch this rule table. Actually, this is a typical description of a programming method (its signature) that has one integer parameter and returns nothing (the type "void"). The integer parameter "hour" is expected to contain the current time of the day. While you can always hide this information from a business user, it is an important specification of this decision table.

You may wonder, where is the implementation logic for this decision table? All rules tables include additional hidden rows (frequently password protected) that you can see if you click on the buttons "+" to open the Technical View below:

	A	B	C	D
1				
2				
3		Rules void helloWorld(int hour)		
4		C1	C2	A1
5		min <= hour	hour <= max	System.out.println(greeting + ", World!")
6		int min	int max	String greeting
7		**Hour From**	**Hour To**	**Greeting**
8		0	11	Good Morning
9		12	17	Good Afternoon
10		18	22	Good Evening
11		23	24	Good Night
12				

This part of the rule table is oriented to a technical user, who is not expected to be a programming guru but rather a person with a basic knowledge of the "C" family of languages which includes Java. Let's walk through these rows step by step:

- Row "Condition and Action Headers" (see row 4 in the table above). The initial columns with conditions should start with the letter "C", for example "C1", "Condition 1". The columns with actions should start with the letter "A", for example "A1", "Action 1".
- Row "Code" (see row 5 in the table above). The cells in this row specify the semantics of the condition or action associated with the corresponding columns. For example, the cell B5 contains the code `min <= hour`. This means that condition C1 will be true whenever the value for *min* in any cell in the column below in this row is less than or equals to the parameter `hour`. If *hour* is 15, then the C1-conditions from rows 8 and 9 will be satisfied. The code in the Action-columns defines what should be done when all conditions are satisfied. For example, cell D5 contains the code: `System.out.println(greeting + ", World!")`. This code will print a string composed of the variable `greeting` and `", World!"`, where `greeting` will be chosen from a row where all of the conditions are satisfied. Again, if `hour` is 15, then both conditions C1 and C2 will be

satisfied only for row 9 (because 9 <= 15 <= 17). As a result, the words "Good Afternoon, World!" will be printed. If the rules table does not contain a row where all conditions have been satisfied, then no actions will be executed. Such a situation can be diagnosed automatically.

- Row "Parameters" (see row 6 in the table above). The cells in this row specify the types and names of the parameters used in the previous row.
- Row "Display Values" (see row 7 in the table above). The cells in this row contain a natural language description of the column content.

The same table can be defined a little bit differently using one condition code for both columns "min" and "max":

Rules void **defineGreeting**(App app, int hour)		
C1		A1
min <= hour && hour <= max		app.greeting = greeting;
int min	int max	String greeting
Hour From	**Hour To**	**Set Greeting**
0	11	Good Morning
12	17	Good Afternoon
18	22	Good Evening
23	24	Good Night

How Decision Tables Are Organized

As you have seen in the previous section, decision tables have the following structure:

Row #	Content	Description
1	Signature	Rules void tableName(Type1 par1, Type2 par2, ..) - Multi-Hit Rules Table Rules <JavaClass> tableName(Type1 par1, Type2 par2, ..) - Single-Hit Rules Table
2	Condition/Action Indicators	The condition column indicator is a word starting with “C”. The action column indicator is a word starting with “A”.

		All other starting characters are ignored and the whole column is considered as a comment
3	Code	The cells in each column (or merged cells for several columns) contain Java Snippets in the language defined in the table Environment (the default language configuration is openrules.java). Condition codes should contain expressions that return *Boolean* values.. If an action code contains any correct Java snippet, the return type is irrelevant.
4	Parameters	Each condition/action may have from 0 to N parameters. Usually there is only one parameter description and it consists of two words: *parameterType parameterName* (for example: *int min*) *parameterName* is a standard one word name that corresponds to Java identification rules. *parameterType* can be represented using the following Java types: - Basic Java types: *boolean, char, int, long, double, String, Date* - Standard Java classes: java.lang.Boolean, java.lang.Integer, java.lang.Long, java.lang.Double, java.lang.Character, java.lang.String, java.util.Date - Any custom Java class with a public constructor that has a String parameter - One-dimensional arrays of the above types. Multiple parameters can be used in the situations when one code is used for several columns. See the standard example Loan1.xls
5	Columns Display Values	Text is used to give the column a definition that would be meaningful to another reader (there are no restrictions on what text may be used)

6 and below	Rules with concrete values in cells	Cells in these rows usually contain literals that correspond to the parameter types. **For Boolean parameters** you may enter the values "TRUE" or "FALSE" (or equally "Yes" or "No") without quotations. **Empty cells** inside rules means "whatever" and the proper condition is automatically considered satisfied. An action with an empty value will be ignored. If the parameter has type String and you want to enter a space character, you must explicitly enter one of the following expressions: :=" " or '=" " or { " "; } **Cells with Dates** can be specified using java.util.Date. OpenRules® uses java.text.DateFormat.SHORT to convert a text defined inside a cell into java.util.Date. Before OpenRules® 4.1 we recommended our customers not to use Excel's Date format and define Date fields in Excel as Text fields. The reason was the notorious Excel problem inherited from a wrong assumption that 1900 was a leap year. As a result, a date entered in Excel as 02/15/2004 could be interpreted by OpenRules® as 02/16/2004. Starting with release 4.1 OpenRules® correctly interprets both Date and Text Excel Date formats. **Valid Java expression** (Java snippets) may be put inside table cells by one of two ways: - by surrounding the expression in curly brackets, for example: { driver.age+1; } - by putting ":=" in front of your Java expression, for example: :=driver.age+1 Make sure that the expression's type corresponds to the parameter type.

Note. Excel is always trying to "guess" the type of text is inside its cells and automatically converts the internal representation to something that may not be exactly what you see. For example, Excel may use a scientific format for certain numbers. To avoid a "strange" behavior try to explicitly define the format "text" for the proper Excel cells.

Separating Business and Technical Information

During rules harvesting, business specialists initially create rules tables using regular Excel tables. They put a table name in the first row and column names in the second row. They start with Conditions columns and end with Action columns. For example, they can create a table with 5 columns [C1,C2,C3,A1,A2] assuming the following logic:

```
IF conditions C1 and C2 and C3 are satisfied

THEN execute actions A1 and A2
```

Then, a business specialist provides content for concrete rules in the rows below the title rows.

As an example, let's consider the decision table "defineSalutation" with the rules that define how to greet a customer (Mr., Ms, or Mrs.) based on his/her gender and marital status. Here is the initial business view (it is not yet syntactically correct):

Rules defineSalutation		
Gender	**Marital Status**	**Set Salutation**
Male		Mr.
Female	Married	Mrs.
Female	Single	Ms.

A business analyst has initially created only five rows:

- A signature "Rules defineSalutation" (it is not a real signature yet)
- A row with column titles: two conditions "Gender", "Marital Status" and one action "Set Salutation"
- Rows with three rules that can be read as:

1) IF Gender is “Male” THEN Set Salutation “Mr."
2) IF Gender is “Female” and Marital Status is “Married” THEN Set Salutation “Mrs.”
3) IF Gender is “Female” and Marital Status is “Single” THEN Set Salutation “Ms.”

While business specialists continue to define such decision tables, at some point a technical specialist should take over and add to these tables the actual implementation. The technical specialist (familiar with the software environment into which these rules are going to be embedded) talks to the business specialist (author of the rule table) about how the rules should be used. In the case of the "defineSalutation" rule table, they agree that the table will be used to generate a salutation to a customer. So, the technical specialist decides that the table will have two parameters:

1) a customer of the type Customer
2) a response of the type Response

The technical specialist will modify the signature row of the table to look like this:

```
Rules void defineSalutation(Customer customer, Response response)
```

Then s/he inserts three more rows just after the first (signature) row:

- Row 2 with Condition/Action indicators
- Row 3 with Condition/Action implementation
- Row 4 with the type and name of the parameters entered in the proper column.

Here is a complete implementation of this decision table:

Rules void **defineSalutation**(Customer customer, Response response)		
C1	C2	A1
customer.gender.equals (gender)	customer.maritalStatus.equal s(status)	response.map.put("salutation", salutation);
String gender	String status	String salutation
Gender	**Marital Status**	**Set Salutation**
Male		Mr.
Female	Married	Mrs.
Female	Single	Ms.

The rules implementer will decide that to support this rule table, type Customer should have at least two attributes, "gender" and "maritalStatus", and the type Response should be able somehow to save different pairs (names,value) like*("salutation","Mr.")*. Knowing the development environment, s/he will decide on the types of attributes. Let's assume that both types Customer and Response correspond to Java classes, and the attributes have the basic Java type of String. In this case, the column "Gender" will be marked with a parameter "String gender" and the condition will be implemented as a simple boolean expression:

```
customer.gender.equals(gender)
```

The second column "C2" is implemented similarly with a String attribute and a parameter maritalStatus. Finally (to make it a little bit more complicated), we will assume that the class Response contains an attribute map of the predefined Java type HashMap, in which we can put/get pairs of Strings. So, the implementation of the action "Set Salutation" will look like:

```
response.map.put("salutation",salutation)
```

How Decision Tables Are Executed

The rules inside rules tables are executed one-by-one in the order they are placed in the table. The execution logic of one rule (row in the vertical table) 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 (row) are ignored and **are not evaluated.** The absence of a parameter in a condition cell means the condition is always **true.** Actions are evaluated only if all conditions in the same row are evaluated to be **true** and the action has non-empty parameters. Action columns with no parameters are ignored.

For the default vertical rules tables, all rules are executed in top-down order. There could be situations when all conditions in two or more rules (rows) are satisfied. In that case, the actions of all rules (rows) will be executed, and the actions in the rows below can **override** the actions of the rows above.

For horizontal decision tables, all rules (columns) are executed in left-to-right order.

Relationships between Rules inside Decision Tables

OpenRules® does not assume any implicit ("magic") execution logic, and executes rules in the order specified by the rule designer. All rules are executed one-by-one in the order they are placed in the rules table. There is a simple rule that governs rules execution inside a rules table:

The preceding rules are evaluated and executed first!

OpenRules® supports 3 types of decision tables that offer different execution logic to satisfy different practical needs:

- Multi-hit rules tables
- Single-hit rules tables
- Rule Sequences.

Multi-Hit Rules Tables

A multi-hit Rules Table evaluates conditions in ALL rows before any action is executed. Thus, actions are executed only AFTER all conditions for all rules have already been evaluated. From this point of view, the execution logic is different from traditional programming if-then logic. Let us consider a simple example. We want to write a program "swap" that will do the following:

```
If x is equal to 1 then make x to be equal to 2.
If x is equal to 2 then make x to be equal to 1.
```

Suppose you decided to write a Java method assuming that there is a class App with an integer variable x. The code may (but should not) look like this:

```
void swapX(App app) {
      if (app.x == 1) app.x = 2;
      if (app.x == 2) app.x = 1;
}
```

Obviously, this method will produce an incorrect result because of the missing "else". This is “obvious” to a software developer, but may not be at all obvious to a business analyst. However, in a properly formatted decision table the following representation would be a completely legitimate:

If x equals to	Then make x to be equal to
1	2
2	1

It will also match our plain English description above. Here is the same table with an extended technical view:

Rules void swapX(App app)	
C	A
app.x == oldValue	app.x = newValue
int oldValue	int newValue
If x equals to	**Then make x to be equal to**
1	2
2	1

Rules Overrides in Multi-Hit Tables

There could be situations when all conditions in two or more rules (rows) are satisfied at the same time (multiple hits). In that case, the actions of all rules (rows) will be executed, but the actions in the rows below can **override** the actions of the rows above. This approach also allows a designer to specify a very natural requirement:

More specific rules should override more generic rules!

The only thing a designer needs to guarantee is that "more specific" rules are placed in the same decision table after "more generic" rules. For example, you may want to execute Action-1 every time that Condition-1 and Condition-2 are satisfied. However, if additionally, Condition-3 is also satisfied, you want to

execute Action-2. To do this, you could arrange your decision table in the following way:

Condition-1	Condition-2	Condition-3	Action-1	Action-2
X	X		X	
X	X	X		X

In this table the second rule may override the first one (as you might naturally expect).

Let's consider the execution logic of the following multi-hit rules table that defines a salutation "Mr.", "Mrs.", or "Ms." based on a customer's gender and marital status:

Rules void **defineSalutation**(Customer customer, Response response)		
Gender	**Marital Status**	**Set Salutation**
Male		Mr.
Female	Married	Mrs.
Female	Single	Ms.

If a customer is a married female, the conditions of the second rules are satisfied and the salutation "Mrs." will be selected. This is only a business view of the rules table. The complete view including the hidden implementation details ("Java snippets") is presented below:

Rules void **defineSalutation**(Customer customer, Response response)		
C1	C2	A1
customer.gender.equals(gender)	customer.maritalStatus.equals(status)	response.map.put("salutation",salutation);
String gender	String status	String salutation

Gender	Marital Status	Set Salutation
Male		Mr.
Female	Married	Mrs.
Female	Single	Ms.

The OpenRulesEngine will execute rules (all 3 "white" rows) one after another. For each row if conditions C1 and C2 are satisfied then the action A1 will be executed with the selected "salutation".

We may add one more rule at the very end of this table:

Rules void **defineSalutation**(Customer customer, Response response)		
Gender	**Marital Status**	**Set Salutation**
Male		Mr.
Female	Married	Mrs.
Female	Single	Ms.
		???

In this case, after executing the second rule OpenRules® will also execute the new 4th rule and will override a salutation "Mrs." with "???". Obviously this is not a desirable result. However, sometimes it may have a positive effect by avoiding undefined values in cases when the previous rules did not cover all possible situations. What if our customer is a Divorced Female?! How can this multi-hit effect be avoided? What if we want to produce "???" only when no other rules have been satisfied?

Single-Hit Decision Tables

To achieve this you may use a so-called "**single-hit**" rules table, which is specified by putting any return type except "void" after the keyword "Rules". The

following is an example of a single-hit rules table that will do exactly what we need:

Rules String **defineSalutation**(Customer customer, Response response)		
Gender	**Marital Status**	**Set Salutation**
Male		Mr.
Female	Married	Mrs.
Female	Single	Ms.
		???

Another positive effect of such "single-hitness" may be observed in connection with large tables with say 1000 rows. If OpenRules® obtains a hit on rule #10 it would not bother to check the validity of the remaining 990 rules.

Having rules tables with a return value may also simplify your interface. For example, we do not really need the special object Response which we used to write our defined salutation. Our simplified rules table produces a salutation without an additional special object:

Rules String **defineSalutation**(Customer customer)		
C1	C2	A1
customer.gender. equals(gender)	customer.maritalStatus .equals(status)	return salutation;
String gender	String status	String salutation
Gender	**Marital Status**	**Set Salutation**
Male		Mr.
Female	Married	Mrs.
Female	Single	Ms.
		???

Please note that the last action in the table should return a value that has the same type as the entire single-hit table. The single-hit table may return any standard or custom Java class such as String or Customer. Instead of basic Java types such as "int" you should use the proper Java classes such as Integer in the table signature.

Here is an example of Java code that creates an `OpenRulesEngine` and executes the latest rules table "defineSalutation":

```
public static void main(String[] args) {
    String fileName = "file:rules/main/HelloCustomer.xls";
    OpenRulesEngine engine =
                    new OpenRulesEngine(fileName);
    Customer customer = new Customer();
    customer.setName("Robinson");
    customer.setGender("Female");
    customer.setMaritalStatus("Married");
    String salutation =
           (String)engine.run("defineSalutation", customer);
    System.out.println(salutation);
}
```

Rule Sequences

There is one more type of decision tables called “Rule Sequence” that is used mainly internally within templates. Rule Sequence can be considered as a multi-hit decision table with only one difference in the execution logic, conditions are not evaluated before execution of the actions. So, all rules will be executed in top-down order with possible rules overrides. Rule actions are permitted to affect the conditions of any rules that follow the action. The keyword “`Rules`” should be replaced with another keyword “`RuleSequence`”. Let’s get back to our “swapX” example. The following multi-hit table will correctly solve this problem:

Rules void swapX(App app)	
C	A
app.x == oldValue	app.x = newValue; app.x;
int oldValue	int newValue
If x equals to	**Then make x to be equal to**
1	2
2	1

However, a similar rule sequence

RuleSequence void swapX(App app)	
C	A
app.x == oldValue	app.x = newValue; app.x;
int oldValue	int newValue
If x equals to	**Then make x to be equal to**
1	2
2	1

will fail because when x is equal to 1, the first rule will make it 2, and then the second rules will make it 1 again.

Relationships among Decision Tables

In most practical cases, business rules are not located in one file or in a single rule set, but rather are represented as a hierarchy of **inter-related rules** tables located in different files and directories - see Business Rules Repository. Frequently, the main Excel-file contains a main method that specifies the execution logic of multiple decision tables. In the Decision Model the table "Decision" plays this role. In many cases, the rule engine can execute decision tables directly from a Java program – see API.

Because OpenRules® interprets rules tables as regular methods, designers of rules frequently create special "processing flow" decision tables to specify the

conditions under which different rules should be executed. See examples of processing flow rules in such sample projects as Loan2 and LoanDynamics.

Simple AND / OR Conditions in Rules Tables

All conditions inside the same row (rule) are considered from left to right using the AND logic. For example, to express

```
if (A>5 && B >10) {do something}
```

you may use the rules table:

Rules void testAND(int a, int b)		
C1	C2	A1
a > 5	b>10	System.out.println(text)
String x	String x	String text
A > 5	**B > 10**	**Do**
X	X	Something

To express the OR logic

```
if (A>5 || B >10) {do something}
```

you may use the rules table:

Rules void testOR(int a, int b)		
C1	C2	A1
a > 5	b>10	System.out.println(text)
String x	String x	String text
A > 5	**B > 10**	**Do**
X		Something
	X	

Sometimes instead of creating a decision table it is more convenient to represent rules using simple Java expressions inside Method tables. For example, the above rules table may be easily represented as the following Method table:

Method void testOR(int a, int b)
if (a > 5 || b>10) System.out.println("Something");

Basic Rule Templates

Rule Templates allow rule designers to completely hide implementation details from business users. Rule templates are regular decision tables such as the "defineGreeting" table described above. However, if we want to use this table as a template for many other tables with the same structure but different rules, it is enough to use only the first 5 rows:

Rules void **defineGreeting**(App app, int hour)			
C1		A1	Signature with parameters Conditions and Actions identifiers
min <= hour && hour <= max		app.greeting = greeting;	Java snippets describe condition/action semantics
int min	int max	String greeting	Parameter types and names
Hour From	**Hour To**	**Set Greeting**	Business names for conditions and actions

We may use this decision table as a template to define different greeting rules for summer and winter time. An actual decision table *implements* the template table with particular rules:

Rules summerGreeting **implements defineGreeting**		
Hour From	**Hour To**	**Set Greeting**
0	10	Good Morning
11	18	Good Afternoon
19	22	Good Evening
23	24	Good Night

and

Rules winterGreeting **implements defineGreeting**		
Hour From	**Hour To**	**Set Greeting**
0	11	Good Morning
12	17	Good Afternoon
18	22	Good Evening
23	24	Good Night

Note that Rules tables "summerGreeting" and "winterGreeting" do not have technical information at all - Java snippets and a signature are defined only once and reside in the template-table "defineGreeting".

Learn more about Rule Templates below.

Horizontal and Vertical Decision Tables

Decision tables can be created in one of two possible formats:

- Vertical Format (default)
- Horizontal Format.

Based on the nature of the decision table, a rules creator can decide to use a vertical format (as in the examples above where concrete rules go vertically one after another) or a horizontal format where Condition and Action are located in the rows and the rules themselves go into columns. Here is an example of the proper horizontal format for the same decision table "helloWorld":

Rules void helloWorld(int hour) //horizontal				
Hour From	0	12	18	23
Hour To	11	17	22	24
Greeting	Good Morning	Good Afternoon	Good Evening	Good Night

OpenRules® automatically recognizes that a table has a vertical or a horizontal format. You can use Excel's Copy and Paste Special feature to transpose a decision table from one format to another.

Note. When a decision table has too many rules (more than you can see on one page) it is better to use the vertical format to avoid Excel's limitations: a worksheet has a maximum of 65,536 rows but it is limited to 256 columns.

Merging Cells

OpenRules® recognizes the powerful Cell Merging mechanism supported by Excel and other standard table editing tools. Here is an example of a rules table with merged cells:

Rules void testMerge(String value1, String value2)				
Rule	C1	C2	A1	A2
	value1.equals(val)	value2.equals(val)	out("A1: " + text);	out("A2: " + text);
	String val	String val	String text	String text
#	**Name**	**Value**	**Text 1**	**Text 2**
1	B	One	11+21	12
2		Two		22
3		Three	31	32
4	D		41	42

The semantics of this table is intuitive and described in the following table:

Value 1	Value 2	Applied Rules	Printed Results
B	One	1	A1: 11+21 A2: 12
B	Two	2	A1: 11+21 A2: 22
B	Three	3	A1: 31 A2: 32

D	Three	4	A1: 41 A2: 42
A	Two	none	
D	Two	none	

Restriction. We added the first column with rules numbers to avoid the known implementation restriction that the very first column (the first row for horizontal rule tables) cannot contain merged rows. More examples can be found in the standard rule project "Merge" - click here to analyze more rules.

Sub-Columns and Sub-Rows for Dynamic Arrays

One table column can consist of several sub-columns (see sub-columns "Min" and "Max" in the example above). You may efficiently use the Excel merge mechanism to combine code cells and to present them in the most intuitive way. Here is an example with an unlimited number of sub-columns:

As you can see, condition C6 contains 4 sub-columns for different combinations of rates. The cells in the Condition, code, parameters and display values, rows are merged. You can insert more sub-columns (use Excel's menu "Insert") to handle more rate combinations if necessary without any changes in the code. The parameter row is defined as a String array, `String[] rates`. The actual values of the parameters should go from left to right and the first empty value in a sub-

column should indicate the end of the array "*rates*". You can see the complete example in the rule table "Rule Family 212" in the file Loan1.xls.

If your decision table has a horizontal format, you may use multiple sub-rows in a similar way (see the example in file UpSell.xls).

Using Expressions inside Decision Tables

OpenRules® allows a rules designer to use "almost" natural language expressions inside decision tables to represent intervals of numbers, strings, dates, etc. You also may use Java expressions whenever necessary.

Integer and Real Intervals

You may use plain English expressions to define different intervals for integer and real decision variables inside rule tables. Instead of creating multiple columns for defining different ranges for integer and real values, a business user 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 but not 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 use many other ways to represent integer intervals as you usually do in plain English. The only limitation is the following: *min* should always go before *max*!

Similarly to integer intervals, one may use the predefined type **`FromToDouble`** to represent intervals of real numbers. The bounds of double intervals could be integer or real numbers such as [2.7; 3.14).

Comparing Integer and Real Numbers

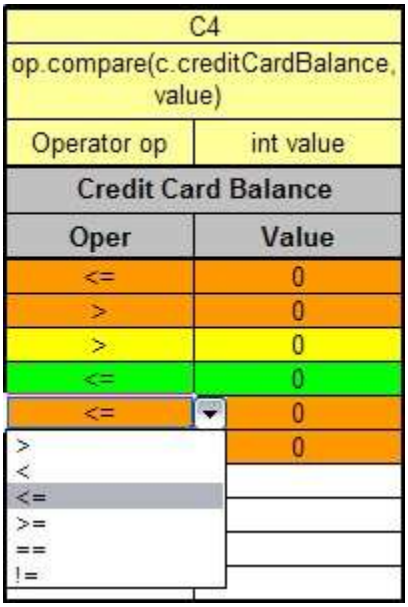
You may use the predefined type **`CompareToInt`** to compare a decision variable with an integer number that is preceded by a comparison operator. Examples of acceptable operators:

Cell Expression	Comment
<= 5	less or equals to 5
< 5	strictly less than 5
> 5	strictly more than 5
>= 5	more or equals to 5
!=	not equal to 5
5	equals to 5. Note that absence of a comparison operator means equality. You cannot use an explicit operator "=" (not to be confused with Excel's formulas).

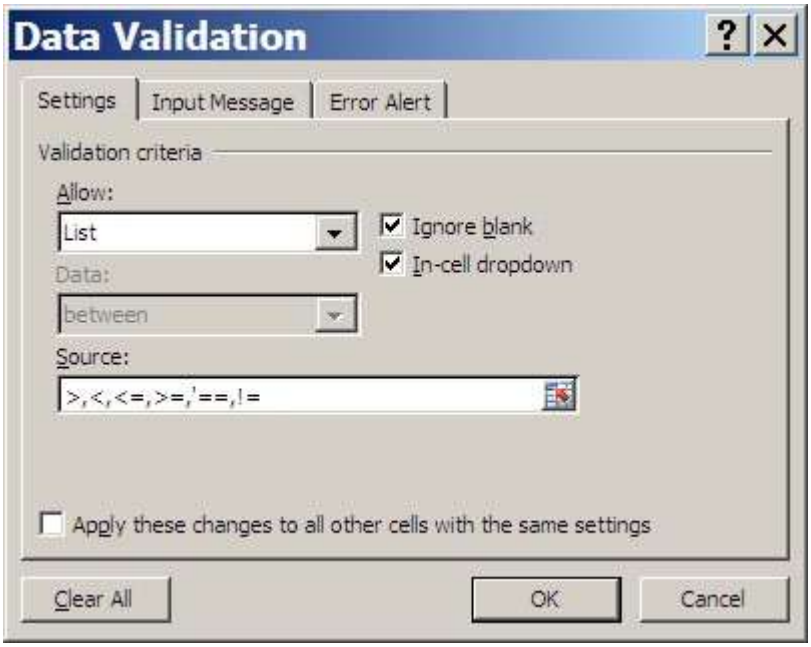
Similarly to **`CompareToInt`** one may use the predefined type **`CompareToDouble`** to represent comparisons with real numbers. The comparison values may be presented as integer or real numbers, e.g. "<= 25.4" and "> 0.5".

Using Comparison Operators inside Rule Tables

A user can employ a comparison operators such as "<" for "less" or ">" for "more" directly inside the rules. There are several ways to accomplish this. Here is an example from the rule table "Rule Family 212" (Loan1.xls):



You may use the Excel Data Validation menu to limit the choice of the operators:



Here the sign "==" has an apostrophe in front to differentiate it from an Excel formula. The actual implementation of possible comparison operators is provided as an example in the project "`com.openrules.tools`" (see `com.openrules.tools.Operator.java`). You may change them or add other operators. In addition to values of the type "int" you may also use Operator to compare long, double, and String types.

Comparing Dates

You may use the standard java.util.Date or any other Java Comparable type. Here is an example of comparing Dates:

C1	
op.compare(visit.date,date)	
Operator op	Date date
Operator	**Date**
==	2/15/2007
!=	1/1/2007
<=	2/15/2007
>	2/15/2007
<	2/15/2007

Please note that the current implementation compares dates without time.

Another way to use operators directly inside a table is to use expressions. In the example above, instead of having two sub-columns "Operator" and "Value" we could use one column and put an expression inside the cell:

```
{ c.creditCardBalance <= 0; }
```

The use of expressions is very convenient when you do not know ahead of time which operator will be required for which columns.

Comparing Boolean Values

If a parameter type is defined as "boolean", you are allowed to use the following values inside rule cells:

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

Here are examples:

C1
condition
boolean condition
IF Condition
TRUE
FALSE
Yes
NO
{ loan.additionalIncomeValidationNeeded; }

Sometimes, you want to use something like X or * just to indicate that a condition is satisfied or an action should be executed. For example in this table (from the standard project VacationDays)

Rules void DecisionTable(Test t)											
C1	t.age >= max	int max	**Age >=**		18	18	18	45	45	45	60
C2	t.age < min	int min	**Age <**	18	45	45	45	60	60	60	
C3	t.service >= max	int max	**Service >=**			25	40		25	40	
C4	t.service < min	int min	**Service <**		25	40		25	40		
A1	t.days = 22	String X	**Assign 22 days**	X	X	X	X	X	X	X	X
A2	t.days += 5	String X	**5 extra days**	X							
A3	t.days += 2	String X	**2 extra days**			X	X	X	X	X	X
A4	t.days += 3	String X	**3 extra days**							X	X

only actions marked with "X" will be executed. You can use any other character instead of "X".

Representing String Domains

Let's express a condition that validates if a customer's internal credit score is one of several acceptable rates such as "A B C" and "D F". To avoid the necessity to create multiple sub-columns for similar conditions, we may put all possible string values inside the same cell and separate them by spaces or commas. Here is an example of such a condition:

Condition
domain.contains(customer.internalCreditRating)
DomainString domain
Internal Credit Rating
A B C
D F
D F
A B C

Here we use the predefined type **`DomainString`** that defines a domain of strings (words) separated by whitespaces. The method "`contains(String string)`" of the class `DomainString` checks if the parameter "string" is found among all strings listed in the current "domain". You also may use the method "`containsIgnoreCase(String string)`" that allows you to ignore case during the comparison.

If possible values may contain several words, one may use the predefined type **`DomainStringC`** where "**`C`**" indicates that commas will be used as a string separator. For example, we may use **`DomainStringC`** to specify a domain such as "Very Hot, Hot, Warm, Cold, Very Cold".

Representing Domains of Numbers

If you need to represent domains of integer or double values, there are several predefined types similar to DomainString:

- `DomainInt`
- `DomainIntC`
- `DomainDouble`
- `DomainDoubleC`

For example, here is a condition column with eligible loan terms:

Condition
domain.contains(c.loanTerm)
DomainIntC domain
Eligible Loan Terms
24,36,72
36,72
72

Using Java Expressions

The use of Java expressions provides the powerful ability to perform calculations and test for complex logical conditions. While the writing of expressions requires some technical knowledge, it does not require the skills of a programmer. Real-world experience shows that business analysts frequently have a need to write these expressions themselves. It is up to the decision table designer to decide whether to show the expressions to business people or to hide them from view. Let's consider a decision table for "Income Validation" from the provided standard example Loan:

Rules void **ValidateIncomeRules**(LoanRequest loan, Customer customer)	
C1	A1
customer.monthlyIncome * 0.8 - customer.monthlyDebt > loan.amount/loan.term	loan.incomeValidationResult = result;
boolean condition	String result
IF **Income is Sufficient for the Loan**	**THEN** **Set Income Vaidation Result**
No	UNSUFFICIENT
Yes	SUFFICIENT

Here the actual income validation expression is hidden from business people inside "gray" technical rows, and a business person would only be able to choose between "Yes" or "No". However, the same table could be presented in this way:

Rules void **ValidateIncomeRules**(LoanRequest loan, Customer customer)	
C1	A1
condition == true	loan.incomeValidationResult = result;
boolean condition	String result
IF **Condition is True**	**THEN** **Set Income Validation Result**
	UNSUFFICIENT
:= customer.monthlyIncome * 0.8 - customer.monthlyDebt > loan.amount/loan.term	SUFFICIENT

Now, a user can both see and change the actual income validation condition.

Notes: Do not use Excel's formulas if you want the content to be recognized by the OpenRules® engine: use OpenRules® expressions instead.

If you want to start your cell with "=" you have to put an apostrophe in front of it i.e. '= to direct Excel not to attempt to interpret it as a formula.

Expanding and Customizing Predefined Types

All the predefined types mentioned above are implemented in the Java package `com.openrules.types`. You may get the source code of this package and expand and/or customize the proper classes. In particular, for internationalization purposes you may translate the English key words into your preferred language. You may change the default assumptions about inclusion/exclusion of bounds inside integer and real intervals. You may add new types of intervals and domains.

Performance Considerations

The use of expressions inside OpenRules® tables comes with some price - mainly in performance, for large decision tables. This is understandable because for every cell with an expression OpenRules® will create a separate instance of the proper Java class during rules execution. However, having multiple representation options allows a rule designer to find a reasonable compromise between performance and expressiveness.

RULE TEMPLATES

OpenRules® provides a powerful yet intuitive mechanism for compactly organizing enterprise-level business rules repositories. Rule templates allow rule designers to write the rules logic once and use it many times. With rule templates you may completely hide rules implementation details from business users. OpenRules® supports several rule templatization mechanisms from simple rule tables that inherit the exact structure of templates to partial template implementations.

Simple Rules Templates

Rule templates are regular decision tables that serve as structural prototypes for many other rule tables with the same structure but different content (rules). A simple rule template usually does not have rules at all but only specifies the table structure and implementation details for conditions and actions. Thus, a simple rule template contains the first 5 rows of a regular decision table as in the following example:

Rules void **defineGreeting**(App app, int hour)		
C1		A1
min <= hour && hour <= max		app.greeting = greeting;
int min	int max	String greeting
Hour From	**Hour To**	**Set Greeting**

Signature with parameters
Conditions and Actions identifiers
Java snippets describe condition/action semantics
Parameter types and names
Business names for conditions and actions

We may use this decision table as a template to define different greeting rules for summer and winter time. The actual decision tables will ***implement*** *(or* ***extend****)* the template table with particular rules:

Rules summerGreeting template defineGreeting		
Hour From	**Hour To**	**Set Greeting**
0	10	Good Morning
11	18	Good Afternoon
19	22	Good Evening
23	24	Good Night

and

Rules winterGreeting **template** defineGreeting		
Hour From	**Hour To**	**Set Greeting**
0	11	Good Morning
12	17	Good Afternoon
18	22	Good Evening
23	24	Good Night

Note that Rules tables "summerGreeting" and "winterGreeting" do not have technical information at all - Java snippets and a signature are defined only once and reside in the template-table "defineGreeting".

Along with the keyword "**template**" you may use other keywords:

- **implements**
- **implement**
- **extends**
- **extend**

We will refer to these rule tables created based on a template as "***template implementations***".

Simple templates require that the extended tables should have exactly the same condition and action columns.

Defining Default Rules within Templates

When many decision tables are created based on the same rule template, it could be inconvenient to keep the same default rules in all extended tables. As an alternative you may add the default rules directly to the template. The location of the default rules depends on the types of your rules tables.

Templates with Defaults Rules for Multi-Hit Tables

Multi-hit decision tables execute all their rules that are satisfied, allowing rules overrides. However, when conditions in all specified rules are not satisfied then a multi-hit table usually uses the first (!) rules to specify the default action. The rules from the template will be executed **before** the actual rules defined inside the extended tables.

Let's consider an example. You may notice that the rules tables above would not produce any greeting if the parameter "hour" is outside of the interval [0;24].

Let's assume that in this case we want to always produce the default greeting "How are you". To do this simply add one default rule directly to the template:

Rules void **defineGreeting**(App app, int hour)		
C1		A1
min <= hour && hour <= max		app.greeting = greeting;
int min	int max	String greeting
		How are you

This rule will be added at the beginning of all template implementations. This greeting will be produced if all other rules in the rule tables fail

A template for multi-hit tables could include more than one default rule each with different conditions - they all will be added to the beginning of the template implementation tables and will execute different default actions.

Templates with Defaults Rules for Single-Hit Tables

Single-hit decision tables usually end their execution when at least one rules is satisfied. However, when conditions in all specified rules are not satisfied then a single-hit table usually uses the **last** rule(s) to specify the default action(s). The rules from the template will be executed **after** the actual rules defined inside the template implementation.

Let's consider an example. We have shown that without modification, the rules tables above would not produce any greeting if the parameter "hour" is outside of the interval [0;24]. Instead of adding the same error message in both "summer" and "winter" rules, we could do the following:

- make our "defineGreeting" template a single-hit table by changing a return type from "void" to "String"
- - add the default reaction to the error in "hour" directly to the template:

Rules String **defineGreeting**(App app, int hour)		
C1		A1
min <= hour && hour <= max		app.greeting = greeting; return greeting;
int min	int max	String greeting
Hour From	**Hour To**	**Set Greeting**
		ERROR: Invalid Hour

Signature now returns String

Conditions and Actions identifiers

"return greeting;" has been added

Parameter types and names

Business names for conditions and actions

This rule will be added at the end of all template implementations tables. The error message will be return instead of a greeting when all other rules fail.

A template for single-hit tables could include more than one rule with different conditions - they all will be added at the end of the template implementation tables to execute different default actions.

Partial Template Implementation

Usually template implementation tables have exactly the same structure as the rule templates they extend. However, sometimes it is more convenient to build your own rules table that contains only some conditions and actions from already predefined rule templates. This is especially important when a library of rule templates for a certain type of business is used to create a concrete rules-based application. How can this be achieved?

The template implementation table uses its second row to specify the names of the used conditions and actions from the template. Let's consider an example.

The DebtResearchRules from the standard OpenRules® example "Loan Origination" may be used as the following template:

Rules void **DebtResearchRules**(LoanRequest loan, Customer c)										
C1	C2		C3	C4		C5		C6	C7	A1
c.mortgageHolder.equals(YN)	c.outsideCreditScore>min && c.outsideCreditScore<=max		c.loanHolder.equals(YN)	op.compare(c.creditCardBalance,value)		op.compare(c.educationLoanBalance,value)		contains(rates,c.internalCreditRating)	c.internalAnalystOpinion.equals(level)	loan.debtResearchResult = level;
String YN	int min	int max	String YN	Operator op	int value	Operator op	int value	String[] rates	String level	String level
IF Mortgage Holder	**AND Outside Credit Score**		**AND Loan Holder**	**AND Credit Card Balance**		**AND Education Loan Balance**		**AND Internal Credit Rating**	**AND Internal Analyst Opinion**	**THEN Debt Research Recommendations**
	Min	**Max**		**Oper**	**Value**	**Oper**	**Value**			

We may create a rule table that implements this template using only conditions C1, C2, C5, C6 and the action A1:

Rules **MyDebtResearchRules** template **DebtResearchRules**										
C1	C2		C5		C6					A1
IF Mortgage Holder	**AND Outside Credit Score**		**AND Education Loan Balance**		**AND Internal Credit Rating**					**THEN Debt Research Recommendations**
	Min	**Max**	**Oper**	**Value**						
Yes										**High**
No	100	550								**High**
No	550	900								**Mid**
No	550	900	>	0						**High**
No	550	900	<=	0	A	B	C			**High**
No	550	900	<=	0	D	F				**Mid**
No	550	900								**Low**
No	550	900	<=	0						**Low**
No	550	900	>	0	D	F				**High**
No	550	900	>	0	A	B	C			**Low**

The additional second row specifies which conditions and actions from the original template are selected by this rules table. The order of conditions and

actions may be different from the one defined in the template. Only names like "C2", "C6", and "A1" should be the same in the template and in its implementation. It is preferable to use unique names for conditions and actions inside templates. If there are duplicate names inside templates the first one (from left to right) will be selected. You may create several columns using the same condition and/or action names.

Templates with Optional Conditions and Actions

There is another way to use optional conditions and actions from the templates. If the majority of the template implementations do not use a certain condition from the template, then this condition may be explicitly marked as optional by putting the condition name in brackets, e.g. "[C3]" or "[Conditon-5]". In this case *it is not necessary to use the second row* to specify the selected conditions in the majority of the extended tables. For example, let's modify the DebtResearchRules template making the conditions C3, C4, and C7 optional:

Rules void **DebtResearchRules**(LoanRequest loan, Customer c)							
C1	C2	**[C3]**	**[C4]**	C5	C6	**[C7]**	A1

Now we can implement this template as the following rules table without the necessity to name all of the conditions and actions in the second row:

Rules MyDebtResearchRules template DebtResearchRules										
IF Mortgage Holder	**AND Outside Credit Score**		**AND Education Loan Balance**		**AND Internal Credit Rating**					**THEN Debt Research Recommendations**
	Min	**Max**	**Oper**	**Value**						
Yes										**High**
No	100	550								**High**
No	550	900								**Mid**
No	550	900	>	0						**High**
No	550	900	<=	0	A	B	C			**High**
No	550	900	<=	0	D	F				**Mid**
No	550	900								**Low**
No	550	900	<=	0						**Low**
No	550	900	>	0	D	F				**High**

No	550	900	>	0	A	B	C			**Low**

However, a template implementation that does want to use optional conditions will have to specify them explicitly using the second row:

Rules MyDebtResearchRules template DebtResearchRules													
C1	C2		C3	C4		C5		C6					A1
IF Mortgage Holder	**AND Outside Credit Score**		**AND Loan Holder**	**AND Credit Card Balance**		**AND Education Loan Balance**		**AND Internal Credit Rating**					**THEN Debt Research Recommendations**
	Min	**Max**		**Oper**	**Value**	**Oper**	**Value**						
Yes													**High**
No	100	550											**High**
No	550	900	Yes	<=	0								**Mid**
No	550	900	Yes	>	0	>	0						**High**
No	550	900	Yes	>	0	<=	0	A	B	C			**High**
No	550	900	Yes	>	0	<=	0	D	F				**Mid**
No	550	900	No	>	0								**Low**

Similarly, optional actions may be marked as [A1]" or "[Action3]".

Implementation Notes:

- Rule templates are supported for both vertical and horizontal rule tables.
- The keywords "**extends**" or "**implements**" may be used instead of the keyword "**template**"
- Template implementations cannot be used as templates themselves.

Templates for the Decision Model

The Decision Model is implemented using several templates located in the following files inside the configuration project "openrules.config":

- **DecisionTemlates.xls**: contains the following rule templates and methods for the Decision Model tables:
  - `DecisionTemplate(Decision decision)`: a template for the tables of type "`Decision`"
  - `initializeDecision()`: the method that initializes the current decision

- `decision()`: the method that returns the current decision
- `getGlossary()`: the method that returns the glossary
- `getDecisionObject(String nameofBusinessConcept)`: the method that returns a business object associated with the `BusinessConcept`
- `isTraceOn()`: returns `true` if the tracing of the decision model is on
- `DecisionObjectTemplate(Decision decision)`: a template for the table of the type "`DecisionObject`"
- `GlossaryTemplate(Decision decision)`: a template for the table of type "`Glossary`"
- Methods that return values of fact types based on their names:
  - `int getInt(String name)`
  - `double getReal(String name)`
  - `String getString(String name)`
  - `Date getDate(String name)`
  - `boolean getBool(String name)`
- Methods that set values of fact types based on their names:
  - `void getInt(String name, int value)`
  - `void getReal(String name, double value)`
  - `void getString(String name, String value)`
  - `void getDate(String name, Date value)`
  - `void getBool(String name, Boolean value)`
- Comparison methods that compare a fact type with a given "name", against a given "value", or another fact type using a given operator, "op":
  - `boolean compareInt(String name, String op, int value)`
  - `boolean compareInt(String name1, String op, String name2)`
  - `boolean compareReal(String name, String op, double value)`

        - `boolean compareReal(String name1, String op, String name2)`
        - `boolean compareBool(String name, String op, boolean value)`
        - `boolean compareBool(String name1, String op, String name2)`
        - `boolean compareDate(String name, String op, Date date)`
        - `boolean compareDate(String name1, String op, String name2)`
        - `boolean compareString(String name, String op, String value)`
        - `boolean compareDomain(String name, String op, String domain)`
    - the Environment table that includes the following references:
        - `RuleFamily${OPENRULES_MODE}Templates.xls`: where `${OPENRULES_MODE}` is an environment variable that has one of the following values:
            - `Execute` – the default value for Rule Family execution templates
            - `Validate` –for Rule Family validation templates
            - `Solve` – a reserve value for the future Rule Family solving templates using Rule Solver.
        - `RuleFamily1ExecuteTemplates.xls`: templates for RuleFamily1
        - `RuleFamily2ExecuteTemplates.xls`: templates for RuleFamily2
- **RuleFamilyExecuteTemplates.xls**: contains the following rule templates:
    - `RuleFamilyTemplate()`: a template for execution of the tables of the type "`RuleFamily`"
    - `customInitializeDecision()`: the method that can be used for initialization of custom objects

- **RuleFamilyValidateTemplates.xls**: contains the following rule templates:
  - `RuleFamilyTemplate()`: a template for validation of the tables of type "`RuleFamily`" against the domains defined in the glossary
  - `customInitializeDecision()`: the method that can be used for the initialization of custom objects
- **RuleFamily1ExecuteTemplates.xls**: contains the rule templates `RuleFamily1Template()` for execution of tables of type "`RuleFamily1`"
- **RuleFamily2ExecuteTemplates.xls**: contains the rule templates `RuleFamily2Template()` for execution of tables of type "`RuleFamily2`".

Decision Templates

The template "DecisionTemplate" contains two mandatory action columns with names "ActionPrint" and "ActionExecute" and three optional columns with the names "Condition", "ConditionAny", and "ActionAny". Here is an example of this template:

RuleSequence void DecisionTemplate(Decision decision)						
[Condition]		[ConditionAny]		ActionPrint	ActionExecute	[ActionAny]
getGlossary().compare($COLUMN_TITLE,op,value);		op.compare(value);		Log.info("Decision " + $TABLE_TITLE + ": " + name);		
Oper op	String value	Oper op	boolean value	String name	Object object	Object value
Fact Type		**Dynamic Condition**		**Decisions**	**Execute Rule Families**	**Title for Action Any**
					{ initializeDecision (decision); }	

Because you can use the same column "Condition" or "ConditionAny" as many times as you wish, you may create tables of type "Decision" with virtually unlimited complexity.

Rule Family Templates

The template "RuleFamilyTemplate" serves as a template for all standard rule families. All columns in this template are conditional meaning their names are always required. Here are the first two rows of this template:

Rules String **RuleFamilyTemplate**()							
[Condition]	[ConditionAny]	[If]	[Conclusion]	[Action]	[ActionAny]	[Then]	[Message]

The template "RuleFamily1Template" serves as a template for all rule families of type "RuleFamily1". Here are the first two rows of this template:

Rules void **RuleFamily1Template**()							
[Condition]	[ConditionAny]	[If]	[Conclusion]	[Action]	[ActionAny]	[Then]	[Message]

The template "RuleFamily2Template" serves as a template for all rule families of type "RuleFamily2". Here are the first two rows of this template:

RuleSequence void **RuleFamily1Template**()							
[Condition]	[ConditionAny]	[If]	[Conclusion]	[Action]	[ActionAny]	[Then]	[Message]

Customization

A user may move the above files from "openrules.config" to different locations and modify the rule family templates (and possible other templates). For example, to have different types of messaging inside a custom decision model, a user may add two more columns to the template "RuleFamilyTemplate":

- `Warning`: similar to `Message` but can use a different log for warning only
- `Errors`: similar to `Message` but can use a different log for errors only.

Rewriting the method `customInitializeDecision()` allows a user to initialize custom objects. Contact support@openrules.com if you need help with more complex customization of the Decision Model templates.

OPENRULES® API

OpenRules® provides an Application Programming Interface (API) that defines a set of commonly-used functions:

- Creating a rule engine associated with a set of Excel-based rules
- Creating a decision associated with a set of Excel-based rules
- Executing different rule sets using application specific business objects
- Creating a web session and controlling client-server interaction.

OpenRulesEngine API

OpenRulesEngine is a Java class provide by OpenRule® to execute different rule sets and methods specified in Excel files using application-specific business objects. OpenRulesEngine can be invoked from any Java application using a simple Java API or a standard JSR-94 interface.

Engine Constructors

OpenRulesEngine provides an interface to execute rules and methods defined in Excel tables. You can see examples of how OpenRulesEngine is used in basic rule projects such as HelloJava, DecisonHellJava, HelloJsr94 and web applications such as HelloJsp, HelloForms, and HelloWS. To use OpenRulesEngine inside your Java code you need to add an import statement for `com.openrules.ruleengine.OpenRulesEngine` and make sure that `openrules.all.jar` is in the classpath of your application. This jar and all 3rd party jar-files needed for OpenRules® execution can be found in the subdirectory `openrules.config/lib` of the standard OpenRules® installation. You may create an instance of OpenRulesEngine inside of your Java program using the following constructor:

```
public OpenRulesEngine(String xlsMainFileName)
```

where xlsMainFileName parameter defines the location for the main xls-file. To specify a file location, OpenRules® uses an **URL pseudo-protocol notation** with prefixes such as ***"file:", "classpath:", "http://", "ftp://", "db:"***, etc. Typically, your main xls-file Main.xls is located in the subdirectory "rules/main" of your Java project. In this case, its location may be defined as "`file:rules/main/Main.xls`". If your main xls-file is located directly in the project classpath, you may define its location as "`classpath:Main.xls`". Use a URL like `http://www.example.com/rules/Main.xls` when `Main.xls` is located at a website. All other xls-files that can be invoked from this main file are described in the table "Environment" using include-statements.

You may also use other forms of the OpenRulesEngine constructor. For example, the constructor

```
OpenRulesEngine(String xlsMainFileName, String methodName)
```

allows you to also define the main method from the file xlsMainFileName that will be executed during the consecutive runs of this engine.

Here is a complete example of a Java module that creates and executes a rule engine (see HelloJava project):

```
package hello;
import com.openrules.ruleengine.OpenRulesEngine;
public class RunHelloCustomer {
  public static void main(String[] args) {
    String fileName = "file:rules/main/HelloCustomer.xls";
    String methodName = "helloCustomer";
    OpenRulesEngine engine = new OpenRulesEngine(fileName);
    Customer customer = new Customer();
    customer.setName("Robinson");
    customer.setGender("Female");
```

```
    customer.setMaritalStatus("Married");

    Response response = new Response();
    Object[] objects = new Object[] { customer, response };
    engine.run(methodName,objects);
    System.out.println("Response: " +
       response.getMap().get("greeting") + ", " +
       response.getMap().get("salutation") +
       customer.getName() + "!" );
  }
}
```

As you can see, when an instance "engine" of OpenRulesEngine is created, you can create an array of Java objects and pass it as a parameter of the method "run".

Engine Runs

The same engine can run different rules and methods defined in its Excel-files. You may also specify the running method using

```
setMethod(String methodName);
```

or use it directly in the engine run:

```
engine.run(methodName,businessObjects);
```

If you want to pass to OpenRulesEngine only one object such as "customer", you may write something like this:

```
engine.run("helloCustomer",customer);
```

If you do not want to pass any object to OpenRulesEngine but expect to receive some results from the engine's run, you may use this version of the method "run":

```
String[] reasons = (String[]) engine.run("getReasons");
```

Undefined Methods

OpenRulesEngine checks to validate if all Excel-based tables and methods are actually defined. It produces a syntax error if a method is missing. Sometimes, you want to execute a rule method/table from an Excel file but only if this method is actually present. To do this, you may use this version of the method "run":

```
boolean mayNotDefined = true;
engine.run(methodName, businessObjects, mayNotDefined);
```

In this case, if the method "methodName" is not defined, the engine would not throw a usual runtime exception "The method <name> is not defined" but rather will produce a warning and will continue to work. The parameter "mayNotDefined" may be used similarly with the method "run" with one parameter or with no parameters, e.g.

```
engine.run("validateCustomer", customer, true);
```

How to invoke rules from other rules if you do not know if these rules are defined? It may be especially important when you use some predefined rule names in templates. Instead of creating an empty rules table with the needed name, you want to use the above parameter "mayNotDefined" directly in Excel. Let's say you need to execute rules tables with names such as "NJ_Rules" or "NY_Rules" from another Excel rules table but only if the proper state rules are actually defined. You may do it by calling the following method from your rules:

Method void runStateRules(OpenRulesEngine engine, Customer customer, Response response)

String methodName = customer.state + "_Rules";
Object[] params = new Object[2];
params[0] = customer;
params[1] = response;
engine.run(methodName, params, true);

We assume here that all state-specific rules ("NJ_Rules", "NY_Rules", etc.) have two parameters, "customer" and "response". To use this method you need to pass the current instance of OpenRulesEngine from your Java code to your main Excel file as a parameter "engine". If you write an OpenRules Forms application, this instance of the OpenRulesEngine is always available as `dialog().getEngine()`, otherwise you have to provide access to it, e.g. by attaching it to one of your own business objects such as Customer.

By default OpenRules will produce a warning when the required Excel rules table or method is not available. You may suppress such warnings by calling:

```
engine.turnOffNotDefinedWarning();
```

Accessing Password Protected Excel Files

Some Excel workbooks might be encrypted (protected by a password) to prevent other people from opening or modifying these workbooks. Usually it's done using Excel Button  and then **Prepare** plus **Encrypt Document**. OpenRules Engine may access password-protected workbooks by calling the following method just before creating an engine instance:

```
OpenRulesEngine.setCurrentUserPassword("password");
```

Instead of "password" you should use the actual password that protects your main and/or other Excel files. Only one password may be used by all protected Excel files that will be processed by one instance of the OpenRulesEngine created after this call. This call does not affect access to unprotected files. The standard project "HelloJavaProtected" provides an example of the protected Excel file - use the word "password" to access the file "HelloCustomer.xls".

Note. The static method "*setCurrentUserPassword*" of the class OpenRulesEngine actually sets the BIFF8 encryption/decryption password for the current thread. The use of a "null" string will clear the password.

Engine Attachments

You may attach any Java object to the OpenRulesEngine using methods `setAttachment(Object attachment)` and *`getAttachment`*`()`.

Engine Version

You may receive a string with the current version number of the OpenRulesEngine using the method *`getVersion`*`()`.

Dynamic Rules Updates

If a business rule is changed, OpenRulesEngine automatically reloads the rule when necessary. Before any engine's run, OpenRulesEngine checks to determine if the main Excel file associated with this instance of the engine has been changed. Actually, OpenRulesEngine looks at the latest modification dates of the file xlsMainFileName. If it has been modified, OpenRulesEngine re-initializes itself and reloads all related Excel files. You can shut down this feature by executing the following method:

```
engine.setCheckRuleUpdates(false);
```

Decision API

OpenRules® provides a special API for the Decision Model using the Java class "`Decision`". The following example from the standard project "Decision1040EZ" demonstrates the use of this API.

```
public class Main {

   public static void main(String[] args) {
         String fileName = "file:rules/main/Decision.xls";
         OpenRulesEngine engine = new OpenRulesEngine(fileName);
         Decision decision = new Decision("Apply1040EZ",engine);
         DynamicObject taxReturn =
               (DynamicObject) engine.run("getTaxReturn");
         engine.log("=== INPUT:\n" + taxReturn);
         decision.put("taxReturn",taxReturn);
         decision.execute();
         engine.log("=== OUTPUT:\n" + taxReturn);
   }
}
```

Here we first created an instance `engine` of the class `OpenRulesEngine` and used it to create an instance `decision` of the class `Decision`. We used the `engine` to get an example of the object `taxReturn` that was described in Excel data tables:

```
DynamicObject taxReturn =
            (DynamicObject) engine.run("getTaxReturn");
```

Then we added this object to the `decision`:

```
decision.put("taxReturn",taxReturn);
```

and simply executed `decision`:

```
decision.execute();
```

The decision model is supposed to modify certain attributes inside the object `decision` and we display all of them before and after the `decision` execution.

Decision Constructors

The class `Decision` provides the following constructor:

```
public Decision(String decisionName, String xlsMainFileName)
```

where "`decisionName`" is the name of the main table of the type "Decision" and "`xlsMainFileName`" is the same parameter as in the OpenRulesEngine's constructor that defines a location for the main xls-file.

There is also another constructor:

```
public Decision(String decisionName, OpenRulesEngine engine)
```

where the parameter `OpenRulesEngine engine` refers to an already created instance of the `OpenRulesEngine` as in the above example.

Each decision has an associated object of type `Glossary`. When a decision is created, it first executes the table "glossary" that must be defined in our rules repository. It fills out the glossary, a step that applies to all consecutive decision executions. You may always access the glossary by using the method

```
Glossary glossary = decision.getGlossary();
```

Decision Parameters

The class Decision is implemented as a subclass of the standard Java class HashMap. Thus, you can put any object into the decision similarly as we did above:

```
decision.put("taxReturn",taxReturn);
```

You may access any object previously put into the decision by calling the method get(name) as in the following example:

```
TaxReturn taxReturn = (TaxReturn)decision.get("taxReturn");
```

You may set a special parameter

```
decision.put("trace","Off");
```

to tell your decision to turn off the tracing . You may use "On" to turn it on again.

Decision Runs

After defining decision parameters, you may execute the decision as follows:

```
decision.execute();
```

This method will execute your decision model starting from the table of type "Decision" whose name was specified as the first parameter of the decision's constructor.

You may reset the parameters of your decision and execute it again without the necessity of constructing a new decision. This is very convenient for multi-transactional systems where you create a decision once by instantiating its glossary, and then you execute the same decision multiple times but with different parameters. To make sure that it is possible, the `Decision's` method `execute()` calls Excel's method "`decisionObjects`" each time before actually executing the decision.

There is one more form of this method:

```
decision.execute(String methodName);
```

It is used within Excel when you want to execute another Excel method. It is implemented as follows:

```
public Object execute(String methodName) {
      return getEngine().run(methodName);
}
```

Decision Execution Modes

Before executing a decision model you may validate it by setting a special "validation" mode. Here is a code example:

```
String fileName = "file:rules/main/Decision.xls";
System.setProperty("OPENRULES_MODE", "Validate");
Decision decision = new Decision("DetermineDecisionFact",fileName);
```

During the validation along with regular syntax check OpenRules® will validate if the values for conditions and actions inside all rule families corresponds to their glossary domains (if they are defined).

As you can see, the system property `"OPENRULES_MODE"` defines which mode to use. By default this property is set to `"Execute"`. If you create an OpenRulesEngine before creation a Decision, you need to set this property first.

JSR-94 Implementation

OpenRules® provides a reference implementation of the JSR94 standard known as Java Rule Engine API (see http://www.jcp.org/en/jsr/detail?id=94). The complete OpenRules® installation includes the following projects:

JSR-94 Project	Description
lib.jsr94	This project contains the standard jsr94-1.0 library
com.openrules.jsr94	This is an OpenRules®'s reference implementation for the JSR94 standard and includes the source code. It uses OpenRulesEngine to implement RuleExecutionSet
HelloJsr94	This is an example of using JSR94 for simple rules that generate customized greetings
HelloJspJsr94	HelloJspJsr94 is similar to HelloJsp but uses the OpenRules® JSR-94 Interface to create and run OpenRulesEngine for a web application.

Multi-Threading

OpenRulesEngine is thread-safe and works very efficiently in multi-threaded environments supporting real parallelism. OpenRulesEngine is stateless, which allows a user to create *only one* instance of the class OpenRulesEngine, and then share this instance between different threads. There are no needs to create a pool of rule engines. A user may also initialize the engine with application data common for all threads, and attach this data directly to the engine using the methods `setAttachment(Object attachment)`. Different threads will receive this instance of the rule engine as a parameter, and will safely run various rules in parallel using the same engine.

The complete OpenRules® installation includes an example "`HelloFromThreads`" that demonstrates how to organize a parallel execution of

the same OpenRulesEngine's instance in different threads and how to measure their performance.

INTEGRATION WITH JAVA AND XML

Java Classes

OpenRules® allows you to externalize business logic into xls-files. However, these files can still use objects and methods defined in your Java environment. For example, in the standard example "`RulesRepository`" all rule tables deal with the Java object `Appl` defined in the Java package `myjava.package1`. Therefore, the proper Environment table inside file Main.xls (see above) contains a property "`import.java`" with the value "`myjava.package1.*`":

Environment	
import.java	myjava.packA1.*
include	SubCategoryA1/RulesA11.xls
	SubCategoryA1/RulesA12.xls

The property "`import.java`" allows you to define all classes from the package following the standard Java notation, for example "`hello.*`". You may also import only the specific class your rules may need, as in the example above. You can define a separate property "`import.java`" for every Java package used or merge the property " `import.java` " into one cell with many rows for different Java packages. Here is a more complex example:

Environment	
import.static	com.openrules.tools.Methods
import.java	my.bom.*
	my.impl.*
	my.inventory.*
	com.openrules.ml.*
	my.package.MyClass

	com.3rdparty.*
include	../include/Rules1.xls
	../include/Rules2.xls

Naturally the proper jar-files or Java classes should be in the classpath of the Java application that uses these rules.

If you want to use static Java methods defined in some standard Java libraries and you do not want to specify their full path, you can use the property "`import.static`". The static import declaration imports static members from Java classes, allowing them to be used in Excel tables without class qualification. For example, many OpenRules® sample projects use static methods from the standard Java library *com.openrules.tools* that includes class Methods. So, many Environment tables have property "`import.static`" defined as "`com.openrules.tools.Methods`". This allows you to write

```
out("Rules 1")
```

instead of

```
Methods.out("Rules 1")
```

XML Files

Along with Java classes, OpenRules® tables can use objects defined in XML files. For example, the standard sample project HelloXMLCustomer uses an object of type Customer defined in the file Customer.xml located in the project classpath:

```
<Customer
     name="Robinson"
     gender="Female"
     maritalStatus="Married"
     age="55"
/>
```

The xls-file, HelloXmlCustomer.xls, that deals with this object includes the following Environment table:

Environment	
import.static	com.openrules.tools.Methods
import.schema	**classpath:/Customer.xml**
import.java	hello.Response
include	include/HelloRules.xls

The property, "`import.schema`", specifies the location of the proper xml-file, in this case "`classpath:/Customer.xml`". Of course, you can use any other location in your local file system that starts with the prefix "`file:`". This example also tells you that this Excel file uses:

1. static Java methods defined in the standard OpenRules® package "`com.openrules.tools.Methods`"
2. xml-file "`classpath:/Customer.xml`"
3. Java class "`Response`" from a package "`hello`"
4. include-file "HelloRules.xls" which is located in the subdirectory "include" of the directory where the main xls file is located.

The object of the type "`Customer`" can be created using the following API:

```
Customer customer = Customer.load("classpath:/Customer.xml");
```

You may use more complex structures defined in xml-files. For example, the project `HelloXMLPeople` uses the following xml-file:

```
<?xml version="1.0" encoding="UTF-8"?>
<People type="Array of Person(s)">
  <Person name="Robinson" gender="Female" maritalStatus="Married"
age="55" />
           <Person name="Robinson" gender="Female"
maritalStatus="Single" age="23" />
           <Person name="Robinson" gender="Male"
maritalStatus="Single" age="17" />
           <Person name="Robinson" gender="Male"
maritalStatus="Single" age="3" />
</People>
```

The method that launches greeting rules for every `Person` from an array `People` is defined as:

Method void **helloPeople()**

```
int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
App app = new App();
defineGreeting(hour, app);
// define and greet People from the XML file People.xml
People people = People.load("classpath:/People.xml");
for(int i = 0; i < people.Person.length; ++i)
{
   People.Person person = people.Person[i];
   defineSalutation(person,app);
   //greet Person
   System.out.println(app.greeting+", "+app.salutation+" "+person.name+"!");
}
```

DATA MODELING

OpenRules® includes an ability to define new data/object types and creates the objects of these types directly in Excel. It allows business analysts to do Rule Harvesting by defining business terms and facts without worrying about their implementation in Java, C#, or XML. It also provides the ability to **test** the business rules in a pre-integrated mode. To do standalone rule testing, a designer of rules and forms specifies his/her own data/object types as Excel tables and creates instances of objects of these types passing them to the rule tables. We describe how to do it in the sections below.

There is one more important reason why a business or even a technical specialist may need data modeling abilities without knowing complex software development techniques. In accordance with the SOA principle of loosely coupled services, rule services have to specify what they actually need from the objects defined in an external environment. For example, if an object "Insured" includes attributes related to a person's military services, it does not require that all business rules that deal with the insured be interested in those attributes. Such encapsulation of only the essential information in the Excel-based data types, together with live process modeling, allows OpenRules® to complete the rules modeling cycle without leaving Excel.

OpenRules® provides the means to make business rules and forms independent of a concrete implementation of such concepts. The business logic expressed in the decision tables should not depend on the implementation of the objects these rules are dealing with. For example, if a rule says: “If driver's age is less than 17 then reject the application” the only thing this business rule should "know" about the object “driver” is the fact that it has a property “age” and this property has a type that support a comparison operator “<” with an integer. It is a question of configuration whether the Driver is a Java class or an XML file or a DB table from a legacy system. Similarly, if a form has an input field "Driver's Age", the form should be able to accept a user's input into this field and automatically convert it into the proper object associated with this field independently of how this object was implemented.

Thus, OpenRules® supports data source independent business rules (decision tables) and web forms. Your business rules can work with an object of type Customer independently of the fact that this type is defined as a Java class, as an XML file or as an Excel table. You can see how it can be done using examples HelloJava, HelloXML, and HelloRules from the OpenRules®'s standard installation. It is a good practice to start with Excel-based data types. Even if you later on switch to Java classes of other data types, you would always be able to reuse Excel-based types for standalone testing of your rules-based applications.

Datatype and Data Tables

OpenRules® allows a non-technical user to represent different data types directly in Excel and to define objects of these types to be used as test data. Actually, it provides the ability to create Excel, Intelligent Business Glossaries, that, in turn, define problem specific business terms and facts. At the same time, a business glossary can include data types specified outside Excel, for example in Java classes or in XML files. Here is an example of a simple data type "PersonalInfo":

Datatype PersonalInfo	
String	id
String	firstName
String	middleInitial
String	lastName
String	address
String	appartment
String	city
String	state
String	zipCode

Now we can create several objects of this type "`PersonalInfo`" using the following data table:

Data PersonalInfo personalInformation			
id	ID	He	She
firstName	First Name	John	Mary
middleInitial	Middle Initial	N.	A.
lastName	Last Name	Smith	Smith
address	Address	25 Maple Street	
appartment	appartment	Apt. 3C	
city	City	Edison	
state	State	NJ	
zipCode	ZipCode	08840	

We can reference to these objects inside rules or forms as in the following snippets:

```
out(personalInformation["He"].lastName);
if (personalInformation["She"].state.equals("NJ")) ...
```

You may use one datatype (such as `PersonalInfo`) to define a more complex *aggregate datatype,* like TaxReturn in this example:

Datatype TaxReturn	
PersonalInfo	taxPayer
PersonalInfo	spouse
boolean	marriedFillingJointly
boolean	claimedAsDependent
boolean	spouseClaimedAsDependent
double	wages
double	taxableInterest
double	unemploymentCompensation
double	adjustedGrossIncome
double	dependentAmount
double	taxableIncome
double	taxWithheld
double	earnedIncomeCredit
double	totalPayments
double	tax
double	refund

You may even create an object of the new composite type "TaxReturn" using references to the objects "He" and "She" as in this example:

Data TaxReturn **taxReturns**					
taxPayer	spouse	wages	taxableInterest	taxWithheld	earnedIncomeCredit
>personalInformation	>personalInformation				
TaxPayer	Spouse	Wages	Taxable Interest	Tax Withheld	Earned Income Credit
He	She	32026	1450	4530	230

Now we can reference these objects from inside rules or forms as in the following snippet:

```
out(taxReturn[0].taxPayer.lastName);
```

The above tables may remind you of traditional database tables simply presented in Excel. While these examples give you an intuitive understanding of OpenRules® Datatype and Data tables, the next sections will provide their formal descriptions.

You may use a type of table "**Variable**". These tables are similar to the Data tables but instead of arrays of variables they allow you to create separate instances of objects directly in Excel files. Here is a simple example:

Variable Customer mary			
name	age	gender	maritalStatus
Name	Age	Gender	Marital Status
Mary Brown	5	Female	Single

The variable "mary" has type Customer and can be used inside rules or passed back from an OpenRulesEngine to a Java program as a regular Java object. As usual, the object type Customer can be defined as a Java class, an Excel Datatype, or an xml structure.

How Datatype Tables Are Organized

Every Datatype table has the following structure:

Datatype tableName	
AttributeType1	AttrubuteName1
AttributeType2	AttrubuteName2
..	..
..	..

The first "signature" row consists of two merged cells and starts with the keyword "Datatype". The "tableName" could be any valid one word identifier of the table (a combination of letters and numbers). The rows below consist of two cells with an attribute type and an attribute name. Attribute types can be the basic Java types:

- `boolean`
- `char`
- `int`
- `double`
- `long`

- `String (java.lang.String)`
- `Date (java.util.Date)`

You may also use data types defined:

- in other Excel Datatype tables
- in any Java class with a public constructor with a single parameter of the type String
- as one-dimensional arrays of the above types.

The datatype "PersonalInfo" gives an example of a very simple datatype. We can define another datatype for a social security number (SSN):

Datatype SSN	
String	ssn1
String	ssn2
String	ssn3

and add a new attribute of this type to the datatype "PersonalInfo":

Datatype PersonalInfo	
String	id
String	firstName
String	middleInitial
String	lastName
String	address
String	appartment
String	city
String	state
String	zipCode
SSN	ssn

It is interesting that these changes do not affect the already existing data objects defined above (like `personalInformation["He"]`) - their SSNs just will not be defined.

Implementation Restriction. Make sure that the very first attribute in a Datatype table has type String or your own type (but not a basic Java type like int).

The following example demonstrates how to create a Data table for a Datatype that includes one-dimensional arrays:

Datatype Order	
String	**number**
String[]	**selectedItems**
String[]	**offeredItems**
double	**totalAmount**
String	**status**

Here is an example of the proper Data table:

Data Order orders			
number	selectedItems	totalAmount	status
Number	**Selected Items**	**Total Amount**	**Status**
6P-U01	INTRS-PGS394	3700	In Progress
	INTRS-PGS456		
	Paste-ARMC-2150		

You may also present the same data in the following way:

Data Order orders				
number	selectedItems			totalAmount
Number	**Selected Items**			**Total Amount**
	Item 1	**Item 2**	**Item 3**	
6P-U01	INTRS-PGS394	INTRS-PGS456	Paste-ARMC-2150	3700

How Data Tables Are Organized

Every Datatype table has a vertical or horizontal format. A typical vertical Data table has the following structure:

Data datatypeName tableName			
AttributeName1 from "datatypeName"	AttributeName2 from "datatypeName"	AttributeName3 from "datatypeName"	...
Display value of the AttributeName1	**Display value of the AttributeName2**	**Display value of the AttributeName3**	**...**
data	data	data	...
data	data	data	...
...	...	...	...

The first "signature" row consists of two merged cells and starts with the keyword "Data". The next word should correspond to a known datatype: it can be an already defined Excel Datatype table or a known Java class or an XML file. The "tableName" is any one word valid identifier of the table (a combination of letters and numbers).

The second row can consists of cells that correspond to attribute names in the data type "datatypeName". It is not necessary to define all attributes, but at least one should be defined. The order of the columns is not important.

The third row contains the display name of each attribute (you may use unlimited natural language).

All following rows contain data values with types that correspond to the types of the column attributes.

Here is an example of the Data table for the datatype "`PersonalInfo`" defined in the previous section (with added SSN):

Data PersonalInfo personalInformation						
id	firstName	middleInitial	lastName	ssn.ssn1	ssn.ssn2	ssn.ssn3
ID	First Name	Middle Initial	Last Name	SSN1	SSN2	SSN3
He	John	N.	Smith	164	86	2298
She	Mary	A.	Smith	627	35	1293

The table name is "`personalInformation`" and it defines an array of objects of the type `PersonalInfo`. The array shown consists only of two elements `personalInformation[0]` for John and `personalInformation[1]` for Mary. You may add as many data rows as necessary.

The attributes after the SSN attribute have not been defined. Please, note that the references to the aggregated data types are defined in a natural way (`ssn.ssn1, ssn.ssn2, ssn.ssn3`) using the dot-convention.

As you can see from this example, the vertical format may not be very convenient when there are many attributes and not so many data rows. In this case, it could be preferable to use a horizontal format for the data tables:

Data datatypeName tableName					
AttributeName1 from "datatypeName"	Display value of the AttributeName1	data	data	data	...
AttributeName2 from "datatypeName"	Display value of the AttributeName2	data	data	data	...
AttributeName3 from "datatypeName"	Display value of the AttributeName3	data	data	data	...
...	..	...	...	...	...

Here is how our data table will look when presented in the horizontal format:

Data PersonalInfo personalInformation			
id	ID	He	She
firstName	First Name	John	Mary
middleInitial	Middle Initial	N.	A.
lastName	Last Name	Smith	Smith
ssn.ssn1	SSN1	164	627
ssn.ssn2	SSN2	86	35
ssn.ssn3	SSN3	2298	1293
address	Address	25 Maple Street	
appartment	appartment	Apt. 3C	
city	City	Edison	
state	State	NJ	
zipCode	ZipCode	08840	

Predefined Datatypes

OpenRules® provides predefined Java classes to create data tables for arrays of integers, doubles, and strings. The list of predefined arrays includes:

1. `ArrayInt` - for arrays of integer numbers, e.g.:

Method int[] getTerms()
return ArrayInt.getValues(terms);

Data ArrayInt terms
value
Term
36
72
108
144

2. `ArrayDouble` - for arrays of real numbers, e.g.:

Method double[] getCosts()
return ArrayDouble.getValues(costs);

Data ArrayDouble costs
value
Costs
$295.50
$550.00
$1,000.00
$2,000.00
$3,295.00
$5,595.00
$8,895.00

3. `ArrayString` - for arrays of strings, e.g.:

Method String[] getRegions()
return ArrayString.getValues(regions);

Data ArrayString regions
value
Region
NORTHEAST
MID-ATLANTIC
SOUTHERN
MIDWEST
MOUNTAIN
PACIFIC-COAST

These arrays are available from inside an OpenRules® table by just calling their names: `getTerms`(), `getCosts`(), `getRegions`(). You may also access these arrays from a Java program, using this code:

```
OpenRulesEngine engine =
        new OpenRulesEngine("file:rules/Data.xls");
int[] terms = (int[])engine.run("getTerms");
```

The standard installation includes a sample project "DataArrays", that shows how to deal with predefined arrays.

Accessing Excel Data from Java - Dynamic Objects

You can access objects created in Excel data tables from your Java program. These objects have a predefined type `DynamicObject`. Let's assume that you defined your own Datatype, Customer, and created an array of customers in Excel:

Data Customer customers			
name	maritalStatus	gender	age
Customer Name	**Marital Status**	**Gender**	**Age**
Robinson	Married	Female	24
Smith	Single	Male	19

Method Customer[] getCustomers()
return customers;

In you Java program you may access these objects as follows:

```
OpenRulesEngine engine =
        new OpenRulesEngine("file:rules/Data.xls");
DynamicObject[] customers =
(DynamicObject[])engine.run("getCustomers");
System.out.println("\nCustomers:");
for(int i=0; i<customers.length; i++)
        System.out.println("\t"+customers[i]);
```

This code will print:

```
Customer(id=0){
	name=Robinson
	age=24
	gender=Female
	maritalStatus=Married
}

Customer(id=1){
	name=Smith
	age=19
	gender=Male
	maritalStatus=Single
}
```

You may use the following methods of the class DynamicObject:

```
public Object getFieldValue(String name);

public void setFieldValue(String name, Object value);
```

For example,

```
String gender = (String) customers[0].getFieldValue("gender");
```

will return "Female", and the code

```
customer.setFieldValue("gender", "Male");
customer.setFieldValue("age", 40);
```

will change the gender of the object customers[0] to "Male" and his age to 40.

How to Define Data for Aggregated Datatypes

When one Datatype includes attributes of another Datatype, such datatypes are usually known as *aggregated datatypes*. You have already seen an example of an aggregated type, `PersonalInfo`, with the subtype SSN. `Similarly`, you may have two datatypes, `Person` and `Address`, where type Person has an attribute "`address`" of the type `Address`. You may create a data table with type Person using aggregated field names such as "`address.street`", "`address.city`", "`address.state`", etc. The subtype chain may have any length, for example "`address.zip.first5`" or "`address.zip.last4`". This feature very

conveniently allows a compact definition of test data for complex interrelated structures.

Finding Data Elements Using Primary Keys

You may think about a data table as a database table. There are a few things that make them different from traditional relational tables, but they are friendlier and easier to use in an object-oriented environment. The very first attribute in a data table is considered to be its *primary key*. For example, the attribute "id" is a primary key in the data table "personalInformation" above. You may use values like "He" or "She" to refer to the proper elements of this table/array. For example, to print the full name of the person found in the array "personalInformation", you may write the following snippet:

```
PersonalInfo pi = personalInformation["He"];
out(pi.fisrtName + " " + pi.middeInitial + ". " + pi.lastName);
```

Cross-References Between Data Tables

The primary key of one data table could serve as a foreign key in another table thus providing a cross-reference mechanism between the data tables. There is a special format for data tables to support cross-references:

Data datatypeName tableName			
AttributeName1 from "datatypeName"	AttributeName2 from "datatypeName"	AttributeName3 from "datatypeName"	...
>referencedDataTable1		>referencedDataTable2	
Display value of the AttributeName1	Display value of the AttributeName2	Display value of the AttributeName3	...
data	data	data	...
data	data	data	...
...	...	...	...

This format adds one more row, in which you may add references to the other data tables, where the data entered into these columns should reside. The sign ">" is a special character that defines the reference, and "referencedDataTable" is the name of another known data table. Here is an example:

Data TaxReturn **taxReturns**					
taxPayer	spouse	wages	taxableInterest	taxWithheld	earnedIncomeCredit
>personalInformation	>personalInformation				
TaxPayer	Spouse	Wages	Taxable Interest	Tax Withheld	Earned Income Credit
He	She	32026	1450	4530	230

Both columns "TaxPayer" and "Spouse" use the reference ">personalInformation". It means that these columns may include only primary keys from the table, "personalInformation". In our example there are only two valid keys, He or She. If you enter something else, for example "John" instead of "He" and save your Excel file, you will receive a compile time (!) error "Index Key John not found" (it will be displayed in your Eclipse Problems windows). It is extremely important that the **cross-references are automatically validated at compile time** in order to prevent much more serious problems at run-time.

Multiple examples of complex inter-table relationships are provided in the sample rule project AutoInsurance. Here is an intuitive example of three related data tables:

Data Driver drivers				
name	age	gender	maritalStatus	dmvPoints
Name	**Age**	**Gender**	**Marital Status**	**DMV Points**
John Smith	24	Male	Single	2
Mary Smith	19	Female	Single	0

Data Vehicle vehicles				
id	make	model	year	hasAbs
ID	**Make**	**Model**	**Year**	**Has ABS**
Veh 1	Nissan	Maxima	2000	TRUE
Veh 2	Toyota	Corrola	1999	FALSE

Data Usage usages		
driver	vehicle	usage
> drivers	>vehicles	
Driver	**Vehicle**	**Usage(%)**
John Smith	Veh 1	100
Mary Smith	Veh 2	100

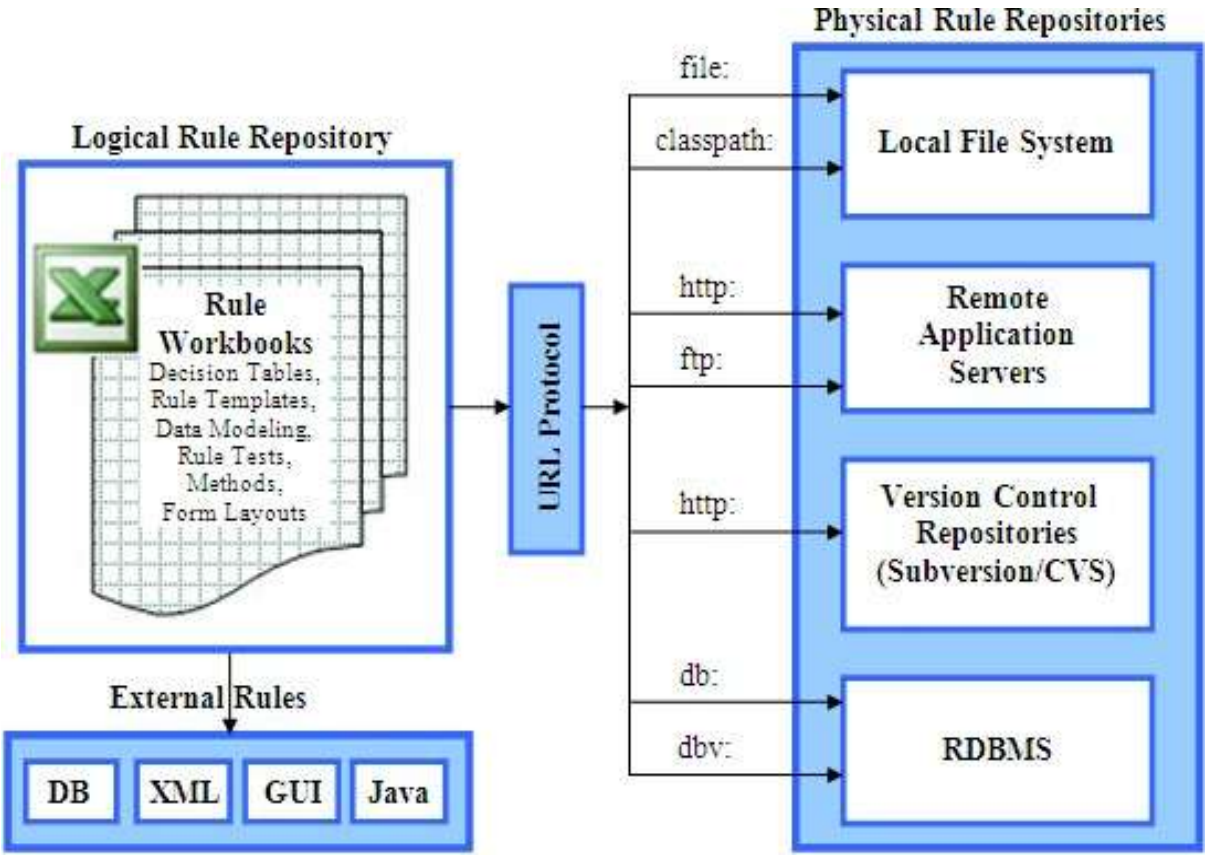
See more complex examples in the standard project "AutoInsurance".

OpenRules® Repository

To represent business rules OpenRules® utilizes a popular spreadsheet mechanism and places rules in regular Excel files. OpenRules® allows users to build enterprise-level rules repositories as hierarchies of inter-related xls-files. The OpenRules® Engine may access these rules files directly whether they are located in the local file system, on a remote server, in a standard version control system or in a relational database.

Logical and Physical Repositories

The following picture shows the logical organization of an OpenRules® repository and its possible physical implementations:



Logically, OpenRules® Repository may be considered as a hierarchy of rule workbooks. Each rule workbook is comprised of one or more worksheets that can be used to separate information by types or categories. Decision tables are the most typical OpenRules® tables and are used to represent business rules. Along with rule tables, OpenRules® supports tables of other types such as: Form Layouts, Data and Datatypes, Methods, and Environment tables. A detailed description of OpenRules® tables can be found here.

Physically, all workbooks are saved in well-established formats, namely as standard xls- or xml-files. The proper Excel files may reside in the local file system, on remote application servers, in a version control system such as Subversion, or inside a standard database management system.

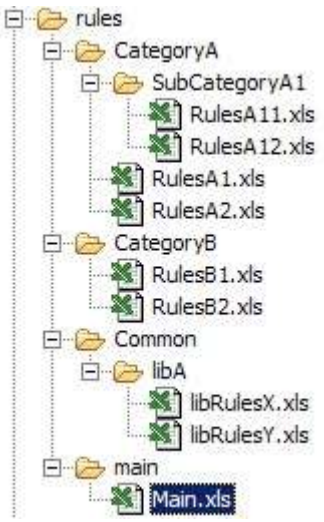
OpenRules® uses an URL pseudo-protocol notation with prefixes such as **`"file:", "classpath:", "http://", "ftp://", "db:"`**, etc.

Hierarchies of Rule Workbooks

An OpenRules® repository usually consists of multiple Excel workbooks distributed between different subdirectories. Each rule workbook may include references to other workbooks thus comprising complex hierarchies of inter-related workbooks and rule tables.

Included Workbooks

Rules workbooks refer to other workbooks using so called "`includes`" inside the OpenRules® "Environment" tables. To let OpenRules® know about such include-relationships, you have to place references to all included xls-files into the table "`Environment`". Here is an example of an OpenRules® repository that comes with the standard sample project "RuleRepository":



The main xls-file "Main.xls" is located in the local directory "rules/main". To invoke any rules associated with this file, the proper Java program creates an OpenRulesEngine using a string "file:rules/main/Main.xls" as a parameter. There are many other xls-files related to the Main.xls and located in different subdirectories of "rules". Here is a fragment of the Main.xls "Environment" table:

include	../CategoryA/RulesA1.xls
	../CategoryA/RulesA2.xls
	../CategoryB/RulesB1.xls
	../CategoryB/RulesB2.xls
	../Common/libA/libRulesX.xls
	../Common/libA/libRulesY.xls

As you can guess, in this instance all included files are defined relative to the directory "`rules/main`" in which “Main.xls” resides. You may notice that files “RulesA11.xls” and “RulesA12.xls” are not included. The reason for this is that only “RulesA1.xls” really "cares" about these files. Naturally its own table "`Environment`" contains the proper "`include`":

Environment	
import.java	myjava.packA1.*
include	SubCategoryA1/RulesA11.xls
	SubCategoryA1/RulesA12.xls

Here, both "includes" are defined relative to the directory "CategoryA" of their "parent" file "RulesA1.xls". As an alternative, you may define your included files relative to a so called "`include.path`" - see sample in the next section.

Include Path and Common Libraries of Rule Workbooks

Includes provide a convenient mechanism to create libraries of frequently used xls-files and refer to them from different rule repositories. You can keep these libraries in a file system with a fixed "`include.path`". You may even decide to move such libraries with common xls-files from your local file system to a remote server. For instance, in our example above you could move a subdirectory "libA" with all xls-files to a new location with an http address http://localhost:8080/my.common.lib. In this case, you should first define a so-called "`include.path`" and then refer to the xls-files relative to this `include.path` using angle brackets as shown below:

include.path	http://localhost:8080/my.common.lib/
include	<libA/libRulesX.xls>
	<libA/libRulesX.xls>

Here we want to summarize the following important points:

- The structure of your rule repository can be presented naturally inside xls-files themselves using "includes"
- The rule repository can include files from different physical locations
- Complex branches on the rules tree can encapsulate knowledge about their own organization.

Imports from Java

OpenRules® allows you to externalize business logic into xls-files. However, these files still can use objects and methods defined in your Java environment.

For example, in the standard example "RulesRepository" all rule tables deal with Java objects defined in the Java package myjava.package1. Therefore, the proper Environment table inside file Main.xls (see above) contains a property "import.java" with value "myjava.package1.*".

Usually, you only place common Java imports inside the main xls-file. If some included xls-files use special Java classes you can reference them directly from inside their own Environment tables.

Imports from XML

Along with Java, OpenRules® allows you to use objects defined in XML files. For example, the standard sample project "HelloXMLCustomer" uses an object of the type, Customer, defined in the file Customer.xml located in the project classpath:

```
<Customer
      name="Robinson"
      gender="Female"
      maritalStatus="Married"
      age="55"
/>
```

The xls-file "HelloCustomer.xls" that deals with this object includes the following Environment table:

Environment	
import.static	com.openrules.tools.Methods
import.schema	**classpath:/Customer.xml**
import.java	hello.Response
include	include/HelloRules.xls

The property "import.schema" specifies the location of the proper xml-file, in this case "classpath:/Customer.xml". Of course, it could be any other location in the file system that starts with the prefix "**file:**". This example also tells you that this Excel file uses:

1. static Java methods defined in the standard OpenRules® package "com.openrules.tools.Methods"
2. xml-file "classpath:/Customer.xml"
3. Java class "Response" from a package "hello"
4. include-file "HelloRules.xls" that is located in the subdirectory "include" of the directory where the main xls file is located.

Parameterized Rule Repositories

An OpenRules® repository may be parameterized in such a way that different rule workbooks may be invoked from the same repository under different circumstances. For example, let's assume that we want to define rules that offer different travel packages for different years and seasons. We may specify a concrete year and a season by using environment variables YEAR and SEASON. Our rules repository may have the following structure:

```
rules/main/Main.xls

rules/common/CommonRules.xls

rules/2007/SummerRules.xls

rules/2007/WinterRules.xls

rules/2008/SummerRules.xls

rules/2008/WinterRules.xls
```

To make the OpenRulesEngine automatically select the correct rules from such a repository, we may use the following parameterized include-statements inside the Environment table of the main xls-file rules/main/Main.xls:

Environment	
import.java	season.offers.*
include	../common/SalutationRules.xls
include	../${YEAR}/${SEASON}Rules.xls

Thus, the same rules repository will handle both WinterRules and SummerRules for different years. A detailed example is provided in the standard project SeasonRules.

Rules Version Control

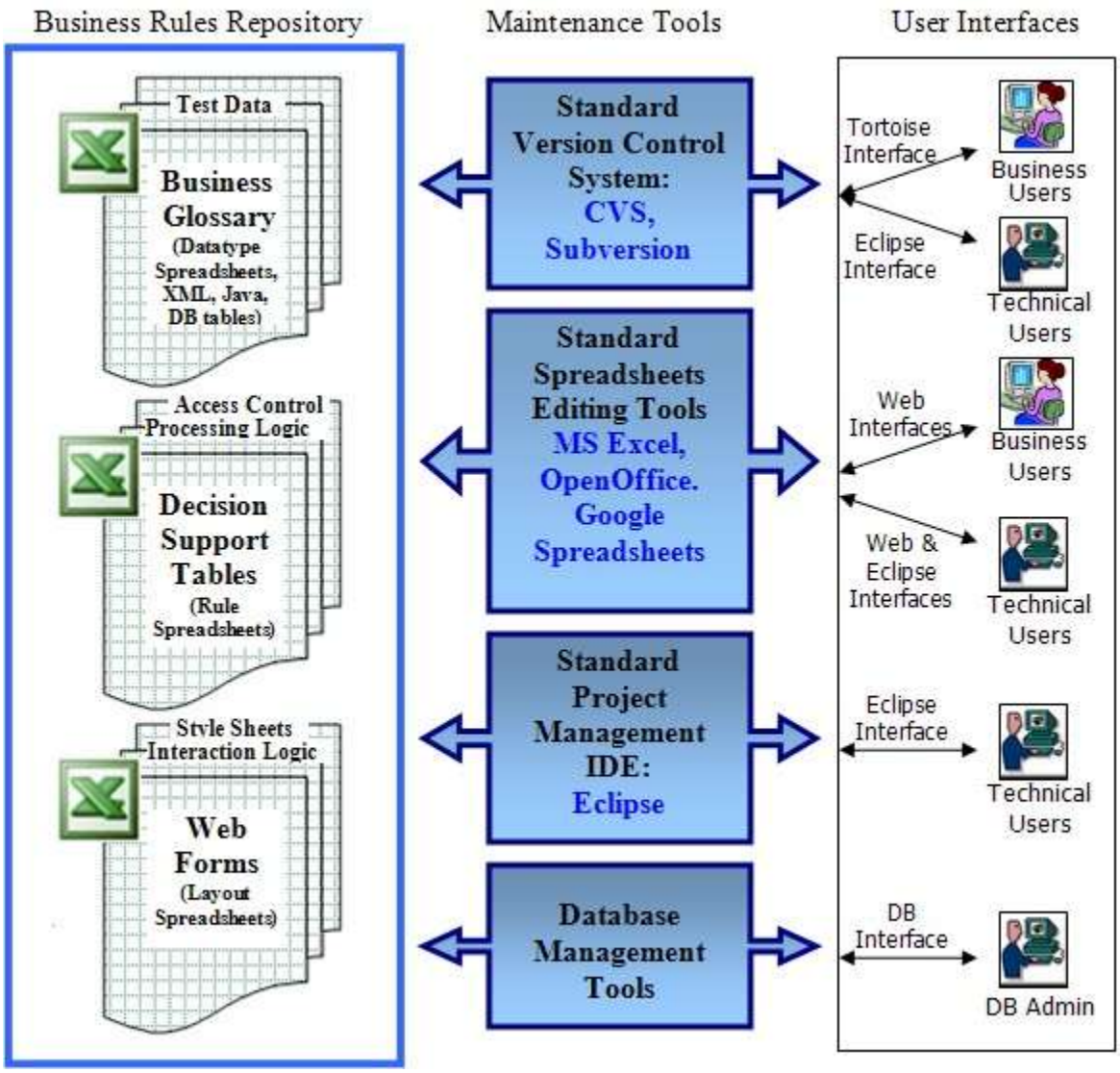
For rules version control you can choose any standard version control system that works within your traditional software development environment. We would recommend using an open source product "Subversion" that is a compelling replacement for CVS in the open source community. For business users, a friendly web interface is provided by a popular open source product TortoiseSVN. For technical users, it may be preferable to use a Subversion incorporated into Eclipse IDE. One obvious advantage of the suggested approach is the fact that both business rules and related Java/XML files will be handled by the same version control system.

You may even keep your Excel files with rules, data and other OpenRules® tables directly in Subversion. If your include-statements use http-addresses that point to a concrete Subversion repository then the OpenRulesEngine will dynamically access SVN repositories without the necessity to move Excel files back into a file system.

Another way to use version control is to place your rule workbooks in a database and use DBV-protocol to access different versions of the rules in run-time - read more.

Rules Authoring and Maintenance Tools

OpenRules® relies on standard commonly used tools (mainly from Open Source) to organize and manage a Business Rules Repository:



To create and edit rules and other tables presented in Excel-files you may use any standard spreadsheet editors such as:

- MS Excel™
- OpenOffice™
- Google Spreadsheets™

Google Spreadsheets are especially useful for sharing spreadsheet editing - see section Collaborative Rules Management with Google Spreadsheets.

For technical people responsible for rules project management OpenRules provides an Eclipse Plug-in that allows them to treat business rules as a natural part of complex Java projects.

EXTERNAL RULES

OpenRules® allows a user to create and maintain their rules outside of Excel-based rule tables. It provides a generic Java API for adding business rules from different external sources such as:

1. Database tables created and modified by the standard DB management tools
2. Live rule tables in memory dynamically modified by an external GUI
3. Java objects of the predefined type `RuleTable`
4. Problem-specific rule sources that implement a newly offered rules provider interface.

With external rules you may keep the business parts of your rules in any external source while the technical part (Java snippets) will remain in an Excel-based template, based on which actual rules will be created by the OpenRulesEngine. For example, you may keep your rules in a regular database table as long as its structure corresponds to the columns (conditions and actions) of the proper Excel template. Thus, the standard DB management tools, or your own GUI that maintains these DB-based rule tables, de-facto become your own rules management environment.

The external rules may also support a preferred distribution of responsibilities between technical and business people. The business rules can be kept and maintained in a database or other external source by business analysts while developers can continue to use Excel and Eclipse to maintain rule templates and related software interfaces.

API for External Rules

OpenRules® provides a generic Java API for external rules. There is a special constructor,

```
OpenRulesEngine(String excelFileName, ExternalRules rules)
```

that has an additional parameter of the predefined Java type `ExternalRules`. You may create an object of this type such as,

```
ExternalRules externalRules = new ExternalRules();
```

and then add different rule tables using the method:

```
addRuleTable(String ruleTableName,
             String ruleTemplateName,
             Object[][] ruleGrid);
```

The complete API is described at OpenRules® API. This simple interface gives a developer the ability to bring rules from any external source and add them to OpenRulesEngine as regular Java objects. If the rules in the external source are changed, a developer may notify the `ExternalRules` object about this change by using the method,

```
externarRules.setModified(true);
```

Then during the next rule engine's run, all rules will be dynamically reloaded.

OpenRules® provides 5 sample projects that demonstrate how to use External Rules:

ExternalRulesFromJava:	shows how to define rules as Java objects
ExternalRulesFromDB:	shows how to define rules in MS Access using JDBC
ExternalRulesFromXML:	shows how to define rules in XML files
ExternalRulesFromExcel:	shows how to define rules as Excel Data tables
ExternalRulesFromGUI:	shows how to build a web application that

allows a user to change and execute rules on the fly without a restart

These projects can be found in the complete OpenRules® installation under the section "External Rules". External rules can be invoked from regular rules described in Excel files. Because these external rules are not known until run-time, OpenRules® will produce warnings about these as yet unknown rules, but the OpenRulesEngine will still execute them without problems. To suppress the warnings and to keep track of all participating rules, you may fill out a newly introduced optional table of the type "ExternalRules" that lists names of all external rules along with their templates as in the following example:

ExternalRules	
greetingRules	defineGreeting
salutationRules	defineSalutation

The projects below will produce greetings like "Good Morning, Mrs. Robinson!" based on the current time and different customer's attributes such as gender and marital status. They are similar to the standard project "HelloJavaTemplates" but instead of using Excel-based rule tables they will use external rules.

The business logic for producing greetings and salutations is presented in the Excel file HelloTemplates.xls. The first template

Rules void **defineGreeting**(App app, int hour)		
C1	C2	A1
min <= hour	hour <= max	app.greeting = greeting;
int min	int max	String greeting
Hour From	**Hour To**	**Set Greeting**
		Unknown Greeting

specifies how to define different greetings (Good Morning, Good Afternoon, etc.) based on the hour of the day. If the parameter "hour" belongs to the interval [min;max] defined by a concrete rule, then the attribute "greeting", of the parameter "app" will be set to the proper greeting. If no rules are satisfied, this multi-hit table will use the default greeting "Unknown Greeting".

The second template

Rules void **defineSalutation**(App app, Customer c)			
C1	C2	C3	A1
c.gender.equals (gender)	c.maritalStatus. equals(status)	c.age < age	app.salutation = salutation;
String gender	String status	int age	String salutation
Gender	**Marital Status**	**Age Less Than**	**Set Salutation**
			Unknown Salutation

specifies how to define different salutations (Mr., Mrs., etc.) based on customer attributes Gender, Marital Status, and Age. If no rules are satisfied, this multi-hit table will use the default salutation "Unknown Salutation".

External Rules from Java

The OpenRulesEngine can be created with an additional parameter of the predefined type ExternalRules that allows for rule tables defined as Java objects. The project "ExternalRulesFromDJava" demonstrates different ways of defining external rules in Java.

Step 1. Defining Rule Tables in Java

All Java classes are typical for basic rule projects. In this project the main Java class `RunExternalRules` shows different ways for adding rule tables to the external rules. Here is the first rule table:

```
externalRules.addRuleTable(
     "ExternalSummerGreeting", //table name
     "defineGreeting",         //template name
     new String[][] {          //rules
             new String[] {"0","11","Good Morning Summer"},
             new String[] {"12","17","Good Afternoon Summer"},
             new String[] {"18","21","Good Evening Summer"},
             new String[] {"22","24","Good Night Summer"}
     }
);
```

The first parameter specifies the rule table name. The second parameter specifies the template upon which this table will be based. The third parameter defines a grid that is a two-dimensional array, `Object[][]`, containing actual rules. This grid corresponds to the template "defineGreeting" - see above. The first rule in the grid states that IF Hour From is "0" AND Hour To is "11" THEN Set Greeting as "Good Morning Summer", etc.

The second rule table,

```
externalRules.addRuleTable(
     "ExternalWinterGreeting", //table name
     "defineGreeting",         //template name
     new String[][] {          //rules
             new String[] {"0","12","Good Morning Winter"},
             new String[] {"13","16","Good Afternoon Winter"},
             new String[] {"17","22","Good Evening Winter"},
             new String[] {"23","24","Good Night Winter"}
     }
);
```

is very similar to the first one but defines greeting rules for a winter season.

The third rule table,

```
externalRules.addRuleTable(
     "ExternalGreetingHorizontal", //table name
     "defineGreetingHorizontal",   //template name
     new String[][] {              //rules
             new String[] {"0","11","Good Morning"},
             new String[] {"12","16","Good Afternoon"},
             new String[] {"17","22","Good Evening"},
             new String[] {"23","24","Good Night"}
     }
);
```

shows that you may use a horizontal template "ExternalGreetingHorizontal" and still use the same vertical structure of your rules.

The fourth rule table,

```
externalRules.addRuleTable(
  "ExternalGreetingReverseOrder", //table name
  "defineGreeting",                  //template name
  new String[] { "A1","C1" }, //labels order differs from template order
  new Object[][] { // not all cells contains strings but other objects
    new Object[] {"Good Morning", new Integer(0),new Integer(11) },
    new Object[] {"Good Afternoon",new Integer(12),new Integer(17)},
    new Object[] {"Good Evening",new Integer(18),new Integer(21)},
    new Object[] {"Good Night",new Integer(22),new Integer(24)}
  }
};
```

shows several additional options that could be added to the `ExternalRules` object. First of all, you can use all optional rules and conditions along with other features available for "normal" rules and templates - as described above. The array of Strings,

```
new String[] { "A1","C1" }
```

placed just before the grid informs OpenRulesEngine that this rule table starts with the action A1 followed by the condition C1, thus violating the default column order in the template. The grid `Object[][]` demonstrates the ability to specify not only String but any Java objects as long as they correspond to the types of parameters specified in the template.

If the type of objects inside the rule tables do not correspond to the templates, the proper error will be produced. For example, if you make a mistake in the first rule table by typing “O” instead of “0”

```
new String[] {"O","11","Good Morning Summer"}
```

you will receive a compilation error that will look like this:

```
org.openl.syntax.SyntaxErrorException: Error: For input string:
"O": java.lang.NumberFormatException
```

*at ExternalRules#**ExternalSummerGreeting**?**row=0**&**column=0**&openl=*
java.lang.NumberFormatException: For input string: "O"

The error message points you to the name of the invalid external table (*ExternalRules#ExternalSummerGreeting*) and to the coordinates of the invalid cells inside the grid (*row=0&column=0*).

Step 2. Executing External Rules from a Java Program

The main file HelloCustomer.xls defines the Environment of our rule project as follows:

Environment	
import.java	hello.*
include	../include/HelloTemplates.xls

This application uses two simple Java beans: "Request" with one integer attribute "hour" and "Response" with one String attribute "result".

The main Java class `RunExternalRules` creates and executes an OpenRulesEngine in the standard way:

```
String fileName = "file:rules/main/HelloCustomer.xls";
OpenRulesEngine engine =
          new OpenRulesEngine(fileName,externalRules);
Response response = new Response();
Request request = new Request();
request.setHour(Calendar.getInstance().get(Calendar.HOUR_OF_DAY));
Object[] params = new Object[] { request, response };

for (int i = 0; i < externalRules.getRuleTables().size(); i++) {
  RuleTable rules = (RuleTable)externalRules.getRuleTables().get(i);
  System.out.println(rules);
  engine.run(rules.getTableName(), params);
  System.out.println("Greeting generated by rules '" +
            rules.getTableName() +
            "' for hour " +request.hour +": " + response.result );
  System.out.println();
}
```

To run the project, you may double-click on the file "run.bat". Here is an expected output:

```
INITIALIZE OPENRULES ENGINE 5.3.0 (build 03092009) for
[file:rules/main/HelloCustomer.xls]
External rules table: ExternalSummerGreeting
External rules table: ExternalWinterGreeting
External rules table: ExternalGreetingHorizontal
External rules table: ExternalGreetingReverseOrder
IMPORT.JAVA=hello.*
INCLUDE=../include/HelloTemplates.xls
[../include/HelloTemplates.xls] has been resolved to
[file:<...>/rules/include/HelloTemplates.xls]

ExternalRules ExternalSummerGreeting template defineGreeting
0 11 Good Morning Summer
12 17 Good Afternoon Summer
18 21 Good Evening Summer
22 24 Good Night Summer

Greeting generated by rules 'ExternalSummerGreeting' for hour 16: Good
Afternoon Summer

ExternalRules ExternalWinterGreeting template defineGreeting
0 12 Good Morning Winter
13 16 Good Afternoon Winter
17 22 Good Evening Winter
23 24 Good Night

Greeting generated by rules 'ExternalWinterGreeting' for hour 16: Good
Afternoon Winter

ExternalRules ExternalGreetingHorizontal template
defineGreetingHorizontal
0 11 Good Morning
12 16 Good Afternoon
17 22 Good Evening
23 24 Good Night

Greeting generated by rules 'ExternalGreetingHorizontal' for hour 16:
Good Afternoon

ExternalRules ExternalGreetingReverseOrder template defineGreeting
Good Morning 0 11
Good Afternoon 12 17
Good Evening 18 21
Good Night 22 24

Greeting generated by rules 'ExternalGreetingReverseOrder' for hour 16:
Good Afternoon
```

External Rules from Database

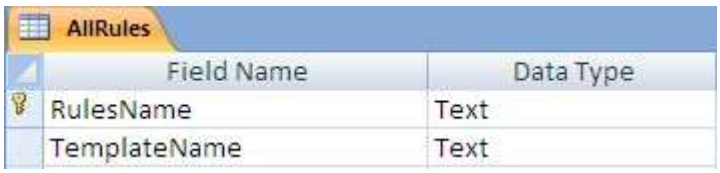
OpenRules® allows you to keep your business rules in regular database tables whose structures correspond to the columns (conditions and actions) of Excel's templates based on which of the proper rule tables will be executed. This way

the standard DB management tools can be used as your own rules management environments.

The project "ExternalRulesFromDB", demonstrates how to define rules in a MS Access database with regular tables (without Excel files saved as blobs). Because we are using a standard JDBC interface, this project should work similarly with other database management systems.

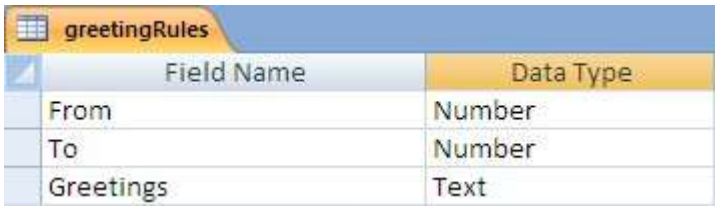
Step 1. Setting Up Database with Rule Tables

Use MS Access to create a new database, labeled "OpenRulesDB", and save it in the subdirectory "DB" of the directory ExternalRulesFromDB. Using MS Access, create the table "AllRules", which looks like this one:



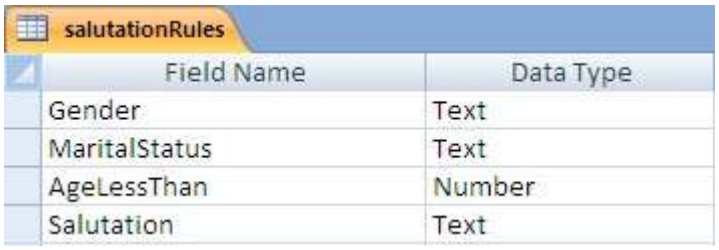
RulesName	TemplateName
greetingRules	defineGreeting
salutationRules	defineSalutation

This DB table has only two text columns "RulesName' and "TemplateName". Now we have to create a simple DB table, "greetingRules", with a structure that corresponds to our template "defineGreeting":



From	To	Greetings
0	11	Good Morning
12	15	Good Afternoon
16	21	Good Evening
22	24	Good Night

Similarly, we will create a table, "salutationRules", that corresponds to our template "defineSalutation":



Gender	MaritalStatus	AgeLessThan	Salutation
Male			Mr.
Male	Single	3	Little
Female	Single		Ms.
Female	Married		Mrs.

To make this database accessible from a Java program we have to setup the proper data source. In Windows, we have to go to Control Panel, open Administrative Tools, and select Data Sources (ODBC). Add a new User Data Source with the following information:

Data Source Name: OpenRulesDB

Description: HelloExternalRulesFromDB

Click on the "Select" button and chose your just created OpenRulesDB.mdb file.

The above DB structure serves only as an example. You may organize your database with rule tables differently with additional information about rule tables including such attributes as "CreatedBy", "CreatedAt", "LastModifiedBy", "LastModifiedAt", "Category", and many more attributes that accommodate your particular needs.

Step 2. Defining a DB interface in Java

To inform an OpenRulesEngine about external rules, you need to create an object of the type ExternalRules and add to it all external RuleTables. Each instance of the class RuleTable consists of:

- rule table name (String)
- template name (String)
- a grid of objects that represent the content of a rule tables (Object[][])

In our case, to create an instance of the class External Rules we should:

1) read all rows from the DB table "AllRules"
2) for every pair (RuleName;TemplateName) find the proper DB table and create the required grid of the type Object[][] for all rows and all columns of the DB table.

To accomplish this, we have created the class OpenRulesDB.java that contains the method `"readRuleTables()"`. This class is inherited from the standard JDBC interface supported by the class DbUtil included in the OpenRules® installation within the project `"com.openrules.tools"`. Here is the code of this method with comments:

```
public synchronized ExternalRules readRuleTables() {

     String mainTable = "AllRules";
     String columnWithRuleNames = "RulesName";
     String columnWithTemplateNames = "TemplateName";

     ExternalRules externalRules = new ExternalRules();

     try {
       String mainSQL = "SELECT * FROM " + mainTable;
       ResultSet mainRS = executeQuery(mainSQL);
       // FOR ALL RULE TABLES
       while(mainRS.next()) {
              String ruleTableName = mainRS.getString(columnWithRuleNames);
              String ruleTemplateName =
                     mainRS.getString(columnWithTemplateNames);
              System.out.println("Rules " + ruleTableName + " template "
                           + ruleTemplateName);
              try {
                int numberOfRows = count(ruleTableName);
                System.out.println("Total " + numberOfRows + " rows");
                String insideSQL = "SELECT * FROM " + ruleTableName;
                ResultSet rs = executeQuery(insideSQL);
                ResultSetMetaData md = rs.getMetaData();
                int numberOfColumns = md.getColumnCount();
```

```
                System.out.println("Total " + numberOfColumns + " columns");
                Object[][] grid = new Object[numberOfRows][numberOfColumns];
                int rowIndex = 0;
                // FOR ALL TABLE's ROWS
                while (rs.next()) {
                    Object[] row = new Object[numberOfColumns];
                    // FOR ALL TABLE's COLUMNS
                    for (int i=0; i< numberOfColumns; i++) {
                        // Add grid element
                        row[i] = rs.getObject(i+1);
                        if (row[i] == null)
                           row[i] = "";
                    }
                    grid[rowIndex++] = row;
                }
                rs.close();
                // ADD RULE TABLE
                externalRules.addRuleTable(ruleTableName,
                                           ruleTemplateName,grid);
            }
            catch(Exception e) {
                System.err.println("ERROR in the DB table " +
                          ruleTableName + "\n" + e.toString());
            }
        }
        mainRS.close();
      }
      catch(Exception e) {
         System.err.println("ERROR in the DB table " +
               mainTable + "\n" + e.toString());
               }
         return externalRules;
      }
}
```

Step 3. Creating and Executing Rules from a Java Program

All other modules are typical for basic rule projects. The main Java file, RunExternalRulesFromDB.java, is used to test the above rules:

```
import com.openrules.ruleengine.ExternalRules;
import com.openrules.ruleengine.OpenRulesEngine;

public class RunExternalRulesFromDB {

  public static void main(String[] args) {
      // Read DB to create ExternalRules
      OpenRulesDB db = new OpenRulesDB();
      ExternalRules externalRules = db.readRuleTables();
      // Create OpenRulesEngine with external rules
      String fileName = "file:rules/main/HelloCustomer.xls";
      OpenRulesEngine engine =
                new OpenRulesEngine(fileName,externalRules);
      // Print external rules
      for (int i = 0; i < externalRules.getRuleTables().size(); i++)

      System.out.println(externalRules.getRuleTables().get(i));
```

```
        // Create a test App with a test customer from HelloData.xls
        App app = (App) engine.run("getDefaultApplication");
        // Run OpenRulesEngine
        engine.run("generateGreeting",app);
        System.out.println("\nGenerated Greeting: " + app.getResult());
    }
}
```

Here we create an instance of OpenRulesEngine using the main Excel-file, HelloCustomer.xls, and external rules received from the DB. The main file HelloCustomer.xls defines the Environment as follows:

Environment	
import.java	hello.*
import.static	com.openrules.tools.Methods
include	../include/HelloTemplates.xls
include	../include/HelloData.xls

This application uses two simple Java beans:

```
Customer.java:
        String name;
        String maritalStatus;
        String gender;
        int age;
App.java:
        Customer customer;
        String greeting;
        String salutation;
        String result;
```

The proper instance of Customer and App are created based on the Excel file, HelloData.xls, using these data tables:

Data App apps

customer.name	customer.maritalStatus	customer.gender	customer.age
Customer Name	**Marital Status**	**Gender**	**Age**
Robinson	Married	Female	24
Smith	Single	Male	19

Method App getDefaultApplication()
return apps[0];

And finally, the engine will execute rules by calling the method "run":

```
engine.run("generateGreeting",app);
```

The proper method, "generateGreeting", is described in the file, HelloCustomer.xl,s in the following table:

Method void generateGreeting(App app)

int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
greetingRules(app, hour);
salutationRules(app, app.customer);
app.result = app.greeting + ", " + app.salutation + " " + app.customer.name + "!";

You may validate the entire rule project by double-clicking on the file "compile.bat". Because the actual external rule tables, "greetingRules" and "salutationRules", will become known only in run-time the proper OpenRules® validator may produce errors (warnings) about unknown rule tables. You may ignore these errors or you may explicitly inform OpenRules® about this fact by adding an optional table to the file HelloCustomer.xls:

ExternalRules	
greetingRules	defineGreeting

salutationRules	defineSalutation

To run the project you may double-click on the file "run.bat". Here is an expected output:

```
ExternalRules greetingRules template defineGreeting
0 11 Good Morning
12 15 Good Afternoon
16 21 Good Evening
22 24 Good Night

ExternalRules salutationRules template defineSalutation
Male   Mr.
Male Single 3 Little
Female Single  Ms.
Female Married  Mrs.

Generated Greeting: Good Morning, Mrs. Robinson!
```

External Rules from XML

OpenRules® allows you to keep your business rules in XML files which correspond to the columns (conditions and actions) of Excel's templates based upon which the proper rule tables will be executed. While you may use any XML processing software, this sample project demonstrates how to use a simple XML interface provided by OpenRules®.

Step 1. Defining Rule Tables in XML

You may create a subdirectory "xml" in the directory "rules" and place different xml-files into it. The first file, "GreetingRules.xml", defines a rule table with the name "greetingRules" that will be based on the template with the name "defineGreeting":

```
<?xml version="1.0" encoding="UTF-8"?>

<GreetingRules ruleTableName="greetingRules"
templateName="defineGreeting" type="Array of Rule(s)">

  <Rule from="0" to="11" greeting="Good Morning" />
  <Rule from="12" to="16" greeting="Good Afternoon" />
  <Rule from="17" to="21" greeting="Good Evening" />
  <Rule from="22" to="24" greeting="Good Night" />
</GreetingRules>
```

Similarly, we create the second file, "SalutationRules.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<SalutationRules ruleTableName="salutationRules"
templateName="defineSalutation">
   <Rule
       gender="Female"
       maritalStatus="Married"
       salutation="Mrs."
   />
   <Rule
       gender="Female"
       maritalStatus="Single"
       salutation="Ms."
   />
   <Rule
      gender="Male"
      maritalStatus=""
      salutation="Mr."
   />
   <Rule
      gender="Male"
      maritalStatus="Single"
      maxAge="5"
      salutation="Little"
   />
</SalutationRules>
```

Please note that the last rule contains an extra attribute, "maxAge". OpenRules® does not require any specification of the XML document and will dynamically recognize its structure.

Step 2. Reading XML rules in Java

To inform an OpenRulesEngine about external rules, you need to create an object of the type ExternalRules and add to it all external RuleTables. Each instance of the class RuleTable consists of:

- rule table name (String)
- template name (String)
- a grid of objects that represent the contents of a rule table (Object[][])

In this project, we will create an instance of the class External Rules directly in the Excel method "getExternalRules":

Method ExternalRules **createExternalRules**()

ExternalRules externalRules = new ExternalRules();
addGreetingRules(externalRules);
addSalutationRules(externalRules);
return externalRules;

This method will execute two other methods, "addGreetingRules" and "addSalutationRules", that will read the above xml-files and will add the proper rule tables to the newly created ExternalRules object.

Before reading the xml files, we have to specify the proper xml schemas in the Environment table placed in the main Excel file HelloXMLRules.xls:

Environment	
import.java	com.openrules.table.external.Objects
import.schema	file:rules/xml/GreetingRules.xml
	file:rules/xml/SalutationRules.xml

OpenRules® dynamically defines the Java classes, GreetingRules and SalutationRules, that will be used to read the proper XML files.

Now we may specify the method "addGreetingRules":

Method ExternalRules addGreetingRules(ExternalRules externalRules)

```
GreetingRules greetings = GreetingRules.load("file:rules/xml/GreetingRules.xml");
Objects[] grid = new Objects[greetings.Rule.length];
for(int i = 0; i < greetings.Rule.length; ++i) {
   GreetingRules.Rule r = greetings.Rule[i];
   Objects row = new Objects(3);
   row.set(0,r.from); row.set(1,r.to); row.set(2,r.greeting);
   grid[i] = row;
}
externalRules.addRuleTable(greetings.ruleTableName, greetings.templateName, grid );
```

First we load the rules from the xml-file defining its relative path using the standard OpenRules® URL notation:

```
file:rules/xml/GreetingRules.xml
```

All objects specified in the file GreetingRules.xml becomes available to the Java code through the object "greetings" of the dynamically defined type GreetingRules. In particular, the object "greetings.Rule" points to the array of objects of the dynamic type "Rule" as it was defined in the xml-file.

Next, we create a "grid" as an array of the predefined type Objects, which is used by OpenRules® to simplify the handling of the multi-dimensional array. Looping through all elements of the array greetings.Rules, we add new rows to the object "grid". Data elements inside each rule are available through their names as defined in the xml-file: `r.from`, `r.to`, and `r.greeting`.

Similarly, we specify the method "addSalutationRules":

Method ExternalRules addSalutationRules(ExternalRules externalRules)

```
SalutationRules salutations = SalutationRules.load("file:rules/xml/SalutationRules.xml");
Objects[] grid = new Objects[salutations.Rule.length];
for(int i = 0; i < salutations.Rule.length; ++i) {
   SalutationRules.Rule r = salutations.Rule[i];
   Objects row = new Objects(4);
   row.set(0,r.gender); row.set(1,r.maritalStatus); row.set(2,r.maxAge); row.set(3,r.salutation);
   grid[i] = row;
}
externalRules.addRuleTable(salutations.ruleTableName, salutations.templateName, grid );
```

Step 3. Creating and Executing Rules from a Java Program

All other modules are typical for basic rule projects. The main Java file, RunExternalRulesFromXML.java, is used to test the above rules:

```
import com.openrules.ruleengine.ExternalRules;
import com.openrules.ruleengine.OpenRulesEngine;

public class RunExternalRulesFromXML {

     public static void main(String[] args){
       // The first engine reads XML-based rules described at HelloXMLRules.xls
       String xlsMainData = "file:rules/main/HelloXMLRules.xls";
       OpenRulesEngine engine1 = new OpenRulesEngine(xlsMainData);
       ExternalRules externalRules =
              (ExternalRules) engine1.run("createExternalRules");
       // Print External Rules
       for (int i = 0; i < externalRules.getRuleTables().size(); i++)
         System.out.println(externalRules.getRuleTables().get(i));
       // The second engine reads test data and execute external rules
       // created by the first engine
       String fileName = "file:rules/main/HelloCustomer.xls";
       OpenRulesEngine engine2 =
                    new OpenRulesEngine(fileName,externalRules);
       App app = (App) engine2.run("getDefaultApplication");
       engine2.run("generateGreeting",app);
       System.out.println("\nGenerated Greeting:");
       System.out.println(app.getResult());

     }
}
```

The first instance, “engine1", of the class OpenRulesEngine is based on the main Excel-file, HelloXMLRules.xls. We execute the method, "createExternalRules", to create external rules from the xml files. The second instance "engine2" of the OpenRulesEngine uses the main Excel-file, HelloCustomer.xls, and the newly created external rules.

The main file, HelloCustomer.xls, defines the Environment as follows:

Environment	
import.java	hello.*
import.static	com.openrules.tools.Methods
include	../include/HelloTemplates.xls

include	../include/HelloData.xls

This application uses two simple Java beans:

```
Customer.java:
      String name;
      String maritalStatus;
      String gender;
      int age;
App.java:
      Customer customer;
      String greeting;
      String salutation;
      String result;
```

The proper instance of Customer and App are created based on the Excel file, HelloData.xls, using these data tables:

Data App apps			
customer.name	customer.maritalStatus	customer.gender	customer.age
Customer Name	**Marital Status**	**Gender**	**Age**
Robinson	Married	Female	24
Smith	Single	Male	19

Method App getDefaultApplication()
return apps[0];

And finally, `engine2` will execute the rules by calling the method "run":

```
engine2.run("generateGreeting",app);
```

The proper method, "generateGreeting", is described in the file, HelloCustomer.xls. in the following table:

Method void generateGreeting(App app)

int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
greetingRules(app, hour);
salutationRules(app, app.customer);
app.result = app.greeting + ", " + app.salutation + " " + app.customer.name + "!";

You may validate the entire rule project by double-clicking on the file "compile.bat". Because the actual external rule tables, "greetingRules" and "salutationRules", will become known only at run-time the proper OpenRules® validator may produce errors (warnings) about unknown rule tables. You may ignore these errors or you may explicitly inform OpenRules® about this fact by adding an optional table to the file, HelloCustomer.xls:

ExternalRules	
greetingRules	defineGreeting
salutationRules	defineSalutation

To run the project you may double-click on the file "run.bat". Here is an expected output:

```
INITIALIZE OPENRULES ENGINE 5.3.0 (build 03092009) for
[file:rules/main/HelloXMLRules.xls]
IMPORT.JAVA=com.openrules.table.external.Objects
IMPORT.SCHEMA=file:rules/xml/GreetingRules.xml
IMPORT.SCHEMA=file:rules/xml/SalutationRules.xml
ExternalRules greetingRules template defineGreeting
0 11 Good Morning
12 16 Good Afternoon
17 21 Good Evening
22 24 Good Night

ExternalRules salutationRules template defineSalutation
Female Married null Mrs.
Female Single null Ms.
Male  null Mr.
Male Single 5 Little

INITIALIZE OPENRULES ENGINE 5.3.0 (build 03092009) for
[file:rules/main/HelloCustomer.xls]
External rules table: greetingRules
External rules table: salutationRules
```

```
IMPORT.JAVA=hello.*
IMPORT.JAVA=com.openrules.tools.Operator
IMPORT.STATIC=com.openrules.tools.Methods
INCLUDE=../include/HelloTemplates.xls
[../include/HelloTemplates.xls] has been resolved to
[file:<..>/ExternalRulesFromXML/rules/include/HelloTemplates.xls]
INCLUDE=../include/HelloData.xls
[../include/HelloData.xls] has been resolved to
[file:<..>/ExternalRulesFromXML/rules/include/HelloData.xls]

Generated Greeting:
Good Afternoon, Mrs. Robinson!
```

External Rules from Excel

OpenRules® allows you to keep your business rules in Excel data tables that correspond to the columns (conditions and actions) of Excel's templates based upon which the proper rule tables will be executed.

Step 1. Defining Rule Tables in Excel Data Tables

We will create the main xls-file HelloRules.xls in the subdirectory "rules/mainl". The first Data table defines "greetingRules" which will be based on the template with the name "defineGreeting":

Data GreetingRule greetingRules		
from	to	greeting
From	**To**	**Greeting**
0	11	Good Morning
12	17	Good Afternoon
18	22	Good Evening
23	24	Good Night

To access this table from java we define the following method:

Method GreetingRule[] getDefaultGreetingRules()
return greetingRules;

This data table uses the datatype, GreetingRules, which is specified in the proper Java class:

```
public class GreetingRule {
  int from;
  int to;
  String greeting;

  public int getFrom() {
          return from;
  }
  public void setFrom(int from) {
          this.from = from;
  }
  public int getTo() {
          return to;
  }
  public void setTo(int to) {
          this.to = to;
  }
  public String getGreeting() {
          return greeting;
  }
  public void setGreeting(String greeting) {
          this.greeting = greeting;
  }
}
```

Similarly, we create the second Data table "salutationRules":

Data SalutationRule salutationRules			
gender	maritalStatus	maxAge	salutation
Gender	**Marital Status**	**Age Less Than**	**Set Salutation**
Female	Married		Mrs.
Female	Single		Ms.
Male			Mr.
Male	Single	10	Little

and the proper method:

Method SalutationRule[] getDefaultSalutationRules()
return salutationRules;

This data table uses the datatype, SalutationRules, which is specified in the proper Java class:

```
public class SalutationRule {

     String  gender;
     String  maritalStatus;
     String  maxAge;
     String  salutation;

     public String getGender() {
            return gender;
     }
     public void setGender(String gender) {
            this.gender = gender;
     }

     public String getMaritalStatus() {
            return maritalStatus;
     }
     public void setMaritalStatus(String maritalStatus) {
            this.maritalStatus = maritalStatus;
     }

     public String getMaxAge() {
            return maxAge;
     }
     public void setMaxAge(String maxAge) {
            this.maxAge = maxAge;
     }

     public String getSalutation() {
            return salutation;
     }
     public void setSalutation(String salutation) {
            this.salutation = salutation;
     }

}
```

Step 2. Creating and Executing External Rules from a Java Program

All other modules are typical for basic rule projects. The main Java file RunExternalRulesFromXML.java is used to test the above rules:

```
import com.openrules.ruleengine.ExternalRules;
import com.openrules.ruleengine.OpenRulesEngine;

public class RunExternalRulesFromExcel {

  public static void main(String[] args)   {
     // The first engine
```

```
    String xlsMainRules = "file:rules/main/HelloRules.xls";
    OpenRulesEngine engine1 = new OpenRulesEngine(xlsMainRules);
    GreetingRule[] greetingRules =
           (GreetingRule[])engine1.run("getDefaultGreetingRules");
    String[][] greetingGrid = new String[greetingRules.length][3];
    for (int i = 0; i < greetingRules.length; i++) {
      GreetingRule rule = greetingRules[i];
      greetingGrid[i] = new String[] {
        Integer.toString(rule.from),
        Integer.toString(rule.to),
        rule.greeting
       };
    }

    SalutationRule[] salutationRules =
       (SalutationRule[])engine1.run("getDefaultSalutationRules");
    String[][] salutationGrid =
                    new String[salutationRules.length][4];
    for (int i = 0; i < salutationRules.length; i++) {
      SalutationRule rule = salutationRules[i];
      salutationGrid[i] = new String[] {
           rule.gender,
           rule.maritalStatus,
           rule.maxAge,
           rule.salutation
       };
    }
    // create external rules
    ExternalRules externalRules = new ExternalRules();
    externalRules.addRuleTable(
           "greetingRules",       //table name
           "defineGreeting",      //template name
           greetingGrid
    );
    externalRules.addRuleTable(
           "salutationRules",     //table name
           "defineSalutation", //template name
           salutationGrid
    );
    // Display external rules
    for (int i = 0; i < externalRules.getRuleTables().size(); i++)
      System.out.println(externalRules.getRuleTables().get(i));

    // The second engine
    String fileName = "file:rules/main/HelloCustomer.xls";
    OpenRulesEngine engine2 =
                  new OpenRulesEngine(fileName,externalRules);
    App app = (App) engine2.run("getDefaultApplication");
    engine2.run("generateGreeting",app);
    System.out.println("\nGenerated Greeting:");
    System.out.println(app.getResult());

  }
}
```

The first instance, “engine1", of the class OpenRulesEngine, is based on the main Excel-file, HelloRules.xls. We create the array, greetingRules, by executing the method, "createExternalRules", to generate external rules from the xml files:

```
GreetingRule[] greetingRules =

      (GreetingRule[])engine1.run("getDefaultGreetingRules");
```

Then we convert this array into a simple "greetingGrid" of the type `String[][]`. Similarly, we create the grid, "salutationRules".

Next, we create an instance of ExternalRules and add two rule tables into it:

```
ExternalRules externalRules = new ExternalRules();
externalRules.addRuleTable(
   "greetingRules", //table name
   "defineGreeting", //template name
   greetingGrid
);
externalRules.addRuleTable(
   "salutationRules", //table name
   "defineSalutation", //template name
   salutationGrid
);
```

The second instance, "engine2," of the OpenRulesEngine uses the main Excel-file, HelloCustomer.xl,s and the newly created external rules:

```
OpenRulesEngine engine2 =

      new OpenRulesEngine(fileName,externalRules);
```

The main file, HelloCustomer.xls, defines the Environment as follows:

Environment	
import.java	hello.*
import.static	com.openrules.tools.Methods

include	../include/HelloTemplates.xls
include	../include/HelloData.xls

This application uses two simple Java beans:

```
Customer.java:
      String name;
      String maritalStatus;
      String gender;
      int age;
App.java:
      Customer customer;
      String greeting;
      String salutation;
      String result;
```

The proper instances of Customer and App are created based on the Excel file, HelloData.xls, using these data tables:

Data App apps			
customer.name	customer.maritalStatus	customer.gender	customer.age
Customer Name	**Marital Status**	**Gender**	**Age**
Robinson	Married	Female	24
Smith	Single	Male	19

Method App getDefaultApplication()
return apps[0];

And finally, `engine2` will execute rules by calling the method "run":

```
engine2.run("generateGreeting",app);
```

The proper method, "generateGreeting", is described in the file, HelloCustomer.xls, in the following table:

Method void generateGreeting(App app)

int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
greetingRules(app, hour);
salutationRules(app, app.customer);
app.result = app.greeting + ", " + app.salutation + " " + app.customer.name + "!";

You may validate the entire rule project by double-clicking on the file "compile.bat". Because the actual external rule tables, "greetingRules" and "salutationRules", will become known only at run-time the proper OpenRules® Validator may produce errors (warnings) about unknown rule tables. You may ignore these errors or you may explicitly inform OpenRules® about this fact by adding an optional table to the file HelloCustomer.xls:

ExternalRules	
greetingRules	defineGreeting
salutationRules	defineSalutation

To run the project you may double-click on the file "run.bat". Here is an expected output:

```
INITIALIZE OPENRULES ENGINE 5.3.0 (build 03092009) for
[file:rules/main/HelloRules.xls]
IMPORT.JAVA=hello.*
ExternalRules greetingRules template defineGreeting
0 11 Good Morning
12 17 Good Afternoon
18 22 Good Evening
23 24 Good Night

ExternalRules salutationRules template defineSalutation
Female Married null Mrs.
Female Single null Ms.
Male null null Mr.
Male Single 10 Little
```

```
INITIALIZE OPENRULES ENGINE 5.3.0 (build 03092009) for
[file:rules/main/HelloCustomer.xls]
External rules table: greetingRules
External rules table: salutationRules
IMPORT.JAVA=hello.*
IMPORT.JAVA=com.openrules.tools.Operator
IMPORT.STATIC=com.openrules.tools.Methods
INCLUDE=../include/HelloTemplates.xls
[../include/HelloTemplates.xls] has been resolved to
[file:<..>/ExternalRulesFromExcel/rules/include/HelloTemplates.xls]
INCLUDE=../include/HelloData.xls
[../include/HelloData.xls] has been resolved to
[file:<..>/ExternalRulesFromExcel/rules/include/HelloData.xls]

Generated Greeting:
Good Afternoon, Mrs. Robinson!
```

External Rules from GUI

OpenRules® allows you to keep your business rules in Excel data tables that correspond to the columns (conditions and actions) of Excel's templates based on which the proper rule tables will be executed.

Step 1. Defining A Graphical User Interface

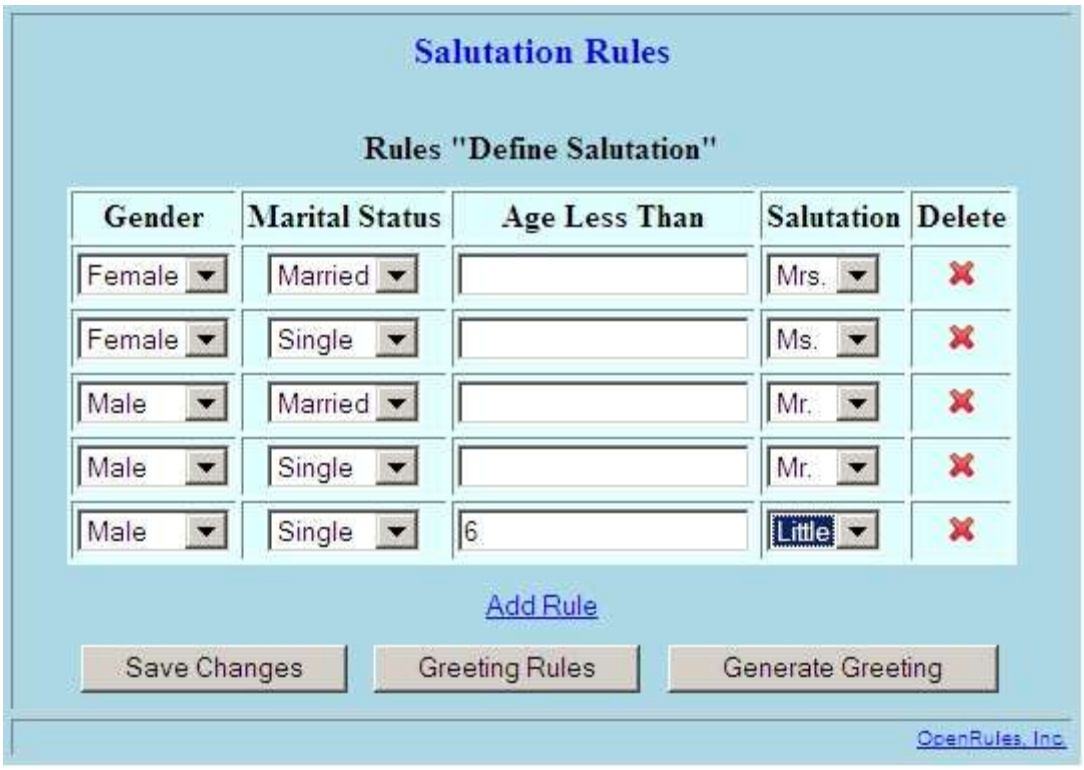
This project illustrates how to create a web application that will consist of two parts:

1) Data input and Rule Engine Execution
2) Online Rules Editing

The view "Generate Customer Greeting" will allow a user to enter basic information about a customer and will generate a greeting like "Good Morning, Mrs. Robinson!" based on the current time. Here is an example of the proper view:

By clicking on the button "Generate Greeting" a user could produce a new greeting in accordance with the latest greeting and salutation rules. By clicking on the button, "Greeting Rules", a user will be taken to a web-based rule editor to modify the Greeting Rules:

By clicking on the button, "Salutation Rules", a user will be taken to a web-based rule editor to modify the Salutation Rules:



This editor shows how to make changes in the rule attributes; it also allows a user to add rules by clicking on the hyperlink "Add Rule", or to delete rules by clicking on the red cross.

Step 2. Defining Implementation Approach

We will build this web application using OpenRules® Forms by defining 3 Excel-based layouts for each of the above views and using navigation logic described as processing flow rules. We will deploy our application on the Tomcat server. As usual, we will create the following files:

File	Directory	Purpose
HelloExternalRulesFromGUI.xls	*./war/rules/main*	Describes the Environment table and the main method that will be executed during every interaction with a web client
HelloForms.xls	*./war/rules/gui*	Describes all screen layouts and processing flow rules

Dialog.xls	*./war/rules/gui*	The standard OpenRules® file borrowed from the project openrules.forms.lib
HelloData.xls	*./war/rules/data*	Rule templates
index.jsp	*./*	The entry point to this JSP-based web application

What makes this application special is the need to reinitialize the rule engine that generates a greeting each time the greetings and/or salutations have been modified. However, it is not necessary to reinitialize a rule engine associated with an already opened `OpenRulesSession` with all layouts and related rule tables. So, we need to carefully distributes greeting generation information and GUI information between two different rule engines while making sure that re-initialization of the first engine is done very quickly.

When we start an application for the first time, we want to display the default rules (defined in an Excel file) and we also want to use the default data about a customer (defined in another Excel file).

In this implementation, we will define a special Java class `HelloManager` whose responsibilities will include these and other data management tasks. The manager will support two rule engines:

1. A rule engine that reads the default greeting and salutation rules from the file, `war/rules/main/HelloDefaultRules.xls`. Only this engine will deal with greeting rules and rule templates presented in the file, `war/rules/logic/HelloTemplates.xls`.

2. A rule engine associated with the `OpenRulesSession` that will handle all GUI problems and will also read the default data about a customer from the Excel file, “HelloData.xls”.

Thus, the entry point to our web application "`index.jsp`" will look as follows:

```
<%@ page import="com.openrules.forms.gui.jsp.*" %>
<%@ page import="com.openrules.forms.*" %>
<%@ page import="hello.rules.*" %>
<%@ page import="com.openrules.ruleengine.*" %>

<%
String s_attr = "openrules_session";
OpenRulesSession openrules_session = (OpenRulesSession)
session.getAttribute(s_attr);
if (openrules_session == null ) {
    // Create manager using data from HelloDefaultRules.xls
    String xlsMainRules =
"file:./webapps/HelloExternaRulesFromGUI/rules/main/HelloDefaultRule
s.xls";
    HelloManager man = new HelloManager(xlsMainRules);
    // Create OpenRulesSession using HelloExternaRulesFromGUI.xls
    String xlsMainForms =
"file:./webapps/HelloExternaRulesFromGUI/rules/main/HelloExternaRule
sFromGUI.xls";
    openrules_session = new OpenRulesSession(xlsMainForms);
    session.setAttribute( s_attr, openrules_session);
    System.out.println("NEW SESSION based on " + xlsMainForms);
    man.setFormsEngine(openrules_session.getOpenRulesEngine());
    // Read default rules and data from Excel files
    man.getDefaults();
    Dialog dialog = openrules_session.getDialog();
    dialog.put("manager",man);
}
%>
<HTML><HEAD><TITLE>OpenRules</TITLE></HEAD>
<body>
<%
     System.out.println("PROCESS REQUEST");
     openrules_session.processRequest(session, request, out);
%>
</body>

</HTML>
```

The first rule engine will be created by the constructor HelloManager(xlsMainRules). The second rule engine, automatically created by the `OpenRulesSession`, will be set for HelloManager by the statement:

```
man.setFormsEngine(openrules_session.getOpenRulesEngine());
```

The Environment table for the first rule engine is located in the file HelloDefaultRules.xls:

Environment	
import.java	hello.rules.*
include	../logic/HelloTemplates.xls

The Environment table for the second rule engine is located in the file HelloExternaRulesFromGUI.xls:

Environment	
import.static	com.openrules.tools.Methods
import.java	hello.rules.*
include	../gui/Dialog.xls
	../data/HelloData.xls
	../gui/HelloForms.xls

The main execution loop is implemented by the following method:

Method TableLayout main(Dialog dialog)

```
HelloManager man = (HelloManager) dialog().get("manager");
if (man == null)
   return fatalErrorLayout("HelloManager is not defined if index.jsp");
defineNextProcessingStep(man);
if (dialog().errors == 0)
{
    processingFlowRules(man);
    defineNextProcessingStep(man);
}
return mainLayout();
```

Step 3. Creating Supporting Java Classes

We define a Java package,, "hello.rules" with the following classes:

Customer (Customer.java)

- String name
- String gender
- String maritalStatus
- int age

App (App.java)

- Customer customer
- String greeting
- String salutation

GreetingRule (GreetingRule.java)

- int from
- int to
- String greeting

These classes are basic Java beans used inside rules and forms. To demonstrate the use of a more complex rule editor, we will implement the rule table for

salutation rules as an OpenRules® dynamic table. To do this, we will define two classes:

SalutationRule implements Checkable (SalutationRule.java)

- String gender
- String maritalStatus
- String maxAge
- String salutation
- HelloManager manager

and the class, **SalutationRules,** that extends `DynamicTable` (see SalutationRules.java) by defining two methods:

```
public String getHeaderLayoutName() {
        return "salutationsTableHeader";
}
public String getRowLayoutName() {
        return "salutationsTableRow";
}
```

The main Java class is a placeholder for all other objects:

HelloManager (HelloManager.java)

- *OpenRulesEngine ruleEngine*
- *OpenRulesEngine formsEngine*
- *GreetingRule[] greetingRules*
- *SalutationRule[] defaultSalutationRules*
- *SalutationRules salutationRules*
- *App app*
- *ExternalRules externalRules*

The object, "ruleEngine", is defined in the constructor for the object, "formsEngine", defined in the index.jsp. When the application is initialized the manager executes the method "getDefaults":

```
public void getDefaults() {

     greetingRules =
        (GreetingRule[])ruleEngine.run("getDefaultGreetingRules");

     defaultSalutationRules =
       (SalutationRule[])ruleEngine.run("getDefaultSalutationRules");
     salutationRules = new SalutationRules(formsEngine);
     for (int i = 0; i < defaultSalutationRules.length; i++) {
            SalutationRule rule = defaultSalutationRules[i];
            rule.setManager(this);
            salutationRules.addNewRow(rule);
     }

     createExternalRules();
     externalRules.setModified(true);
     ruleEngine.log("There is " +
                    getExternalRules().getRuleTables().size()
                    + " external tables");

     Customer customer =
            (Customer) formsEngine.run("getDefaultCustomer");
     app = new App();
     app.setCustomer(customer);
}
```

This method receives the *greetingRules* from the file *HelloDefaultRules.xls* using the method "getDefaultGreetingRules". It receives the *defaultSalutationRules* using the method "getDefaultSalutationRules" and then creates *salutationRules* to support the proper dynamic graphical table. It then creates an instance of the type ExternalRules, using this method:

```
public void createExternalRules() {

     String[][] greetingGrid = new String[greetingRules.length][3];
     for (int i = 0; i < greetingRules.length; i++) {
            GreetingRule rule = greetingRules[i];
            greetingGrid[i] = new String[] {
                     Integer.toString(rule.from),
                     Integer.toString(rule.to),
                     rule.greeting
            };
     }
```

```
    String[][] salutationGrid =
                    new String[salutationRules.getRows().size()][4];
    for (int i = 0; i < salutationRules.getRows().size(); i++) {
      SalutationRule rule =
                   (SalutationRule)salutationRules.getRows().get(i);
      salutationGrid[i] = new String[] {
               rule.gender,
               rule.maritalStatus,
               rule.maxAge,
               rule.salutation
       };
    }

    externalRules = new ExternalRules();
    externalRules.addRuleTable(
           "greetingRules",       //table name
           "defineGreeting",      //template name
           greetingGrid);

    externalRules.addRuleTable(
           "salutationRules",     //table name
           "defineSalutation", //template name
           salutationGrid);

    externalRules.setModified(false);
    ruleEngine.setExternalRules(externalRules);

}
```

And finally, the manager creates the default application, "app", with a customer received from the file *HelloData.xls*:

Data Customer customers			
customer.name	customer.maritalStatus	customer.gender	customer.age
Customer Name	**Marital Status**	**Gender**	**Age**
Robinson	Married	Female	24
Smith	Single	Male	19

Method Customer getCustomer()
return customers[0];

Step 4. Creating Graphical Layouts in Excel

All GUI realted forms and rules are described in the file HelloForms.xls. The "mainLayout" specifies a general layout for all three layouts:

Layout TableLayout mainLayout()		
properties	width	100%
	cellspacing	4
	cellpadding	2
	border	1
	style	background-color:lightblue
dialog().nextLayout		
  OpenRules, Inc. 		

The layout "GenerateGreeting":

Layout TableLayout generateGreetingLayout(App app, Customer c)		
Generate Customer Greeting		
currentTime()		
"Name:"	[c.name]	
"Age:"	[c.age]	
"Gender:"	[c.gender]["Male,Female"]	
"Marital Status:"	<F type="radio" >[c.maritalStatus]["Single,Married"] </F>	
actionButton("Generate Greeting");	actionButton("Greeting Rules");	actionButton("Salutation Rules");
"Generated Greeting:"	<C> app.result </C>	

There are two layouts to support "GreetingRules":

Layout TableLayout greetingRulesLayout(HelloManager man)		
Greeting Rules		
Rules "Define Greeting"		
greetingRulesTable(man);		
actionButton("Save Changes");	actionButton("Salutation Rules");	actionButton("Generate Greeting");

Layout TableLayout greetingRulesTable(HelloManager man)		
Hour From	Hour To	Greeting
[man.greetingRules[0].from]	[man.greetingRules[0].to]	[man.greetingRules[0].greeting][getPossibleGreetings()]
[man.greetingRules[1].from]	[man.greetingRules[1].to]	[man.greetingRules[1].greeting][getPossibleGreetings()]
[man.greetingRules[2].from]	[man.greetingRules[2].to]	[man.greetingRules[2].greeting][getPossibleGreetings()]
[man.greetingRules[3].from]	[man.greetingRules[3].to]	[man.greetingRules[3].greeting][getPossibleGreetings()]

This form has a fixed number of rules (rows), so a user may only change the values of rules attributes. The layout, "SalutationRules", represents a dynamic table:

Layout TableLayout salutationRulesLayout(HelloManager man)		
Salutation Rules		
Rules "Define Salutation"		
man.salutationRules.createTable();		
actionHyperlink("Add Rule");		
actionButton("Save Changes");	actionButton("Greeting Rules");	actionButton("Generate Greeting");

Layout TableLayout salutationsTableHeader()

Gender	Marital Status	Age Less Than	Salutation	Delete

Layout TableLayout salutationsTableRow(SalutationRule rule)				
[rule.gender] ["Male,Female"]	[rule.maritalStatus] ["Married,Single"]	[rule.maxAge]	[rule.salutation] [getPossibleSalutations()]	deleteRuleButton (rule);

Layout TableLayout deleteRuleButton(SalutationRule rule)
<F type="image" src="../openrules.forms.lib/images/delete.png"> [][] [rule.manager.salutationRules.deleteRow(rule); dialog().setLastAction("Delete Rule")] </F>

Here is the rule table that specifies processing flow:

Rules void processingFlowRules(HelloManager man)			
IF Current Step is	**AND Action is**	**THEN Execute Code**	**AND Go To The Step**
	Init		GenerateGreeting
GenerateGreeting		{ man.cleanUp(); }	GenerateGreeting
GenerateGreeting	Generate Greeting	{ man.generateGreeting(); }	GenerateGreeting
GenerateGreeting	Greeting Rules		GreetingRules
GenerateGreeting	Salutation Rules		SalutationRules
GreetingRules	Save Changes	{ man.updateRules(); }	GreetingRules
GreetingRules	Salutation Rules		SalutationRules
GreetingRules	Generate Greeting		GenerateGreeting
SalutationRules	Save Changes	{ man.updateRules(); }	SalutationRules
SalutationRules	Add Rule	{ man.addSalutationRule(); }	SalutationRules
SalutationRules	Delete Rule		SalutationRules
SalutationRules	Greeting Rules		GreetingRules
SalutationRules	Generate Greeting		GenerateGreeting

As you can see, the action "Save Changes" leads to the execution of the manager's method "updateRules":

```
public void updateRules() {
          createExternalRules();
          getExternalRules().setModified(true);
          showRules();

}
```

This method will create a new instance of the external rules, (based on the latest changes introduced by the rule editor), and it will mark the external rules as "modified", which will force a rule engine to reinitialize itself before the next run of the method "generateGreeting":

```
public void generateGreeting() {
          ruleEngine.run("greetingRules",app);
          ruleEngine.run("salutationRules",app);

}
```

Step 5. Deploying and Executing the Web Application

To deploy this web application on the Tomcat server specified in the file *build.properties*, it is enough to double-click on *deploy.bat*. To start the application make sure that your Tomcat is up and running and double-click on *run.bat*.

OPENRULES® PROJECTS

Pre-Requisites

OpenRules® requires the following software:

- Java SE JDK 1.5 or higher
- Apache Ant 1.6 or higher
- MS Excel or OpenOffice or Google Docs (for rules and forms editing only)
- Eclipse SDK (optional, for complex project management only)

Sample Projects

The complete OpenRules® installation includes the following workspaces:

openrules.decisions - decision models projects

openrules.rules - various rules projects

openrules.dialog – rules-based web questionnaires

openrules.web - rules-based web applications & web services

openrules.solver - constraint-based applications.

Each project has its own subdirectory, e.g. "DecisionHello". OpenRules® libraries and related templates are located in the main configuration project, "openrules.config", included in each workspace. A detailed description of the sample projects is provided in the Installation Guide.

Main Configuration Project

OpenRules® provides a set of libraries (jar-files) and Excel-based templates in the folder "`openrules.config`" to support different projects.

Supporting Libraries

All OpenRules® jar-files are included in the folder, "openrules.config/lib". For the decision management projects you need at least the following jars:

- openrules.all.jar
- poi-3.6-20091214.jar
- commons-logging-1.1.jar (or higher)
- commons-logging-api-1.1.jar (or higher)
- commons-lang-2.3.jar (or higher)
- log4j-1.2.15.jar (or higher)
- commons-beanutils.jar (or higher)

If you use supporting libraries with convenience Operators, methods like "out", and a simple JDBC interface you will also need

- com.openrules.tools.jar

If you use the JSR-94 interface you will also need

- com.openrules.jsr94.jar

If you use external rules from a database you will also need

- openrules.db.jar
- openrules.dbv.jar
- derby.jar
- commons-cli-1.1.jar.

Different workspaces like "`openrules.decisions`", "`openrules.rules`", etc. include the proper versions of the folder "`openrules.config`".

Predefined Types and Templates

The Excel-based templates that support the Decision Model included in the folder, "openrules.config":

- `DecisionTempaltes.xls`
- `RuleFamilyExecuteTemplates.xls`
- `RuleFamilyValidateTemplates.xls`
- `RuleFamily1ExecuteTemplates.xls`
- `RuleFamily2ExecuteTemplates.xls`

Sample decision projects include Excel tables of the type "Environment" that usually refer to "`../../../openrules.config/DecisonTemplates.xls`". You may move all templates to another location and simply modify this reference making it relative to your main xls-file.

TECHNICAL SUPPORT

Direct all your technical questions to support@openrules.com.