Oracle® Cloud

Designing Business Rules with Oracle Business Process Management

12*c* (12.1.3)

E60652-03

Ocotber 2016

Documentation for developers and business users that provides information about using and developing applications involving facts, rules, and decision tables for Oracle Business Rules by using design-time tools, such as Oracle JDeveloper with Oracle SOA extension, and a runtime application such as Oracle SOA Composer.

ORACLE

Oracle Cloud Designing Business Rules with Oracle Business Process Management, 12*c* (12.1.3)

E60652-03

Primary Author: Oracle Corporation

Contents

2 Working with Data Model Elements

3 Working with Facts and Value Sets

4 Working with Rulesets and Rules

6 Working with Decision Functions

9 Creating a Rule-enabled Non-SOA Java EE Application

10 Working with Oracle Business Rules and ADF Business Components

Preface

This guide describes how to design Oracle Business Rules.

Audience

Designing Business Rules with Oracle Business Process Management is intended for application programmers, system administrators, and other users who perform the following tasks:

- Create Oracle Business Rules programs
- Modify or customize existing Oracle Business Rules programs
- Create Java applications using rules programs
- Add rules programs to existing Java applications

To use this document, you need a working knowledge of Java programming language fundamentals.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documentation

For more information, see the following Oracle Resources:

- *Rules Language Reference forOracle Business Process Management*
- *Managing and Monitoring Processeswith Oracle Business Process Management*
- *Developing SOA Applications with Oracle SOA Suite*
- Java API Reference for Oracle Business Rules

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in This Guide

This section summarizes the new features and significant product changes for Oracle Business Rules in the Oracle Fusion Middleware Release 12*c* (12.1.3) release.

Screens shown in this guide may differ from your implementation, depending on the skin used. Any differences are cosmetic.

Follow the pointers into this guide to get more information about the features and how to use them. This document is the new edition of the formerly titled Oracle Fusion Middleware User's Guide for Oracle Business Rules.

For a list of known issues (release notes), see the "Known Issues for Oracle SOA Products and Oracle AIA Foundation Pack" at `http://www.oracle.com/technetwork/middleware/docs/soa-aiafp-knownissuesindex-364630.html`

New and Changed Features for 12*c* (12.1.3)

Oracle Rules Language release 12*c* (12.1.3) includes the following new and changed features:

- Support for a new non-Rete Business Rules algorithm. See The Non-Rete Algorithm.
- Support for business phrases and verbal rules. Verbal rules work hand in hand with business phrases to provide a flexible way author rules using natural language statements to express rule logic in domain specific sentences that are similar to spoken language. See Introduction to Verbal Rules and Business Phrases.
- Support for a simpler way to test rules with complex input. The test feature enables both developers and business users to quickly check that a rule satisfies the expected behavior or, if modified, to see if a rule regresses existing functionality. See Testing and Validating Business Rules.
- Support for comparing and merging two or more dictionaries. See How to Compare or Merge Two or More Dictionaries.
- Support for editing Decision Tables in Microsft Excel. Business users may find that editing Decision Tables is easier to do in Microsoft Excel. New functionality enables both developers and business users to export and edit Decision Tables in Excel and then import the Decision Tables back into the dictionary. See Editing Decision Tables in Microsoft Excel.

1

Overview of Oracle Business Rules

This chapter describes the concepts of business rules and provides an overview of the Oracle Business Rules runtime and design-time elements such as facts, valuesets, rulesets, decision tables, Oracle BP Composer and Oracle SOA Composer. It also describes the Oracle Business Rules engine architecture.

This chapter includes the following sections:

- Introduction to Oracle Business Rules
- Understanding Oracle Business Rules Formats
- Oracle Business Rules Runtime and Design Time Elements
- Oracle Business Rules Engine Architecture

For more information on any references to Rules Language and Rules API in this book, see the Related Documentation section of this book.

Differences Between Using this Component in the Cloud and On-Premises Environments

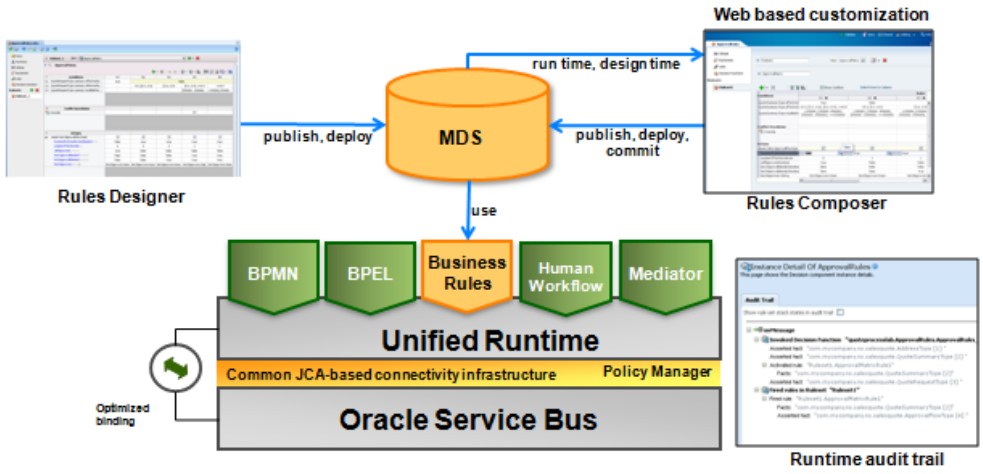
There may be differences between using this component in the cloud and on-premises environments that impact the information described in this guide.

For information about differences, see Differences Between the Cloud and On-Premises Environments and Known Issues for Oracle SOA Cloud Service.

Introduction to Oracle Business Rules

Oracle Business Rules makes processes and applications more flexible by enabling business analysts and non-developers to easily define and modify business logic without programming. By leveraging the unified JDeveloper design platform, and maintaining business rules outside of the related process or application, Oracle Business Rules provides faster, easier rule modifications and reduces subsequent redeployment costs.

Figure 1-1 Oracle Business Rules



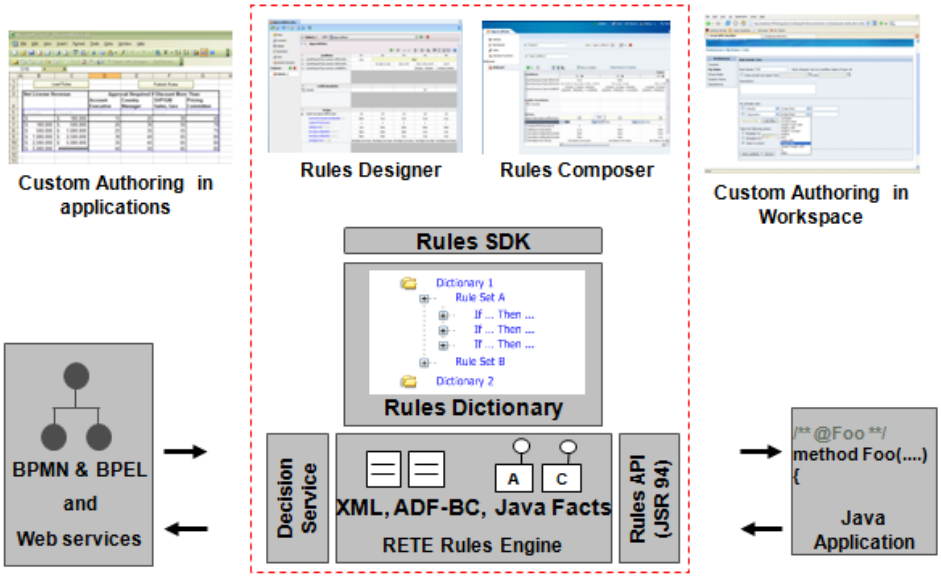
Using Oracle Business Rules you can automate policies, computations, and reasoning while separating rule logic from underlying application code. This allows more agile rule maintenance and empowers business analysts to modify rule logic without programmer assistance and without interrupting business processes.

As a business analyst, a user can model a rule in Process Composer and further refine and complete the process in Process Studio.

An IT developer uses the BPM Studio and talk to its business catalogue with well known data types, services, and human tasks implementation. The developer, then, creates a project and publishes it into the business catalogue. Then a business analyst or a business user can go in Process Composer and check out the projects in business catalogue and make changes to the existing process models.

While some users want to model simple calculations with a handful of rules, others use rules for complex decision making and hence a need to have a methodology to approach the decision modeling problem.

Figure 1-2 Oracle Business Rules Components



The Oracle Business Rules includes the rule editor, rule browser, rules engine and rule repository for rule discovery, governance, versioning, traceability and availability across the enterprise. Business rules are defined using the Business Rules editor and stored and managed in a central Business Rules repository. You can reference pre-defined business process rules within the modeler. The Business Rules activity in the business process model gets converted to a decision service that in turn invokes the business rules engine in the executable business process. Business users can change these business policies on the fly via an intuitive web browser interface without having to redeploy or re-implement the business process.

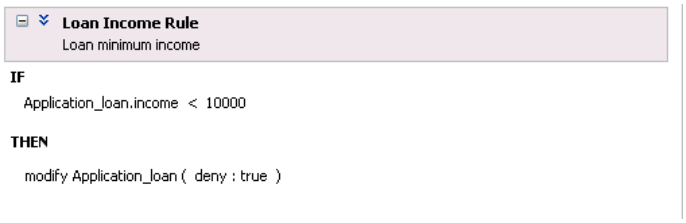
Why Use Oracle Business Rules?

Oracle Business Rules is a high performance lightweight business rules product that addresses the requirements for agility, business control, and transparency.

Business rules are statements that describe business policies or describe key business decisions. For example, business rules can include:

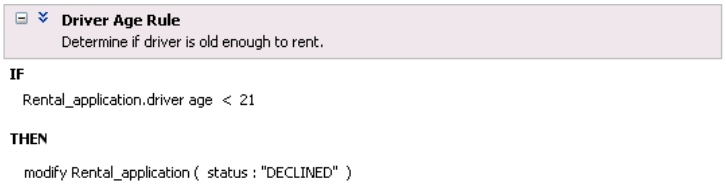
- Business policies such as spending policies and approval matrices.

  A financial institution could use a business rule such as:



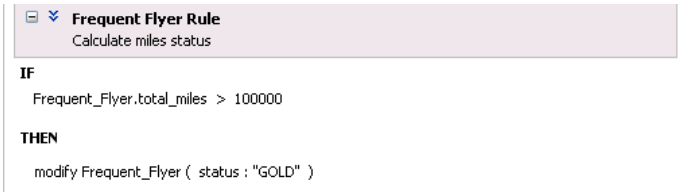
- Constraints such as valid configurations or regulatory requirements.

  For example, a car rental company might use the following business rule:



- Computations such as discounts or premiums.
- Reasoning capabilities such as offers based on customer value.

  An airline might use a business rule such as the following:



These examples represent individual business rules. In practice, you can use Oracle Business Rules to combine many business rules or to use more complex tests.

For the car rental example, you can name the rule the Driver Age Rule. Traditionally, business rules such as the Driver Age Rule are buried in application code and might appear in a Java application as follows:

```
public boolean checkDriverAgeRule (Driver driver) {
   boolean declineRent = false;
   int age = driver.getAge();
   if(  age < 21 ) {
      declineRent = true;
   }
   return declineRent;
}
```

This code may be difficult for nontechnical users to understand and modify. For example, suppose that the rental company changes its policy so that all drivers under 18 are declined using the Driver Age Rule. In many production environments the developer must modify the application, recompile, and then redeploy the application. This process is simplified because a business analyst can change policies that are expressed as business rules, with little or no assistance from a programmer. Applications using Oracle Business Rules support continuous change that allows the applications to adapt to new government regulations, improvements in internal company processes, or changes in relationships between customers and suppliers.

Understanding Oracle Business Rules Terminology

A business rule must contain:

- Rulesets: A set of conditions or actions that determines the outcome of the rule.
- Facts: Data objects used by the ruleset.
- Decision functions: Reference to the code that executes the rule.

Additionally, a business rule may contain:

- Functions: Functions that may be called in the ruleset. An example of this type of function is one that initializes a data object.
- Globals: Data objects that are used in the ruleset. May be constants.
- Valuesets: Lists or ranges of values used by the condition.
- Links: Links to other business rules dict.

The following sections provide additional details about these components.

What Are Facts and Valuesets?

In Oracle Business Rules, rules are written in terms of fact types. Each fact is an instance of a fact type. You must import or create one or more fact types before you can create rules, unless you use Verbal Rules, where you have the option of deferring fact type modeling until the executable rule is defined.

In Oracle Business Rules, a FactType is a type definition in the data model and a fact is an instance of that type. For example, rules are written in terms of fact types. The Oracle Business Rules runtime, or a developer writing in the RL Language, uses the RL Language `assert` function to add an instance of a fact to the Oracle Business Rules. In Rules Designer you can define a variety of fact types based on XML Schema, Java classes, Oracle RL definitions, and ADF Business Components view objects.

You can create valuesets to define a list of values or a range of values of a specified type. After you create a valueset you can associate the valueset with a fact property of

matching type. Oracle Business Rules uses the valuesets that you define to specify constraints on the values associated with fact properties in rules or in Decision Tables. You can also use valuesets to specify constraints for variable initial values and function return values or function argument values.

For more information, see:

- Working with Facts and Value Sets
- Oracle Business Rules Engine Architecture

What Are Rulesets?

A ruleset is an Oracle Business Rules container for IF-THEN rules and Decision Tables. A ruleset provides a namespace, similar to a Java package, for rules and Decision Tables. In addition you can use rulesets to partially order rule firing.

For more information, see:

- Working with Rulesets and Rules
- Ordering Rule Firing in the *Rules Language Reference for Oracle Business Process Management*

What Are Dictionaries?

A dictionary is an Oracle Business Rules container for facts, business phrases, functions, globals, valuesets, links, decision functions, and rulesets. A dictionary is an XML file that stores the application's rulesets and the data model. Dictionaries can link to other dictionaries. Oracle JDeveloper creates an Oracle Business Rules dictionary in a `.rules` file. You can create as many dictionaries as you need. A dictionary may contain any number of rulesets. For more information, see Introduction to Dictionaries and Dictionary Links.

What Are Globals?

Globals are any variables or constants that may be accessed anywhere in the business rule. When you create globals you ensure that a business user can alter the rule behavior without touching the rule logic.

What Are Decision Functions?

A decision function provides a contract for invoking rules from Java or SOA (from a SOA/BPM composite application or from components within the composite application). The contract includes input fact types, rulesets to run, and output fact types. For more information, see Working with Decision Tables.

What Are Decision Points?

Oracle Business Rules SDK (Rules SDK) provides APIs that let you write applications that access, create, modify, and execute rules in Oracle Business Rules dictionaries (and all the contents of a dictionary). The Rules SDK provides the Decision Point API to access and run rules or Decision Tables from a Java application. For more information, see Working with Rules in Standalone (Non SOA/BPM) Scenarios.

What Are Business Phrases?

Business phrases are vocabulary elements that are used to construct tests and actions for verbal rules. As you write a verbal rule, a set of business phrases, derived

automatically from terms, facts, globals and other dictionary elements, is made available for inclusion in tests and actions. You can define your own business phrases.

Business phrases are not used in general rules.

Understanding Oracle Business Rules Formats

Oracle Business Rules provides multiple approaches to writing rules. Rules can be modeled in different ways - as IF/THEN rules, and as Decision Tables.

There are two approaches to writing IF/THEN rules (or just rules) - as general rules, and as verbal rules.

- General rules use a pseudo-code language to express rule logic
- Verbal rules use natural language statements to express rule logic
- Decision Tables are multiple related rules expressed in a spreadsheet-like format.

You write rules and Decision Tables in terms of fact types and properties. See Decision Tables. Fact types are often imported from the Java classes, XML schema, Oracle ADF Business Components view objects, or may be created in Rules Designer. Fact properties have a name, value, data type, and an optional valueset. A valueset splits the value space of the data type into values or ranges that can be used in Decision Tables, choice lists, and for design time validation (see What Are Facts and Valuesets?).

You can write verbal rule tests and actions using derived business phrases as well as user-defined business phrases. Derived business phrases are automatically created using facts, globals and other information in the dictionary while user-defined phrases can be explicitly authored to augment derived phrases. Further, user-defined phrases can either be pre-created or created as needed while composing the verbal rule.

Rules and Decision Tables are grouped in an Oracle Business Rules object called a ruleset (see What Are Rulesets?).

You group one or more rulesets and their facts and valuesets in an Oracle Business Rules object called a dictionary (see What Are Dictionaries?).

For more information, see Oracle Business Rules Runtime and Design Time Elements.

Rules

Rules are used to evaluate conditions and specify actions when the conditions are met (evaluate to true).

You can model rules using two different paradigms:

- General rules - use a pseudo-code language to express rule logic.
- Verbal rules - use natural language statements to express rule logic in domain specific sentences that are akin to spoken language. See How are Verbal Rules Different from General Rules?

Rules follow an if-then structure and consist of the following parts:

- **IF** part: a condition or pattern match (see What Are Rule Conditions?).
- **THEN** part: a list of actions (see What Are Rule Actions?).

What Are Rule Conditions?

The rule **IF** part is composed of conditional expressions that refer to fact types.

For example, for a general rule:

IF Rental_application.driver age < 21

The general rule conditional expression compares a business term (Rental_application.driver age) to the number 21 using a less than comparison.

And for a verbal rule:

IF rental car driver is an underage driver

The verbal rule condition is a business phrase that can specify one or more logical tests. (See What Are Business Phrases?).

The rule condition activates the rule whenever a combination of facts makes the conditional expression true. In some respects, the rule condition is like a query over the available facts in the Rules Engine, and for every row returned from the query the rule is activated.

For more information, see:

- Working with Facts and Value Sets
- Working with Rulesets and Rules
- Rule Conditions in the *Rules Language Reference for Oracle Business Process Management* guide.

What Are Rule Actions?

The rule **THEN** part contains the actions that are executed when the rule is fired. A rule is fired after it is activated and selected among the other rule activations using conflict resolution mechanisms such as priority.

A rule might perform several kinds of actions. An action can add facts, modify facts, or remove facts. An action can execute a Java method or perform a function which may modify the status of facts or create facts.

Rules fire sequentially, not in parallel. Note that rule actions often change the set of rule activations and thus can affect which rule fires next.

For more information, see:

- Rule Firing and Rule Sessions
- Working with Rulesets and Rules
- Ordering Rule Firing in the *Rules Language Reference for Oracle Business Process Management* guide

How are Verbal Rules Different from General Rules?

Verbal rules allow you to use pseudo-natural language statements to express rule logic. They provide a way to write rules using domain specific sentences that are similar to spoken language.

Verbal rules work closely with business phrases, which provide the vocabulary for you to compose natural language tests and actions. See What Are Business Phrases?.

For example, a general rule test as shown in the example below:

```
IF
all of the following are true
policy is a policy
policyScore.type == Score Type.Policy
policyScore.id == policy.id
car is a Car
carScore is a Score Tracker
carScore.type == Score Type.Car
carScore.id == car.id
customer is a Customer
customerScore is a Score Tracker
customerScore.type == Score Type.Customer
customerScore.id == customer.id
score of a car == carScore.score
score of customer == customerScore.score
score of policy = policyScore.score

THEN
assign new BigDecimal var = newBigDecimal((1+((2-((customerScore + carScore +
policyScore)/150))/100))*(Lower Threshold))
assign new double premium = var.setScale(1,BigDecimal.ROUND_HALF_UP).doubleValue()
```

The verbal rule expression of this same test might be:

```
IF
ready to calculate premium
THEN
calculate premium base on score of customer, score of policy and score of car
```

Business phrases such as 'ready to calculate premium', 'score' and so on would detail the logic for the conditions.

You can write verbal rules in a way that suits your style, 'top down' or 'bottom up'.

For example, you can write a verbal rule using business phrases that are not yet defined. Once you have the a verbal rule that expresses the logic for your tests and actions, you can then define the specifics of the business phrases.

You can also compose verbal rules using system provided derived business phrases. These are business phrases that are automatically created based on the existing terms, facts, globals and other dictionary elements.

Alternatively, you can write verbal rules using a bottom up style, by defining all the business phrases you'll need first, and then using them in the tests and actions of your verbal rules.

For more information, see Working with Rulesets and Rules

Decision Tables

A Decision Table is an alternative business rule format that is more compact and intuitive when many rules are needed to analyze many combinations of property values. You can use a Decision Table to create a set of rules that covers all combinations or where no two combinations conflict.

Although a decision table is functionally equivalent to if-then rules, you will find decision tables are ideal for specific circumstances:

- Complexity

Decision tables simplify complex rules. When there are multiple rules, each of which have multiple conditions and actions, a decision table is much easier to work with.

- Conflict resolution

  A decision table will indicate if any of the conditions are in conflict.

- Gap analysis

  You can analyze a decision table to determine if some conditions are not being accommodated.

For more information, see Working with Decision Tables.

Oracle Business Rules Runtime and Design Time Elements

Oracle Business Rules provides support for using business rules as a Decision Component or as a library in a Java application. A Decision Component is a mechanism for publishing rules and rulesets as a reusable service that can be invoked from multiple business processes.

To create and use rules in the Oracle SOA Suite or Oracle BPM Suite, or to create rules and integrate these rules into your applications, Oracle Business Rules provides the following runtime and design time elements:

- Decision Component (Business Rules) in a SOA Composite Application
- Using Rules Engine with Oracle Business Rules in a Java EE Application
- Oracle Business Rules RL Language
- Oracle Business Rules SDK
- Rules Designer
- Oracle SOA Composer Application
- Using BP Composer

Decision Component (Business Rules) in a SOA Composite Application

Oracle SOA Suite provides support for Decision Components that support Oracle Business Rules. A Decision Component is a mechanism for publishing rules and rulesets as a reusable service that can be invoked from multiple business processes.

A Decision Component is an SCA component that can be used within a composite and wired to a BPEL component. Apart from that, Decision Components are used for dynamic routing capability of Mediator and Advanced Routing Rules in Human Workflow.

Oracle Business Rules Rules Engine (Rules Engine) is available in a SOA composite application using the SOA Business Rule service engine that efficiently applies rules to facts and defines and processes rules.

For more information, see Oracle Business Rules Engine Architecture.

Using Rules Engine with Oracle Business Rules in a Java EE Application

The Rules Engine is available as a library for use in a Java EE application (non-SOA). Rules Engine efficiently applies rules to facts and defines and processes rules. Rules

Engine defines a Java-like production rule language called Oracle Business Rules RL Language (RL Language), provides a language processing engine (inference engine), and provides tools to support debugging.

Using Rules Designer you can specify business rules separately from application code which allows you to change business policies quickly with graphical tools. The Rules Engine evaluates the business rules and returns decisions or facts that are then used in the business process.

A rule-enabled Java application can load and run rules programs. The rule-enabled application passes facts and rules to the Rules Engine (facts are asserted in the form of Java objects or XML documents). The Rules Engine runs in the rule-enabled Java application and uses the Rete algorithm to efficiently fire rules that match the facts.

For more information, see Oracle Business Rules Engine Architecture and Oracle Business Rules SDK.

Oracle Business Rules RL Language

Oracle Business Rules supports a high-level Java-like language called Oracle Business Rules RL Language (RL Language). RL Language defines the valid syntax for Oracle Business Rules programs. RL Language includes an intuitive Java-like syntax for defining rules that supports the power of Java semantics, providing an easy-to-use syntax for application developers. RL Language consists of a collection of text statements that can be generated dynamically or stored in a file.

Using RL Language application programs can assert Java objects as facts, and rules can reference object properties and invoke methods. Likewise, application programs can use XML documents or portions of XML documents as facts.

Programmers can use RL Language as a full-featured rules programming language both directly and as part of the Oracle Business Rules SDK (Rules SDK).

Business analysts can use Rules Designer to work with rules. In this case, the business analyst does not need to directly view or write RL Language programs. For more information, see Rules Designer.

Oracle Business Rules SDK

Oracle Business Rules SDK (Rules SDK) is a Java library that provides business rule management features that a developer can use to write a rule-enabled program that accesses a dictionary, or to write customized rules programs that add rules or modify existing rules. Rules Designer uses Rules SDK to create, modify, and access rules and the data model using well-defined interfaces. Customer applications can use Rules SDK to access, display, create, and modify collections of rules and the data model.

You can use the Rules SDK APIs in a rule-enabled application to access rules or to create and modify rules. The rules and the associated data model could be initially created in a custom application or using Rules Designer.

This guide describes the Oracle Business Rules SDK Decision Point API. Using a Decision Point you can access a dictionary and run the rules in the dictionary.

For more information, see Working with Rules in Standalone (Non SOA/BPM) Scenarios.

Rules Designer

The Oracle Business Rules Designer (Rules Designer) extension to Oracle JDeveloper is an editor that enables you to create and edit rules.

Rules Designer provides a point-and-click interface for creating and editing General Rules and Decision Tables. Because you can work directly with business rules and a data model, you do not need to understand the RL Language to work with Rules Designer.

Rules Designer also provides Verbal Rules, with guided authoring (auto-suggest and filtering), and a keyboard based interface. For more information on using guided authoring and keyboard based interface, see How to Add Verbal Rules in SOA Composer.

Rules Designer supports several types of users, including the application developer and the business analyst. The application developer uses Rules Designer to define a data model and an initial set of rules. The business analyst uses Rules Designer either to work with the initial set of rules or to modify and customize the initial set of rules according to business needs. Using Rules Designer, a business analyst can create and customize rules with little or no assistance from a programmer.

Alternatively, in top-down modeling, a Business Analyst can descriptively define the rules which can be implemented by the developer later. These different modeling approaches require collaboration between the developer and the analyst.

In most cases, Rule modeling is done iteratively, with both of them contributing to the creation of a Domain Specific Language that can be used to define rules using less technical and more natural-language like sentences.

For more information about verbal rules, see Working with Rulesets and Rules.

Oracle SOA Composer Application

When a dictionary is deployed in a SOA composite application, Oracle Business Rules lets you view the dictionary or edit and save changes to the dictionary. You can use the SOA Composer application (SOA Composer) to work with a deployed dictionary that is part of a SOA composite application.

For more information, see Using Oracle SOA Composer with Oracle Business Rules at Runtime.

Oracle Business Process Composer Application

The Business Process Composer rules editor enables you to view and edit a rules dictionary. Rules dictionaries are displayed in a tabbed window similar to the process editor and data association editor.

For more information on using Rules in BP Composer, see Working with Oracle Business Process Composer Rules Editor in *Oracle Fusion Middleware Developing Business Processes with Oracle Business Process Composer*.

Oracle Business Rules Engine Architecture

A rule-based system using the Rete algorithm is the foundation of Oracle Business Rules.

A rule-based system consists of the following:

- The rule-base: Contains the appropriate business policies or other knowledge encoded into IF/THEN rules, verbal rules and Decision Tables.
- Working memory: Contains the information that has been added to the system. With Oracle Business Rules you add a set of facts to the system using assert calls.

- Inference Engine: The Rules Engine, which processes the rules, performs pattern-matching to determine which rules match the facts, for a given run through the set of facts.

In Oracle Business Rules the rule-based system is a data-driven **forward chaining system**. The facts determine which rules can fire so when a rule fires that matches a set of facts, the rule may add facts and these facts are again run against the rules. This process repeats until a conclusion is reached or the cycle is stopped or reset. Thus, in a forward-chaining rule-based system, facts cause rules to fire and firing rules can create more facts, which in turn can fire more rules. This process is called an **inference cycle.**

A Non-Rete Algorithm is also available for use. For more information about both, see The Rete Algorithm and The Non-Rete Algorithm.

Declarative Rules

With Oracle Business Rules you can use declarative rules, where you create rules that make declarations based on facts rather than coding. Here is an example of declarative rules:

```
IF a Customer is a Premium customer, offer them 10% discount
IF a Customer is a Gold customer, offer them 5% discount
```

In declarative rules:

- Statements are declared without any control flow.
- Control flow is determined by the Rules Engine.
- Rules are easier to maintain than procedural code.
- Rules relate well to business user work methods.

When a rule adds facts and these facts run against the rules, this process is called an **inference cycle**. An **inference cycle** uses the initial facts to cause rules to fire and firing rules can create more facts, which in turn can fire more rules. For example, using the initial facts, Rules Engine runs and adds an additional fact, and an additional rule tests for conditions on this fact creating an inference cycle:

```
IF a Customer is a Premium customer, offer them 10% discount
IF a Customer is a Gold customer, offer them 5% discount
IF a Customer spends > 1000, make them Premium customer
```

The inference cycle that Oracle Business Rules provides enables powerful and modular declarative assertions.

The Rete Algorithm

The Rete algorithm was first developed by artificial intelligence researchers in the late 1970s and is at the core of Rules Engines from several vendors. Oracle Business Rules uses the Rete algorithm to optimize the pattern matching process for rules and facts. The Rete algorithm stores partially matched results in a single network of nodes in working memory.

By using the Rete algorithm, the Rules Engine avoids unnecessary rechecking when facts are deleted, added, or modified. To process facts and rules, the Rete algorithm creates and uses an input node for each fact definition and an output node for each rule.

Fact references flow from input to output nodes. In between input and output nodes are test nodes and join nodes. A test occurs when a rule condition has a Boolean

expression. A join occurs when a rule condition ANDs two facts. A rule is activated when its output node contains fact references. Fact references are cached throughout the network to speed up recomputing activated rules. When a fact is added, removed, or changed, the Rete network updates the caches and the rule activations; this requires only an incremental amount of work.

The Rete algorithm provides the following benefits:

- Independence from rule order: Rules can be added and removed without affecting other rules.
- Optimization across multiple rules: Rules with common conditions share nodes in the Rete network.
- High performance inference cycles: Each rule firing typically changes just a few facts and the cost of updating the Rete network is proportional to the number of changed facts, not to the total number of facts or rules.

The Non-Rete Algorithm

The Non-Rete algorithm (NRE) is an alternative to the Rete algorithm that consumes less memory than the Rete algorithm. For many business rules use cases it will also result in improved performance. The core of NRE algorithm is a new rule condition evaluation approach. Key points about the new algorithm:

- Simpler internal rule representation.
- Byte code generated for rule tests, rule actions, and user defined functions.
- More efficient modify operation.
- Rule conditions not evaluated until the containing ruleset is on the top of the stack. After initial evaluation, re-evaluation occurs on fact operations as needed.
- Ability to avoid unnecessary re-evaluation when rulesets are only present on the ruleset stack once during rule execution.
- Preserves rule execution semantics.

The two main differences between the two algorithms are:

- Rule condition evaluation:
  - In the Rete algorithm, rule conditions are evaluated when fact operations occur (assert, modify, retract).
  - In the Non-Rete algorithm, rule conditions are evaluated for the first time when the ruleset is on the top of the stack, then on fact operations after that.
- Rule firing order. There are cases where the rule firing order is not defined, for example when a single fact activates multiple rules at the same time and the priorities are identical. In these cases, the order in which the rule activations fire may be different.

Note:

It is possible that an existing set of rules has an implicit dependency on the order in which the rules fire with the Rete algorithm even though that order may not be defined. The order may be different with the Non-Rete algorithm which may expose a latent bug in the rules as authored.

Configuring the Non-Rete Algorithm

In Rule Designer, the algorithm can be selected in the Dictionary Settings panel in the preferences tab. Algorithm selection is automatically handled for SOA and BPM composite applications. For JEE applications or other non-SOA/BPM applications, the algorithm selection will need to be specified when the RuleSession or RuleSessionPool is created.

For more information about RuleSessions, see Using a RuleSession in the *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*.

It is common that multiple rulesets are executed during a rule execution. It is also common that each ruleset is pushed onto the ruleset stack once and once rules in that ruleset have completed firing, it is not pushed onto the stack again during that rule execution. With the Non-Rete algorithm additional performance gain can be realized for these cases by specifying that the rulesets will only appear on the stack once. When the Non-Rete algorithm is selected, click the 'Rulesets Are On Stack Once' check box in a decision function definition to enable this feature.

For information about when to use the Rete or Non-Rete algorithms, see Rules Engine Algorithm in the *Rules Language Reference for Oracle Business Process Management*.

What Is Working Memory?

Oracle Business Rules uses *working memory* to contain facts. Facts do not exist outside of working memory. A RuleSession contains the Oracle Business Rules working memory.

Rule Firing and Rule Sessions

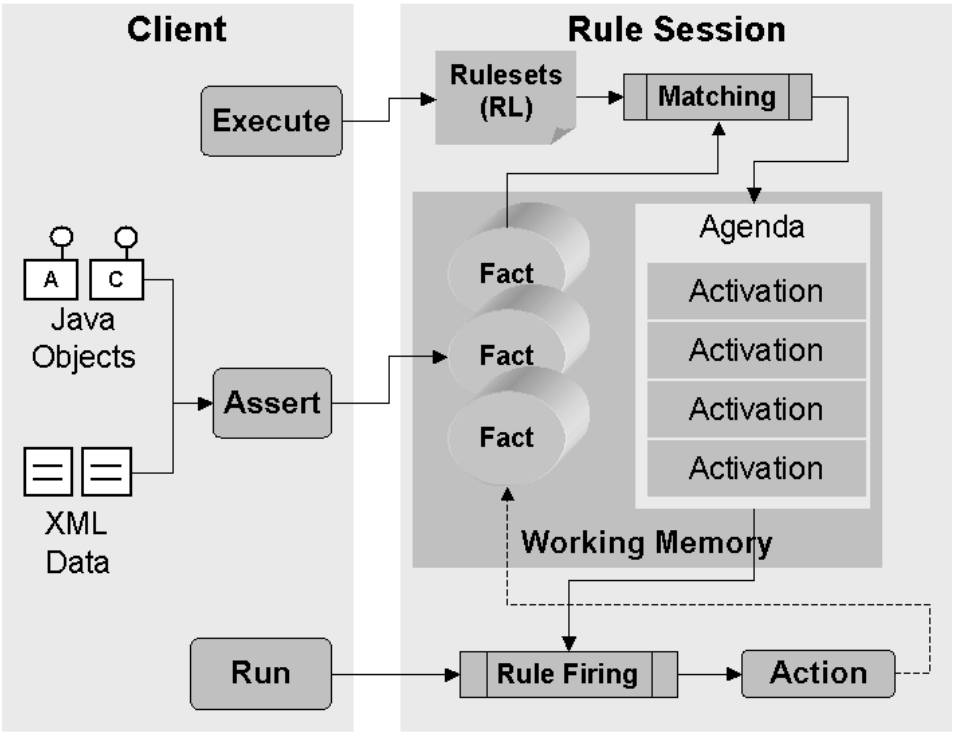
A Rule Session consists of rules, facts and an agenda. An assert or retract adds or removes fact instances from working memory.

When facts in working memory are changed:

- Conditions for rules are evaluated
- Matching rules are added to the agenda (Activated)
- Rules which no longer match are removed from agenda
- Rules Engine runs and executes actions (fires), for activated rules

Figure 1-3 shows these parts of Oracle Business Rules runtime.

Figure 1-3 Rules in Rule Session with Working Memory and Facts



A rule action may assert, modify, or retract facts and cause activations to be added or removed from the agenda. There is a possible loop if a rule's action causes it to fire again. Rules are fired sequentially, but in no pre-defined order. The rule session includes a ruleset stack. Activated rules are fired as follows:

- Rules within top-of-the-stack ruleset are fired
- Within a ruleset, firing is ordered by user-defined priority
- Within the same priority, the default is that the most recently activated rule is fired first. For more information, see the setStrategy function in the *Rules Language Reference for Oracle Business Process Management*.

For the Rete algorithm, only rules within rulesets on the stack are fired, but all rules in a rule session are matched and, if matched, activated. For the non-Rete algorithm, this is true for rules in the ruleset on the top of the stack. It is also true for rules in rulesets that have been popped from the ruleset stack unless "Rulesets Are On Stack Once" has been checked.

2

Working with Data Model Elements

This chapter describes the Oracle Business Rules data model comprising fact types, functions, globals, value sets, decision functions, and dictionary links.

The chapter includes the following sections:

Introduction to Working with Data Model Elements

To implement the data model portion of an Oracle Business Rules application you create a dictionary and add data model elements. To complete the dictionary, you create one or more rulesets containing rules that use or depend upon these data model elements.

For more information, see:

- Working with Facts and Value Sets
- Working with Rulesets and Rules
- Working with Decision Tables

Introduction to Dictionaries and Dictionary Links

A dictionary is an Oracle Business Rules container for facts, functions, globals, valuesets, links, decision functions, and rulesets. A dictionary is an XML file that stores the rulesets and the data model for an application. Dictionaries can link to other dictionaries.

You can create as many dictionaries as you need. A dictionary may contain any number of rulesets and data model elements. A data model can be contained in one or more dictionaries. All the data model elements referenced by the rulesets must be available in the dictionary either directly or through links.

A dictionary is stored in a `*.rules` file.

Working with Dictionaries and Dictionary Links

When you create a dictionary, you give it a name and a package, similar to a Java class. You can create data model elements and rulesets inside this dictionary, and you can also reference the data models and rulesets of other dictionaries by creating a dictionary link and specifying the name and package of the target dictionary. Each dictionary logically contains the built-in dictionary. This dictionary includes standard functions and types that all Oracle Business Rules applications need. You cannot modify the built-in dictionary.

In addition to the main dictionary, you can create one or more application-specific dictionaries, such as `PurchaseItems.rules`. You can modify the properties of these dictionaries.

The complete data model defined by a dictionary and its linked dictionaries is called a **combined data model**. You can create multiple links to the same dictionary; in this case, all but the first link is ignored.

For more information, see What You Need to Know About Dictionary Linking.

How to Create a Dictionary in the SOA Tier Using Rules Designer

Oracle JDeveloper provides multiple ways to create dictionaries for Oracle Business Rules. You can create a dictionaries for use in a SOA applications. This section illustrates one way to create a dictionary in a SOA project.

A typical SOA composite design pattern provides each application with its own dictionaries. Each application is self-contained and can be deployed independently of other applications.

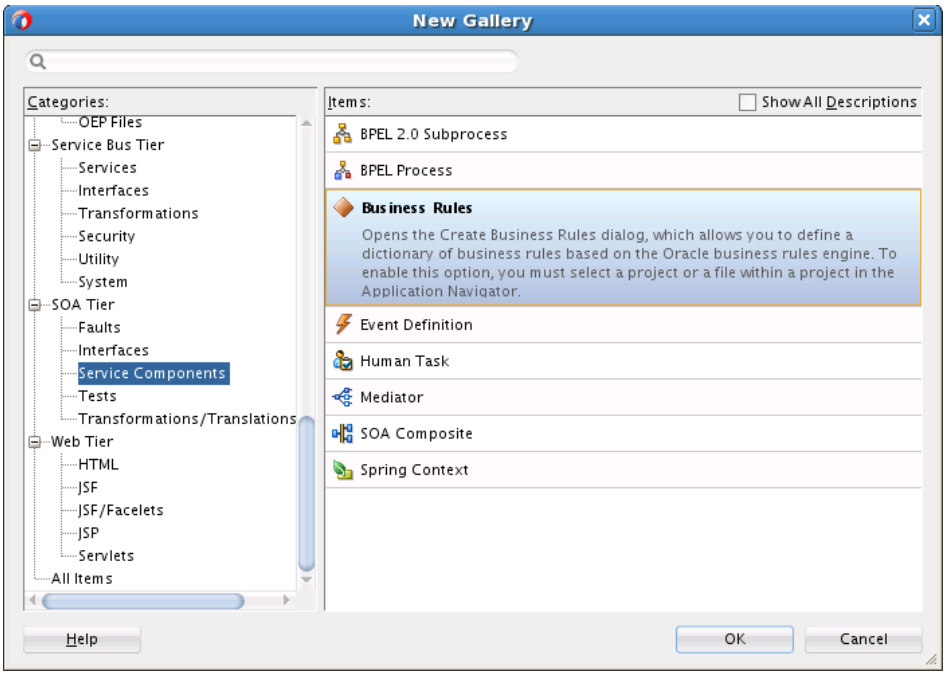
Sometimes multiple applications will require access to common parts of a common data model. In this case, use dictionary links to include a target application's dictionary in the source application. The source application copies the target application's dictionary and retains the contents of the copies linked to the source. When you use the linked elements, they are shown as local contents.

You can also create a dictionary in the business tier for use outside of a SOA application. For more information, see How to Create an Oracle Business Rules Dictionary in the Grades Project.

To create a dictionary in the SOA Tier using Rules Designer:

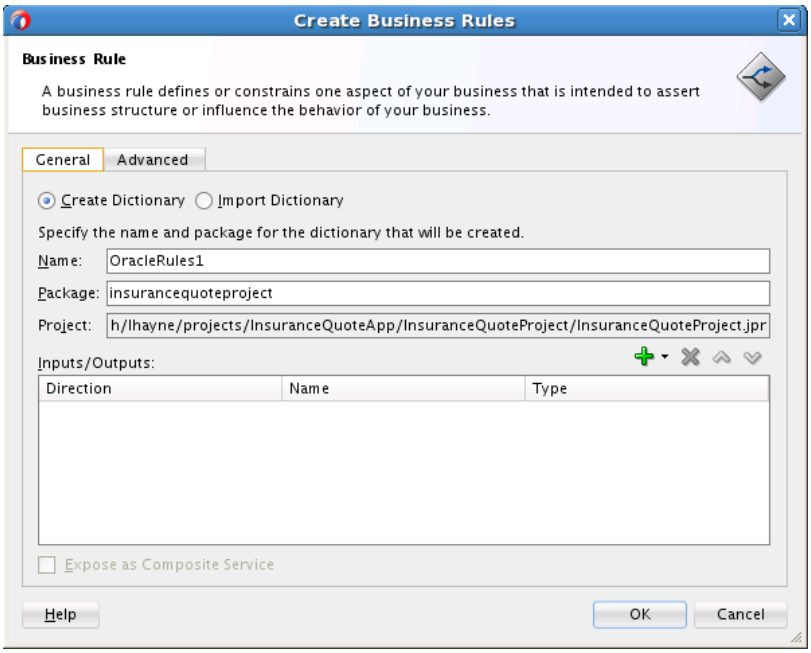
1. In the Application Navigator, select a SOA application and select or create a SOA project.
2. Click the down arrow, and select **New, From Gallery** from the list.
3. In the New Gallery dialog, expand **SOA Tier** as shown in Figure 2-1.

Figure 2-1 Creating a Business Rules Dictionary for a SOA Project



4. In the New Gallery window, select **Business Rules**.
5. Click **OK**. This displays the Create Business Rules dialog.
6. In the Create Business Rules dialog, enter fields as shown in Figure 2-2:
   - In the **Name** field, enter the name of your dictionary. For example, enter `PurchaseItems`.
   - In the **Package** field, enter the Java package to which your dictionary belongs. For example, `com.example`.

Figure 2-2 Create Business Rules Dialog



7. To specify the inputs and outputs:

   a. Click the **Add** button and select **Input** to create an input or **Output**, to create an output.

   b. In the Type Chooser dialog, expand the appropriate XSD and select the appropriate type.

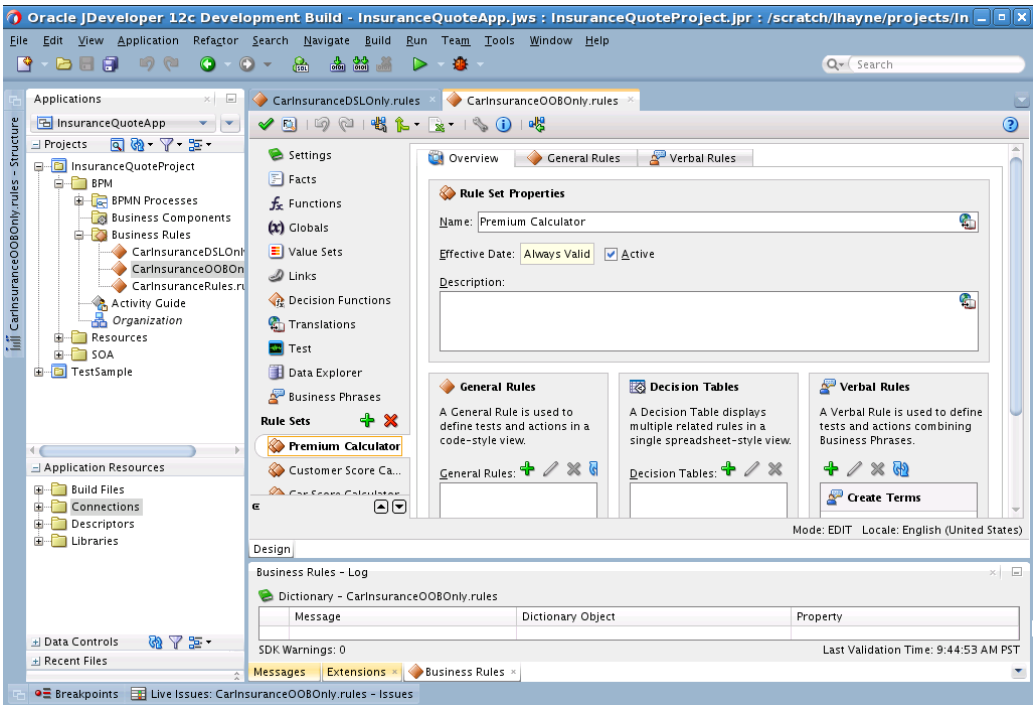
   c. Click **OK** to close the Type Chooser dialog.

   You can later add inputs or outputs, or remove the inputs or outputs. For more information, see Working with Decision Functions.

8. In the Create Business Rules dialog, click **OK** to create the Decision Component and the Oracle Business Rules dictionary.

   Oracle JDeveloper creates the dictionary in a file with a `.rules` extension, and starts Rules Designer as shown in Figure 2-3. Note the screen shot shows some BPM functionality that you may not have access to in SOA if you do not have BPM installed.

Figure 2-3 Creating a New Oracle Business Rules Dictionary



9. Oracle JDeveloper also creates a Decision Component in composite.xml. To view this component double-click the `composite.xml` file.

How to Create a Dictionary in the Business Tier Using Rules Designer

Use Rules Designer to create a rules dictionary for use in the business tier, outside of a SOA application. For information on using Oracle Business Rules without SOA, see Creating a Rule-enabled Non-SOA Java EE Application.

How to View and Edit Dictionary Settings

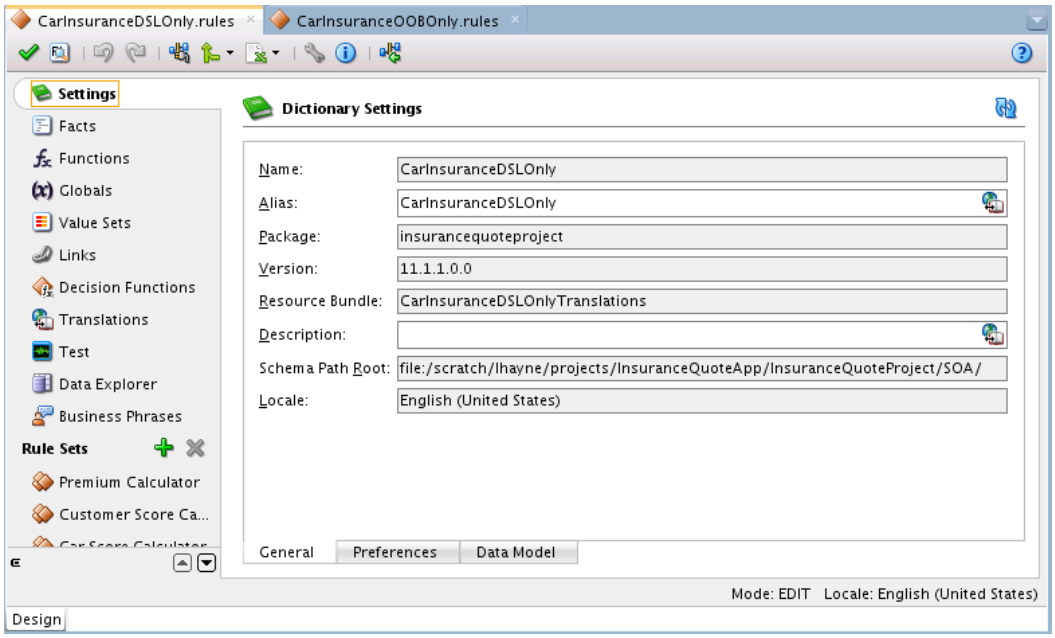
You can view and edit dictionary settings using the **Settings** tab. The **Settings** tab has three tabs: **General, Preferences,** and **Data Model.** Use the **Preferences** tab to select the execution algorithm and specify phrase suggestions that appear when you are using Verbal Rules. Use the **Data Model** tab to specify the global qualifier pattern, also for Verbal Rules. The pattern must contain two fragments: {member}, {fact}. For example, {member} of {fact}.

How to Change the Dictionary Alias

To change the Dictionary alias:

1. In Oracle JDeveloper, open an Oracle Business Rules dictionary.
2. In Rules Designer, click the **Settings** tab.
3. In the Dictionary Settings dialog, in the **Alias** field, change the alias to the name you want to use. This field is shown in Figure 2-4.

Figure 2-4 Dictionary Settings, General Tab



How to Edit the Preferences tab

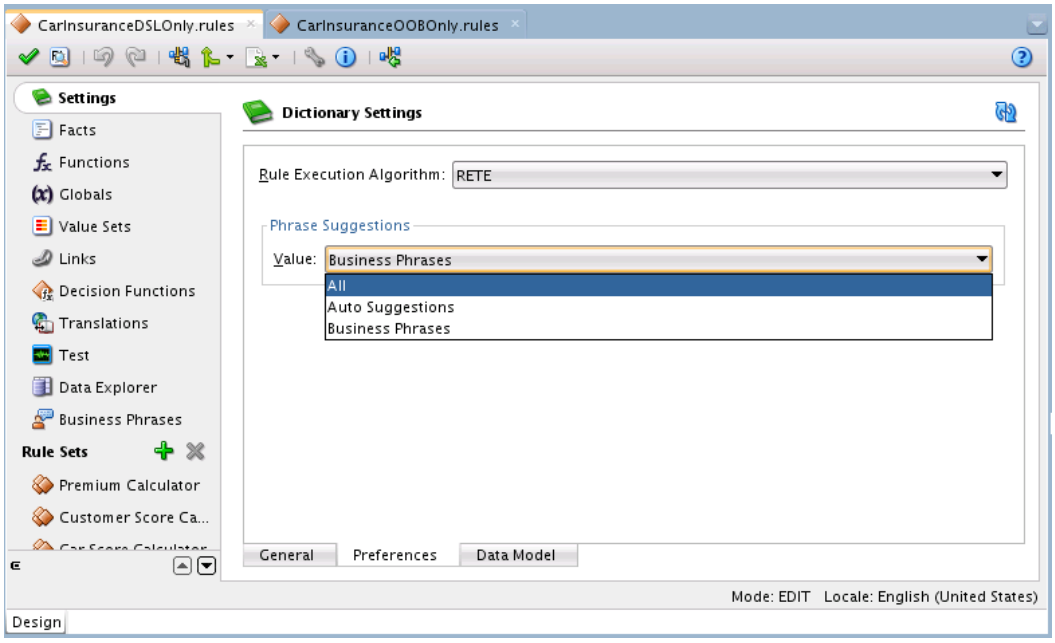
To edit the Preferences tab:

1. On the **Settings** tab, click the **Preferences** tab, shown in Figure 2-5.
2. In the **Rule Execution Algorithm** field, choose **RETE** or **Non-RETE.**

   For more information about the RETE or Non-RETE algorithm, see Configuring the Non-Rete Algorithm.
3. In the **Phrase Suggestions** field, choose **All, Auto Suggestions,** or **Business Phrases.** Phrase suggestions are for verbal rules. You can choose to see auto suggestions only, business phrases only, or both.

   For more information about business phrases, see Introduction to Verbal Rules and Business Phrases.

Figure 2-5 Dictionary Settings, Preferences Tab



How to Edit the Data Model tab

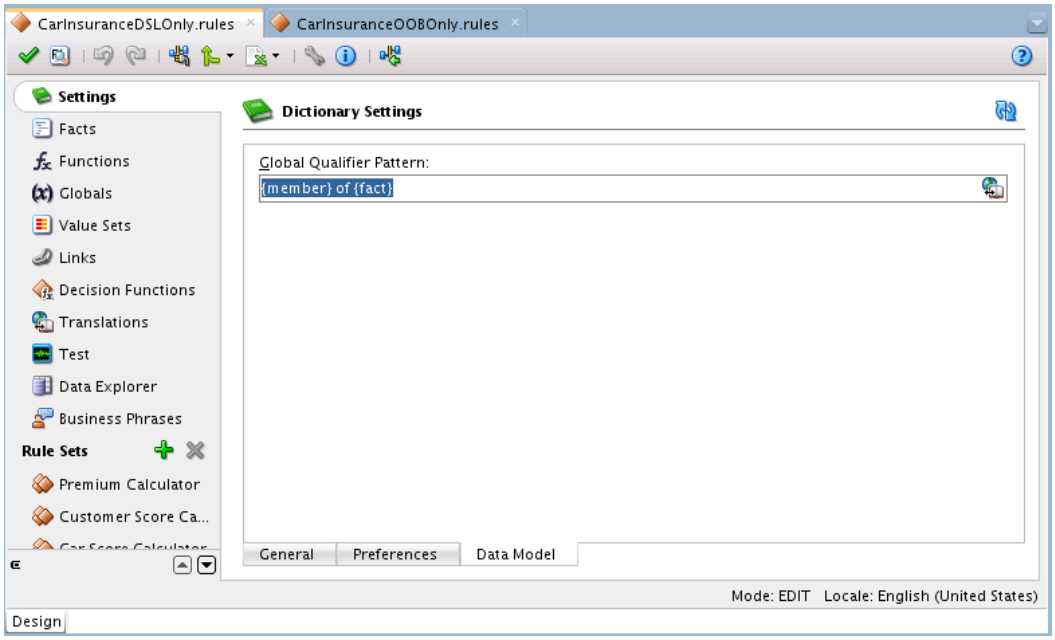
To edit the Data Model tab:

1. In the Global Qualifier Pattern field, shown in Figure 2-6, click the **Bundle Editor** button to update the resource bundles for this translatable value.

   You can specify the global qualifier pattern here. This is used in verbal rules. The pattern must contain two fragments: {member} and {fact}. The fragments {member} as well as {fact} are mandatory.

2. Click **Close** when done.

Figure 2-6 Dictionary Settings, Data Model Tab



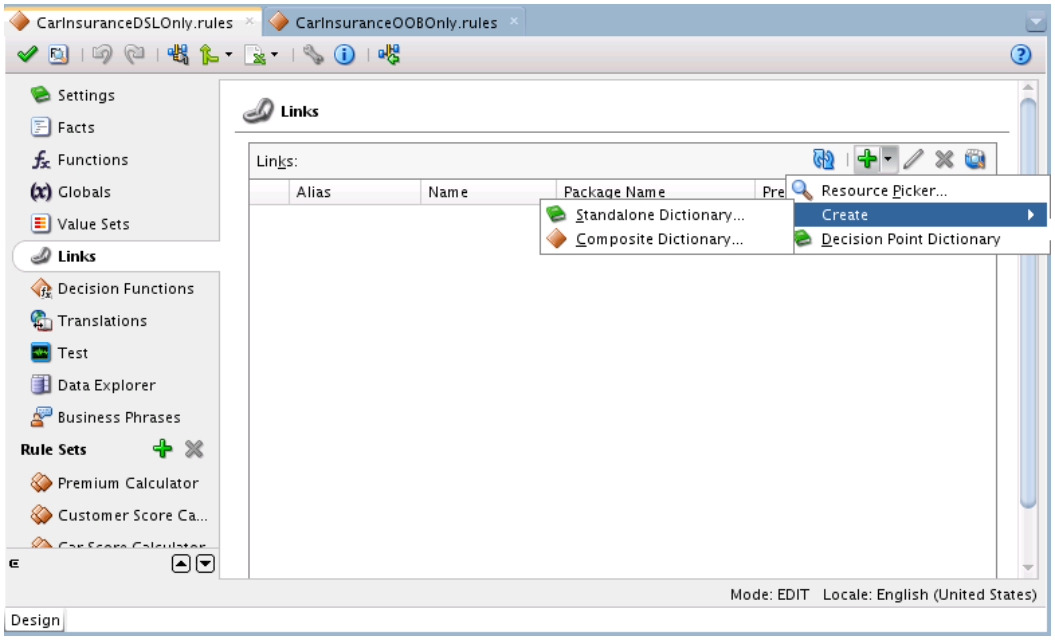
How to Link to a Dictionary

You can link to a dictionary in the same application using the **Links** navigation tab in Rules Designer. To link to another dictionary you need at least one other dictionary available.

To link to a dictionary using resource picker:

1. In Rules Designer, click the **Links** navigation tab as shown in Figure 2-7.

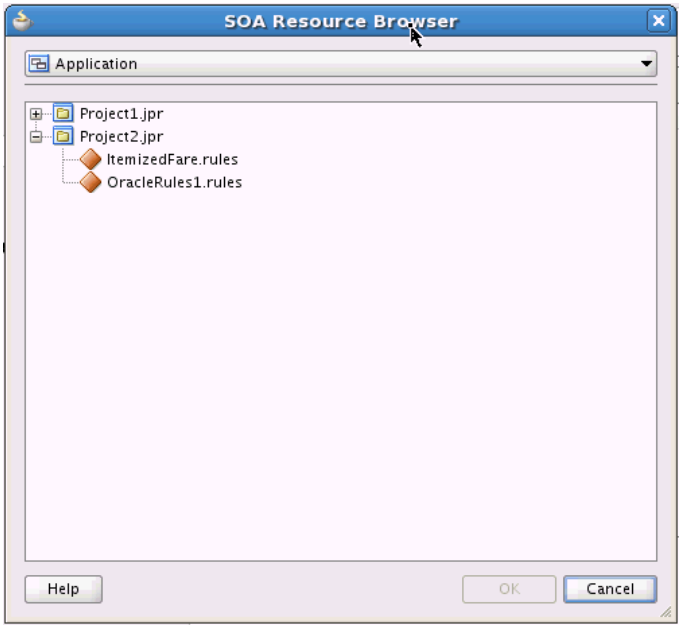
Figure 2-7 Rules Designer Links Tab



2. In the **Links** area, click the **Create** button and from the list select **Browse Existing Dictionaries**. This displays the SOA Resource Browser dialog.

3. In the SOA Resource Browser dialog navigate to select the dictionary you want to link to as shown in Figure 2-8.

Figure 2-8 Resource Picker



4. Click **OK**.

When you work with ADF Business Components Facts you should create a link to the Decision Point Dictionary. For more information, see Working with Oracle Business Rules and ADF Business Components.

In order to link the decision point dictionary, click the **Links** navigation tab in Rules Designer. In the **Links** area, click **Create** and from the list select **Decision Point Dictionary**. This operation takes awhile. You need to wait for the Decision Point Dictionary to load.

How to Update a Linked Dictionary

When you have a dictionary, for example, Project_rules1 that links to another dictionary, for example, Shared_rules you need to see changes made to either dictionaries in both. For example, you can modify the Shared_rules dictionary and see those modifications in Project_rules1 by updating the Project_rules1 dictionary, or by closing and reopening the Rules Designer. Note that you can only see the changes in the linked dictionary from the dictionary which defines the link and not vice versa.

To update a linked dictionary:

1. Using these sample dictionary names click the **Save** button to save the Shared_rules dictionary.

2. Select the Project_rules1 dictionary.

3. Select the **Links** navigation tab.

4. Click the **Dictionary Cache...** button.
5. In the Dictionary Finder Cache dialog, select the appropriate linked dictionary.
6. Click the **Clear** button.
7. In the Dictionary Finder Cache dialog, click **Close**.
8. Click the **Validate** button.

What You Need to Know About Dictionary Linking

Using a dictionary with links to another dictionary is useful in the following cases:

- **Data Model Sharing**, to share portions of a data model within a project. When you link to a dictionary in another project it is copied to the local project.

  For example, consider a project where you would like to share some Oracle Business Rules Functions. You can create a dictionary that contains the functions, and name it `DictCommon`. Then, you can create two dictionaries, `DictApp1` and `DictApp2` that both link to `DictCommon`, and both can use the same Oracle Business Rules functions. When you want to change one of the functions, you only change the version in `DictCommon`. Then, both dictionaries use the updated function the next time RL Language is generated from either `DictApp1` or `DictApp2`.

In Oracle Business Rules a fully qualified dictionary name is called a DictionaryFQN and this consists of two components:

- **Dictionary Package**: The package name
- **Dictionary Name**: The dictionary name

A dictionary refers to a linked dictionary using its DictionaryFQN and an alias. Oracle Business Rules uses the DictionaryFQN to find a linked dictionary.

The following are the naming constraints for combined dictionaries:

- The full names of the dictionaries, including the package and name, must be distinct. In addition, the dictionary aliases must be distinct.
- The aliases of data model definitions of a particular kind, for example, function, Oracle RL class, or value set, must be unique within a dictionary.
- A definition may be qualified by the alias of its immediately containing dictionary. Definitions in the top and built-in dictionaries do not have to be qualified. Definitions in other dictionaries must be qualified and this qualification is controlled by the **prefix linked names** property of the dictionary link.
- Ruleset names must be unique within a dictionary. When RL Language for a ruleset is generated, the dictionary alias is not part of any generated name. For example, if the dictionary named dict1 links to dict2 to create a combined dictionary, and dict1 contains ruleset_1 with rule_1 and dict2 also contains ruleset_1 with rule_2, then in the combined dictionary both of these rules, rule_1 and rule_2 are in the same ruleset (ruleset_1).
- All rules and Decision Tables must have unique names within a ruleset.

  For example, within a combined dictionary that includes dictionary d1 and dictionary d2, dictionary d1 may have a ruleset named `Ruleset_1` with a rule

rule_1. If dictionary d2 also has a ruleset named `Ruleset_1` with a rule_2, then when Oracle Business Rules generates RL Language from the combined, linked dictionaries, both rules rule_1 and rule_2 are in the single ruleset named `Ruleset_1`. If you violate this naming convention and do not use distinct names for the rules within a ruleset in a combined dictionary, Rules Designer reports a validation warning similar to the following:

```
RUL-05920: Rule Set Ruleset_1 has two Rules with name rule_1
```

For more information, see Oracle Business Rules Files and Limitations.

What You Need to Know About Dictionary Linking and Dictionary Copies

When you create a dictionary link using the resource picker, the dictionary is copied to the source project (the project where the dictionary that you are linking from resides). Thus, this type of linking creates a local copy of the dictionary in the project. This is not a link to the original target, no matter where the target dictionary is. Thus, Rules Designer uses a copy operation for the link if you create a link with the resource picker.

Also note the following regarding linked dictionaries in SOA and non-SOA rule dictionaries:

- SOA Applications
  1. Only dictionaries from within the same project, system dictionaries seeded in soa/shared or dictionaries available in the classpath can be used as linked dictionaries.
  2. If the same linked dictionary needs to be used across rules in multiple composites, then the linked dictionary should be referenced via the classpath.
- Non_SOA Applications
  1. Linked dictionaries can be located in the same application, in a shared location within MDS or the classpath. Appropriate dictionary finders need to be provided to locate and resolve the dictionaries.

What You Need to Know About Dictionary Linking to a Deployed Dictionary

When you are using Rules Designer you can browse a deployed composite application and any associated Oracle Business Rules dictionaries in the MDS connection. However, you cannot create a dictionary link to a dictionary deployed to MDS.

What You Need to Know About Business Rules Inputs and Outputs with BPEL

Decision function inputs are available as variables to the initial actions of the decision function. When the inputs are facts, the facts are asserted into working memory and rules must match the facts based on type and property values and not on decision function input name. For example, if you have inputs of same type, *input1* and *input2*, rules distinguish these inputs based on type or property values and not on the different names they have.When the inputs are not visible facts, for example String or int, then a wrapper type named <*decision function name*> is created, and rules must match this type.

How to Compare or Merge Two or More Dictionaries

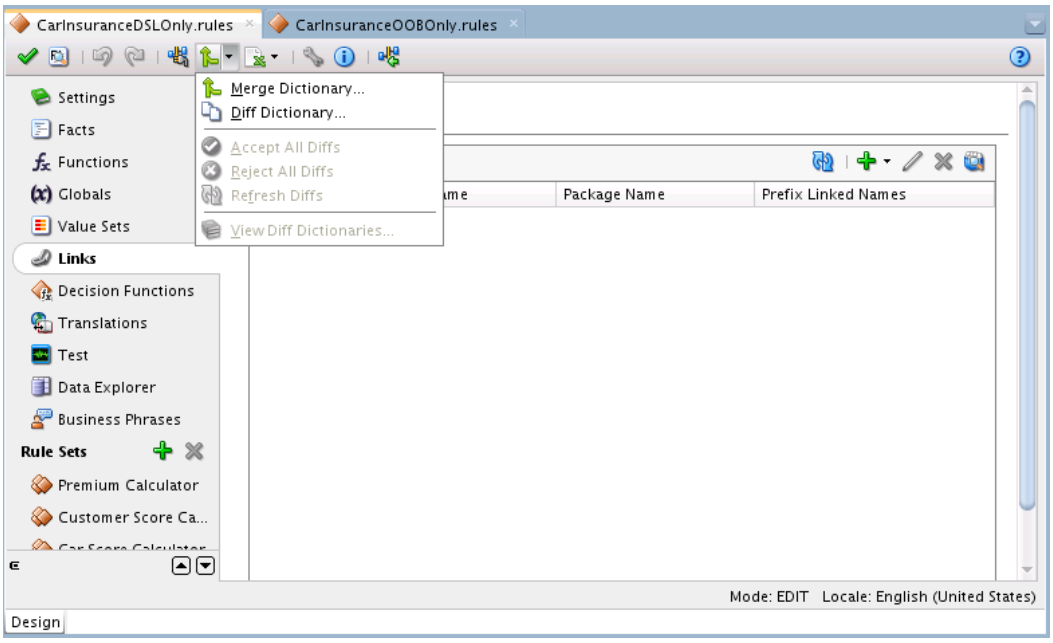
The Diff Dictionary feature enables you to review any differences in the latest revision of a dictionary against any previous revision and be able to roll back any changes since then. The differences are viewed from the perspective of the latest revision.

The Merge Dictionary feature enables you to review any differences between the base version and up to 3 changed versions and be able to resolve or merge the differences among them. The differences are viewed from the perspective of the changed versions.

Both Diff Dictionary and Merge Dictionary allow you to view and resolve the differences, but the basic difference between the two features is that you can Merge more than one dictionaries but you can not Diff more than one dictionaries.

The **Merge Dictionary** and **Diff Dictionary** options are available in the Rules Designer toolbar, as shown in Figure 2-9.

Figure 2-9 Diff-Merge Dictionary Button



You can compare up to three different dictionaries and merge into a fourth at design-time in Oracle JDeveloper. At runtime, you can use SOA Composer to do limited comparisons. For more information, see Using Oracle SOA Composer with Oracle Business Rules at Runtime.

In Rules Designer, you can compare a base version (which you must be editing) with two independently changed versions (relative to the base), and then merge selected changes into the base version (which must be saved as a new version).

Warning:

Before you decide to run either of these features, you must be ready resolve all changes because the dictionary becomes read-only when in diff or merge mode.

Merging dictionaries should be done with care. You must identify and manage the different versions involved (base, version 1, version 2, and the results).

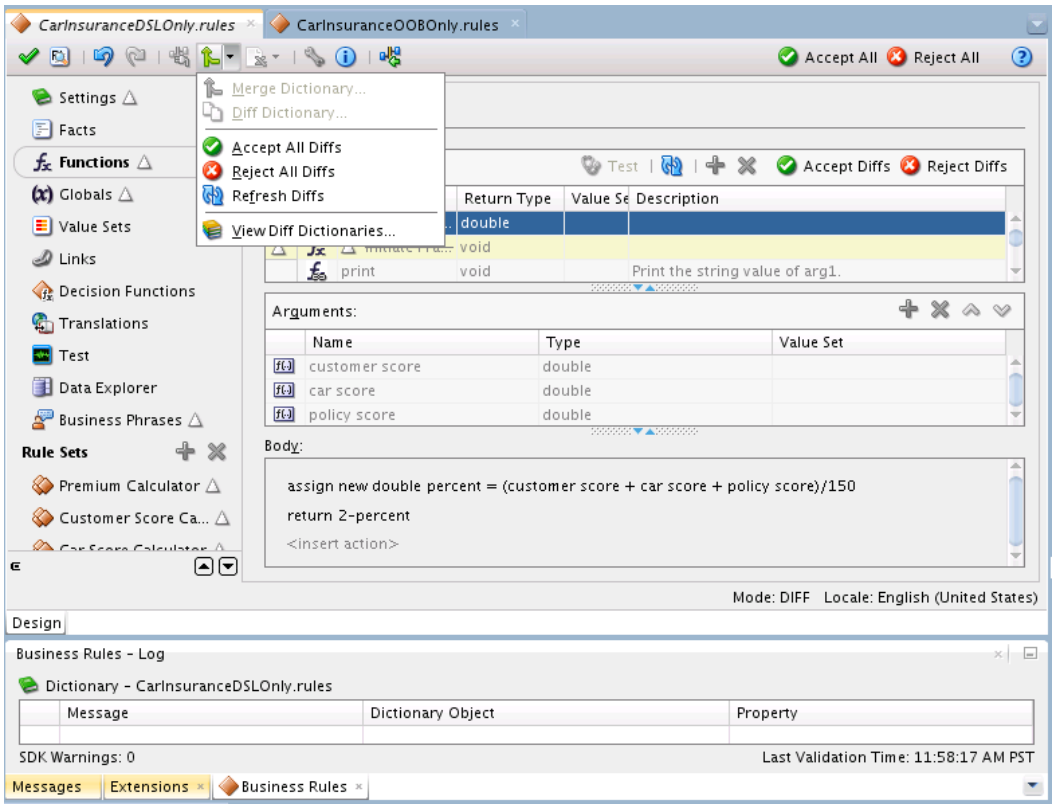
How to See Differences Between Dictionaries

When you want to compare dictionaries, you open the newer dictionary first in the Rules Designer, then use the Diff Dictionary dialog to select the older dictionary to compare with. Anything missing from the newer dictionary is flagged as a deletion from the newer version.

To see the differences between dictionaries:

1. In the Rules Designer, with the newer dictionary open, click **Diff Dictionary.**
2. In the **Diff Dictionary** dialog, click **Browse** to open the **Select Dictionary to Merge** dialog and find the dictionary that you want to compare with.
3. Click **OK**.
4. Enter a short version name or number.
5. Click **OK** when done.
6. All differences between the two dictionaries will be flagged with change icons, as shown in Figure 2-10.

Figure 2-10 Diff Changes Displayed



The change icons are shown for all tabs on the left, and for the specific artifacts within each tab.

7. Click each tab and decide to **Accept Diffs** or **Reject Diffs**. Alternatively, you can choose to **Accept All** or **Reject All** in the toolbar.

   Diffs can be Accepted or Rejected at any level in the dictionary by clicking on the appropriate change icon. For example, to revert Rule1 to the older version but keep everything else in the newer dictionary, first drill down to Rule1 and choose Reject Diffs, then chose Accept All from the toolbar. Note that in 'Diff mode', Accept keeps the newer version and Reject reverts to the older version.

You can view the Diff Dictionaries option. This choice is available after you have compared dictionaries. The View Diff Dictionaries option, shown in Figure 2-10 lists information about the dictionaries being compared.

How to Merge Dictionaries

When you want to merge dictionaries, you open the older dictionary first in the Rules Designer, then use the Merge Dictionary dialog to select the newer dictionary to merge with. Anything missing from the old dictionary is flagged as an addition in the latest version.

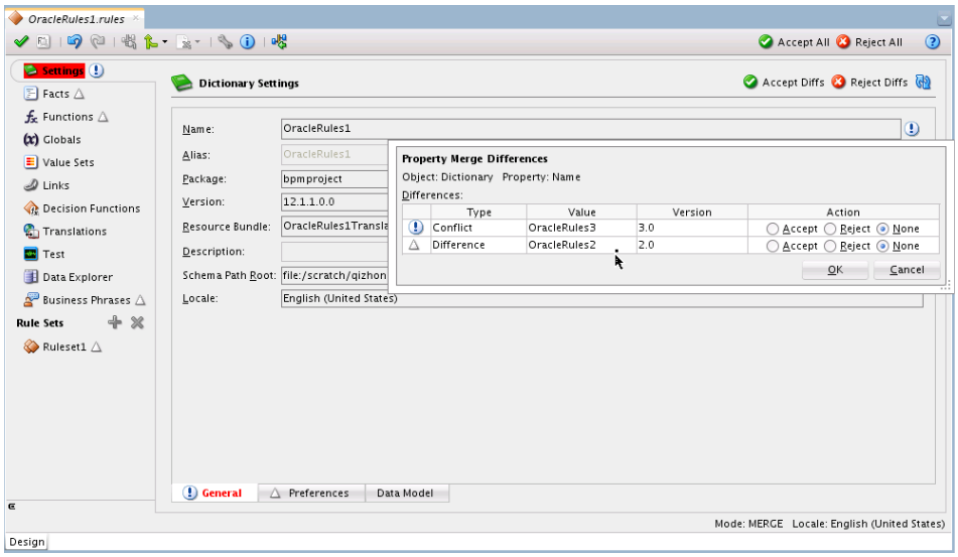
This works with two or more dictionaries, so you should use oldest, then one or more newer, and finally save the result in newest.

Use care when merging dictionaries. Because general editing is disabled until all diffs are resolved, you may want to provisionally accept or reject conflicting values and then return to finish the editing after handling remaining diffs.

To merge dictionaries:

1. In the Rules Designer, with the oldest dictionary open, click Merge Dictionary.
2. In the **Merge Dictionary** dialog, click **Browse** to open the **Select Dictionary to Merge** dialog and find the dictionary that you want to compare with.
3. Click **OK.**
4. Enter a short version name or number.
5. Click **OK** when done.
6. All changes are flagged in Rules Designer, as shown in Figure 2-11. Use the **Merge Differences** dialog to **Accept** or **Reject** or take no action. You can also use the **Accept Diffs** or **Reject Diffs** in the tab. Alternatively, you can **Accept All** or **Reject All** from the toolbar.

Figure 2-11 Merge Differences Dialog



Working with Oracle Business Rules Globals

You can use Rules Designer to add Oracle Business Rules globals.

In Oracle Business Rules a global is similar to a public static variable in Java. You can specify that a global is a constant or is modifiable.

You can use global definitions to share information among several rules and functions. For example, if a 10% discount is used in several rules you can create and use a global Gold Discount, so that the appropriate discount is applied to all the rules using the global.

Using global definitions can make programs modular and easier to maintain.

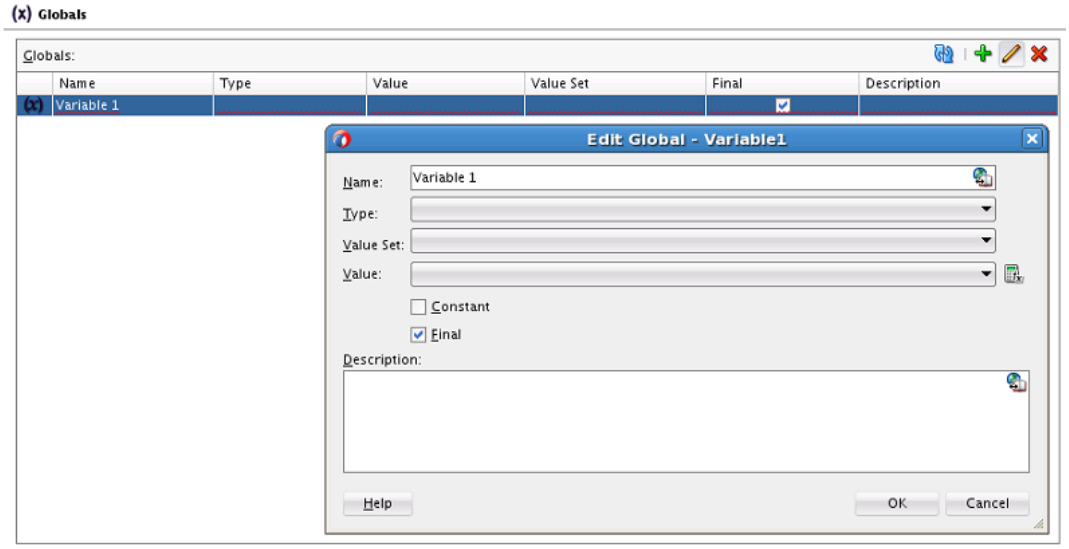
How to Add Oracle Business Rules Globals

You can use Rules Designer to add globals.

To add a global:

1. In Rules Designer, select the **Globals** navigation tab.
2. In the globals table, click the **Create** button. This adds a global and displays the Edit Global dialog, as shown in Figure 2-12.

Figure 2-12 Adding a Global in Rules Designer



3. In the **Name** field, enter a name or accept the default value.
4. In the **Type** field, select the type from the list.
5. Optionally, in the **Value Set** field, select a value from the list.
6. In the **Value** field, enter a value, select a value from the list, or click the **Expression Builder** button to enter an expression.

   For more information, see Introduction to Expression Builder.
7. If the global is a constant, then select the **Constant** check box. When selected, this option specifies that the global is a constant value.

   For more information, see What You Need to Know About the Final and Constant Options.
8. If the global is a nonfinal, then clear the **Final** check box. When cleared, this option specifies that the global is modifiable, for instance, in an assign action.

How to Edit Oracle Business Rules Globals

You can use Rules Designer to edit globals.

To edit a Global:

1. In Rules Designer, select the **Globals** navigation tab.
2. Click the Edit button to open the **Edit Global -** *Global Name* window. In this window you can edit a global and change field values, including the **Final** field and the **Constant** field.

What You Need to Know About the Final and Constant Options

The Edit Global dialog shows the **Constant** and **Final** check boxes that you can select for a global.

Note the following when you use globals:

- When you clear **Final**, this specifies that the global is modifiable, for instance, in an assign action.
- When you select **Final**, this specifies that you can use the globals in a test in a rule (nonfinal globals cannot be used in a test in a rule).
- When you select **Final**, this specifies that the global is initialized one time at runtime and cannot be changed.

When you select the **Constant** option in the Edit Global dialog, this specifies the global is a constant. In Oracle Business Rules a constant is a string or numeric literal, a final global whose value is a constant, or a simple expression involving constants and `+`, `-`, `*`, and `/`.

Selecting the **Constant** option for a global has three effects:

- You do not have to surround string literals with double quotes.
- Only constants appear in the expression value choice list.
- The expression value must be a constant to be valid.

Selecting the **Constant** option is optional. Note that Value Set values, Value Set range endpoints, and ruleset filter values are always constant.

Working with Decision Functions

The data model includes decision functions.

For information on working with decision functions, see Introduction to Decision Functions.

Introduction to Oracle Business Rules Functions

Oracle Business Rules provides functions to hide complexity when you create rules. Oracle Business Rules lets you use built-in or user-defined functions in rule and Decision Table conditions and actions.

In Oracle Business Rules you define a function in a manner similar to a Java method, but an Oracle Business Rules function does not belong to a class. You can use Oracle Business Rules functions to extend a Java application object model so that users can perform operations in rules without modifying the original Java application code.

You can use an Oracle Business Rules function in a condition or in an action associated with a rule or a Decision Table.

You can also use an Oracle Business Rules function definition to share the same or a similar expression among several rules, and to return results to the application.

An Oracle Business Rules function includes the following:

- Name: The Oracle Business Rules function name.

- Return Type: A return type for the Oracle Business Rules function, or void if there is no return value.
- Value Set: The value set to associate with the Oracle Business Rules function. This is optional.
- Arguments: The function arguments. Each function argument includes a name and a type and an optional value set.
- Function Body: The function body includes predefined actions. Using predefined actions Rules Designer assures that an Oracle Business Rules function is well formed and can be validated.

You can also use functions to test rules from within Rules Designer. For more information, see Testing Decision Functions Using a Rules Function.

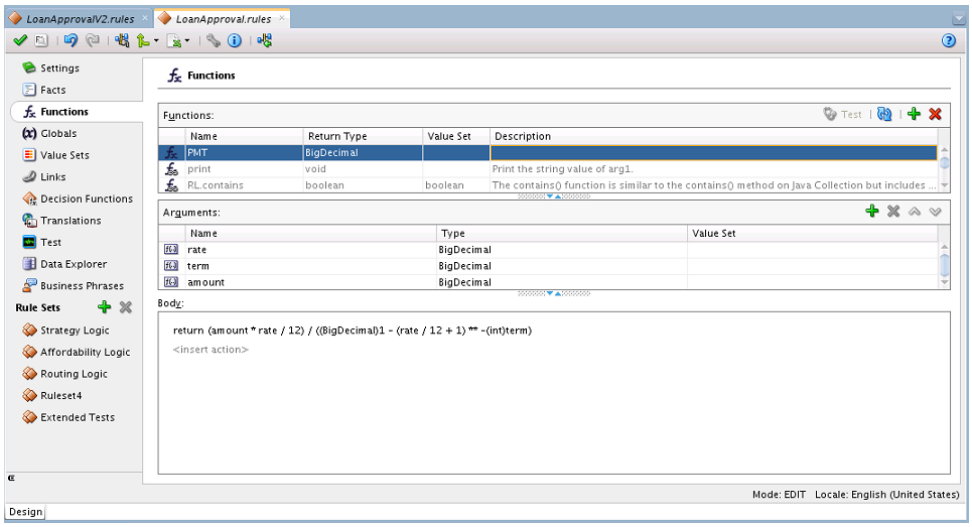
How to Add an Oracle Business Rules Function

You use Rules Designer to add an Oracle Business Rules function.

To add an Oracle Business Rules Function:

1. In Rules Designer, select the **Functions** navigation tab.
2. Select the **Create...** button.
3. Enter the function name in the **Name** field, or use the default name.
4. Select the return type from the **Return Type** list. For example, select `void`.
5. Optionally, select a value set to associate with the function return type from the list in the **Value Set** field.
6. Optionally, in the **Description** field enter a description.
7. In the Arguments table, click **Add** to add one or more arguments for the function.
8. For each argument in the **Type** field, select the type from the list.
9. For each argument in the **Value Set** field, to limit the argument values as specified by a value set constraint, select a value set from the list.
10. In the **Body** area, enter actions and arguments for the function body. You can add any required action ranging from `assert`, `call`, `modify` to even conditional actions such as `if`, `else`, `elseif`, `while`, `for`, `if (advanced)`, and `while (advanced)`. For example, see Figure 2-13.

Figure 2-13 Adding an Oracle Business Rules Function



Localizing Oracle Business Rule Resources

You can localize the names, aliases and descriptions of rules resources. This enables better control of these resources in Workspace and SOA Composer. You can localize most of the resources like Value Sets, Globals, Rulesets, Rules and so on. With Verbal Rules, you can also localize the value of Business Phrases.

When you create these resources, you can add locale-specific information from the Translations tab. Each locale is stored in a separate resource bundle.

Note:

You should not manually edit the resource bundle to add or edit localized strings. You must edit the resource bundle using the Translation tab of the Rules Designer in JDeveloper, BP Composer, or SOA Composer.

How to Localize the Resources in Oracle Business Rules

You can use the Rules Designer of JDeveloper to localize the resources of a business rule.

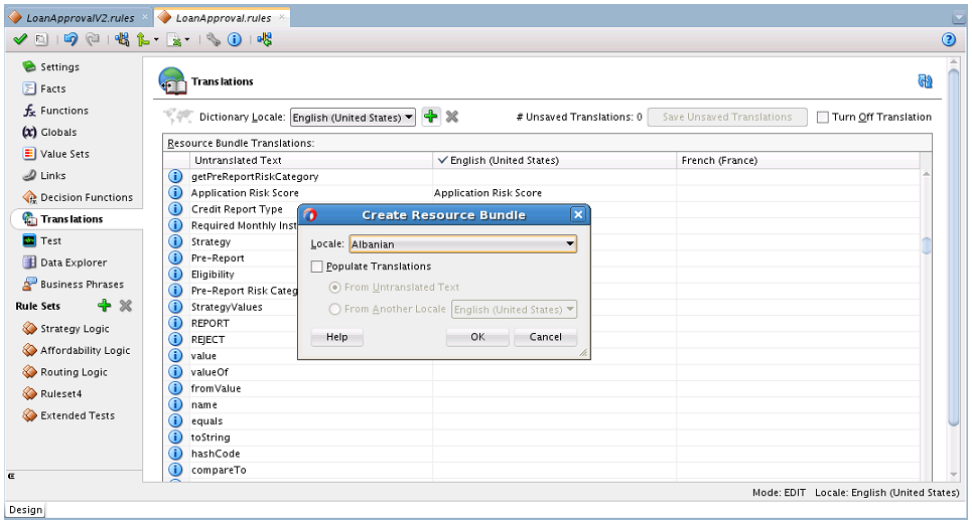
To localize business rule resources:

1. In Rules Designer, select the **Translations** tab.
2. Click the **Create Resource Bundle** button.

   Create **Resource Bundle** screen appears.
3. Select the **Locale** from the list, as shown in Figure 2-14.

   Each locale that you add appears as a column in the Resource Bundle Translations table. Each resource of the business rule appears as a row in this table. Each locale is stored as a separate resource bundle.

Figure 2-14 Adding New Locales



4. Click the cell of the table corresponding to the resource and locale and enter the localized text.

Note:

The translated value is validated only in the current locale. Validations are not done for translations in other locales that are not used.

---

5. Select **Populate Translations** and a radio button to populate the translation of the new bundle from untranslated text or from another locale.

6. Click **OK**.

3

Working with Facts and Value Sets

This chapter describes the Oracle Business Rules data model element called fact types, which are the objects that rules reason on. It also covers another element called value sets that define groupings of fact property values.

The chapter includes the following sections:

Introduction to Working with Facts and Value Sets

In Rules Designer, you make business objects and their methods known to Oracle Business Rules using fact types that are part of a data model. A fact type is a type definition in the data model. A fact is an instance of that fact type and is a data structure that rules can operate on.

For example, a fact type is a collection of related properties (business terms), and a fact is therefore a collection of related data bound to the business terms. A customer fact may include not only name, but address, history, credit rating, and so forth.

You can create fact types and value sets before you create rules.

In Rules Designer you can work with the following kinds of facts:

- **XML Facts**: XML Facts are imported from existing sources by specifying XML Schema. You can add aliases to imported XML Facts or use XML Facts with RL Facts to change the data model according to your business needs.

  For more information, see Working with XML Facts.

- **Java Facts**: Java Facts are imported from existing sources. You can add aliases to Java Facts or use them with RL Facts to target the data model to business needs. Java Facts are also used to import supporting Java classes for use with the rules or Decision Tables that you create.

  For more information, see Working with Java Facts.

- **RL Facts**: RL Facts are the only kind of facts that you can create directly and do not have an external source. All other types of Oracle Business Rules facts are

imported. An RL Fact is similar to a relational database row or a JavaBean with properties. An RL Fact contains a set of named, typed properties. Property values can be primitives such as String, another structured fact, or a list. RL Facts are useful for rapid and independent development and testing of decision logic. Input data that will ultimately come from an imported fact type (for example, an XML Schema) can be modeled using RL Facts before the imported schema is available or stable. Intermediate decisions that should not be returned to the application (for example, sub-decisions that categorize a customer as GOOD or BAD). It is usually best to import the fact types that are used for the input and output data of a decision. You can use RL Facts to extend a Java application object model by providing virtual dynamic types.

For more information, see Working with RL Facts.

- **ADF Business Components Facts**: ADF Business Components Facts allow you to use ADF Business Components as Facts in rules and in Decision Tables. By using ADF Business Components Facts you can assert view object graphs representing the business objects upon which rules should be based, and let Oracle Business Rules deal with the complexities of managing the relationships between the various related view objects in the view object graph.

  For more information, see Working with ADF Business Components Facts.

You typically use Java fact types and XML fact types to create rules that examine the business objects in a rule-enabled application, or to return results to the application. You use RL Language fact type definitions to create intermediate facts that can trigger other rules in the Rules Engine. ADF Business Components fact types enables you to use ADF Business Components as Facts in rules and in Decision Tables.

In Oracle Business Rules, facts that you can run against the rules are data objects that have been asserted. Each object instance corresponds to a single fact. If an object is re-asserted (whether it has been changed or not), the Rules Engine is updated to reflect the new state of the object. Re-asserting the object does not create a fact. To have multiple facts of a particular fact type, separate object instances must be asserted.

You can create value sets to define a list of values or a range of values of a specified type. After you create a value set, you can associate the value set with a business term of matching type. When a value set is associated with a business term, Oracle Business Rules uses the values or ranges that you define as constraints for the values for the business terms for the business terms in rules that are in the Decision Tables.

For more information, see:

- Working with Value Sets
- Associating a Value Set with Business Terms

Working with XML Facts

The XML fact type allows XML Schema types, elements, and attributes to be used when writing rules. Elements and types defined in XML Schema can be imported into the data model and can then be used to create IF/THEN rules and Decision Table rules, just as with Java fact types and RL Fact types. The mapping between the XML Schema definition and the XML Fact types uses the Java Architecture for XML Binding (JAXB).

By default, Oracle Business Rules uses the JAXB 2.0 shipped with the Oracle Application Server. JAXB as defined in JSR-222 provides a mapping between the types, names, and conventions in an XML Schema definition and the available types, allowed names and conventions in Java. For example, an element named `order-id`

and of type `xsd:integer` is mapped to a Java Bean property named `orderID` of type `BigInteger` (and `xsd:int` type maps to Java `int`).

Thus, with Oracle Business Rules if you have an XML document that contains data associated with your application and you have the schema associated with the XML document then you can use Rules Designer to define rules based on elements that you specify from the XML Schema.

Note: When xsd has primitive or non primitive root elements of simple type, JAXB maps the simple type elements to a JAXBElement and generates only ObjectFactory class. You must create Facts of complex type to be used in business rules.

How to Create XML fact types

1. Define or obtain an XML Schema.
2. Use Rules Designer to import the XML Schema into a dictionary.

   This step uses the JAXB compiler to generate Java classes from the XML Schema. After you compile the XML Schema, you select the desired elements from the schema to add XML Facts in the data model and import the generated JAXB classes into the data model.

   For more information on these steps, see How to Import the XML Schema and Add XML Facts.
3. Write rules or create Decision Tables based on these XML Facts that you added to the data model.

   For more information, see Working with Rules and Creating Decision Tables.

Elements and types defined in the XML Schema can be imported into the data model so that instances of types can be created, asserted, modified, and retracted by rules. Most XML documents describe hierarchical information, where each element contains subelements. It is common for users to want to write individual rules based on multiple elements in this hierarchy, and the hierarchical relationship among the elements.

In Oracle Business Rules the default behavior when you `assert` a fact is to only assert the single fact instance, and none of the child objects it may reference in the hierarchy of subelements. When you create rules or a Decision Table it is often desirable to assert an entire hierarchy of elements based on a reference to a root element. Oracle Business Rules provides the `assertTree` action type that allows for a recursive assert for a hierarchy. For more information, see Working with Tree Mode Rules.

How to Import the XML Schema and Add XML Facts

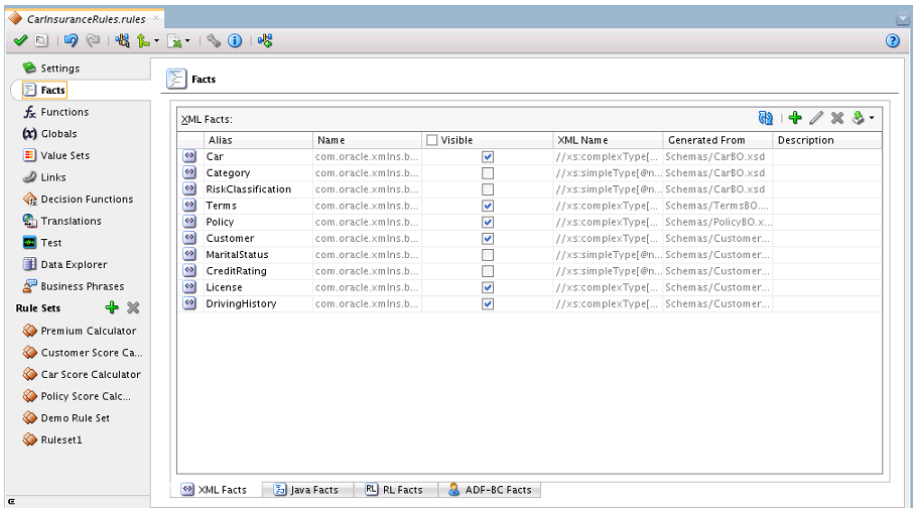
Before you can use the XML Schema definitions in a data model you must import the XML schema. This step generates the JAXB classes and makes the generated classes and packages associated with the XML schema visible in Rules Designer.

To import XML schema and add XML facts:

1. In Rules Designer, select the **Facts** navigation tab.

2. Select the **XML Facts** tab on the **Facts** navigation tab, as shown in Figure 3-1.

Figure 3-1 The XML Facts Tab in Rules Designer



3. In the **XML Facts** tab, click **Create...**. This displays the Create XML Fact dialog.

4. In the Create XML Fact dialog, in the **Source Schemas** area, click **Add Source Schema...** to open the Add Source Schema dialog.

5. In the Add Source Schema dialog,

   - Enter the location of the XML Schema you want to import, or click **Browse** to locate the XML schema in the **Schema Location** field. During the import the file is copied into the project.

     Note:

     Typically, the XML schema (xsd) file is located inside the xsd folder called Schemas, because any XML schema that is created needs to be stored inside the xsd folder under SOA.

   - Accept the default path or enter the directory where you want Rules Designer to store the JAXB-generated Java source and class files in the **JAXB Classes Directory** field.

   - Enter a target package name or leave this field empty in the **Target Package** field. If you leave this field empty the JAXB classes package name is generated from the XML target namespace of the XML schema using the default JAXB XML-to-Java mapping rule or explicitly defined package name using annotations, or "`generated`" if no namespace or annotation is defined. Using the schema namespace is preferred.

     For example, the namespace `http://www.oracle.com/as11/rules/demo` is mapped to `com.oracle.as11.rules.demo`.

   - Click **OK**.

     Rules Designer processes the schema and compiles the JAXB, so depending on the size of the schema this step may take some time to complete. When this step

completes Rules Designer displays the Create XML Fact dialog with the **Target Classes** area updated to include the JAXB classes.

6. In the Create XML Fact dialog, in the **Target Classes** area, select the classes you want to import as XML fact types.
7. Click **OK**.

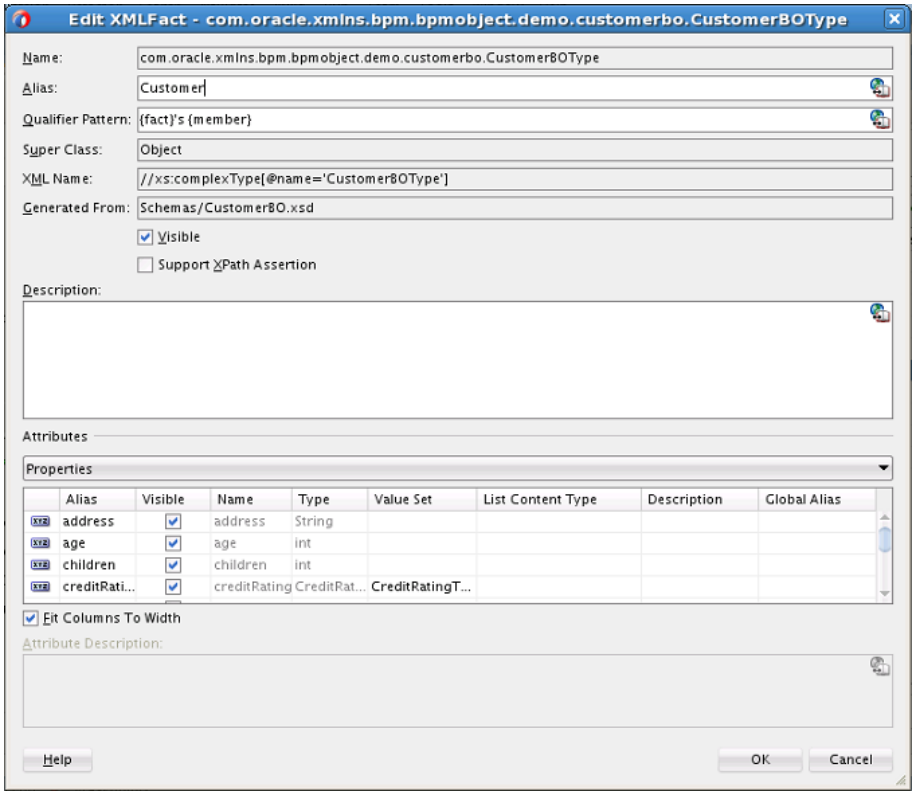
How to Display and Edit XML Facts

To work with an XML Fact, in Rules Designer open the Edit XML Fact dialog.

To display and edit XML facts:

1. In Rules Designer, select the **Facts** navigation tab.
2. Select the **XML Facts** tab on the **Facts** navigation tab.
3. In the XML Facts table, double-click the icon for the XML Fact you want to edit. This displays the Edit XML Fact dialog, as shown in Figure 3-2.

Figure 3-2 Edit XML Fact Dialog



The Edit XML Fact dialog includes the fields shown in Table 3-1.

Table 3-1 XML Fact: Edit XML Fact Dialog Fields

Field	Description
Name	Displays the XML Fact name. You cannot change the name of JAXB generated class.

Table 3-1 (Cont.) XML Fact: Edit XML Fact Dialog Fields

Field	Description
Alias	Enter the XML Fact alias. You can change this value. This defaults to the unqualified name of the class.
Qualifier Pattern	This field is used for verbal rules. If nothing is specified here, then the system uses the global qualifier. If a custom qualifier pattern should be specified for a fact, it has to contain two parts in the pattern: {member}, {fact}. Specify patterns as follows: • {fact}'s {member} • {member} in {fact} • {member} of {fact} (This is the default). For more information about using verbal rules, see Introduction to Verbal Rules and Business Phrases.
Super Class	Displays Java super class associated with this fact.
XML Name	Displays the XML name associated with the XML Fact.
Generated From	Displays the XML schema file that was the source for the XML Fact when it was copied into the business rules data model.
Visible	Select to show the XML Fact in lists in Rules Designer. XML Facts often reference other XML Facts, forming a tree. You should make all the XML fact types visible that contain properties that you reference in rules.
Support XPath Assertion	Select to enable XPath assertion for the fact. This feature is provided for backward compatibility only. Typically, this option is not used.
Description	Enter the XML Fact description.
Attributes area	Select the available constructors, properties, methods, or fields associated with the JAXB class for the XML Fact to display or edit.
Fit Columns to Width	Select this check box to adjust column width.

How to Reload XML Facts with Updated Schema

If an XML schema changes in a project, the schema must be reimported into the Oracle Business Rules dictionary. When you reimport the schema, Oracle Business Rules uses JAXB to recompile all source schemas for every XML fact type and updates the XML fact type definitions with the updated XML schema definitions. You should reimport facts if you changed the schema or classes and you want to use the changed schema or classes at runtime.

Note:

When the XML schema on which an XML fact is based changes, on reimporting the schema, the facts are updated and imported into the base dictionary. When working with facts in a linked dictionary, you need to reload the XML facts for the changed schema from the base dictionary instead of the linked dictionary.

To reimport XML facts:

1. In Rules Designer, select the **Facts** navigation tab.
2. Select the **XML Facts** tab on the **Facts** navigation tab.
3. On the XML Facts page, click **Reload Facts Based on Modified Schemas (Excluding Linked Facts)** or **Reload All Facts (Excluding Linked Facts)**.

After the reimport operation you need to correct any validation warnings that may be caused by incompatible changes (for example, the updated schema may include a change that removed a property that is referenced by a rule).

What You Need to Know About XML Facts

Keep the following points in mind when you work with XML Facts:

When XML Schema contain a `restriction` definition, this allows a user to restrict the types that are valid for use in an element. A common use of restriction is to define an enumeration of strings which can be used for an element, as shown in the XML Schema Restriction example below:

- ```
  <xs:simpleType name="status-type">
          <xs:restriction base="xs:string">
              <xs:enumeration value="manual"/>
              <xs:enumeration value="approved"/>
              <xs:enumeration value="rejected"/>
          </xs:restriction>
  </xs:simpleType>
  ```

  Oracle JAXB 2.0 maps a restriction to a Java enum type. When you use Rules Designer to import either a Java enum type or an element with an XML restriction, the static final fields representing the enums are available for use in expressions. Additionally, Oracle Business Rules creates a value set for each enum containing all of the enum values and null. For more information on value sets, see Working with Value Sets.

- There is a default version of the JAXB binding compiler supplied with Oracle Application Server. You can use a different JAXB binding compiler. However, to use a different JAXB binding compiler you must manually perform the XML schema processing and then import the generated Java packages and classes into the data model as Java Facts.

  For more information about JAXB, see

  `http://java.sun.com/webservices/jaxb/`

- You should reimport facts if you changed the schema or classes and you want to use the changed schema or classes at runtime. You should correct any validation warnings that may be caused by incompatible changes (for example, removing a
  ```

property that is referenced by a rule). For more information, see How to Reload XML Facts with Updated Schema.

- Most users should not need to use the ObjectFactory or import it. If you do need to import and use the ObjectFactory, then use a different package name for every XML Schema that you import; otherwise the different ObjectFactory classes conflict.
- The use of XML schema with elements that have `minOccurs="0"` and `nillable="true"` has special handling in JAXB. For more information, see Why do XML Schema with xsd:string Typed Elements Import as Type JAXBElement?.
- The default element naming conventions for JAXB can cause XML schema containing the underscore character in XML-schema element names to fail to compile. For more information, see Why Does XML Schema with Underscores Fail JAXB Compilation?.
- There are certain restrictions on the types and names of inputs for the Decision Service. For more information, see How Are Decision Service Input Output Element Types Restricted?.
- The built-in dictionary includes support for the Java wrappers `Integer`, `Long`, `Short`, `Float`, `Double`, `BigDecimal`, and `BigInteger`. These types can appear in XML Fact Types.

Working with Java Facts

In Rules Designer, importing a Java Fact makes the Java classes and their methods become visible to Rules Designer. Rules Designer does not copy the Java code or bytecode into the data model or into the dictionary.

A Java fact type allows selected properties and methods of a Java class to be imported to the Rules Engine so that rules can access, create, modify, and delete instances of the Java class.

Importing a Java fact type allows the Rules Engine to access and use public attributes, public methods, and bean properties defined in a Java class (bean properties are preferable because they can be modified using the modify action).

How to Import Java Classes and Define Java Facts

Before you can use Java Facts in rules and in Decision Tables, you must make the classes and packages that contain the Java Facts available to Rules Designer. To do this you use Rules Designer to specify the classpath that contains the Java classes, and then you import the Java Facts.

Java fact types allow methods with and without side effects to be imported. Side effects refer to expensive operations such as I/O, Database Query or web service and so on. When using Java classes as facts, remember the following about side effects:

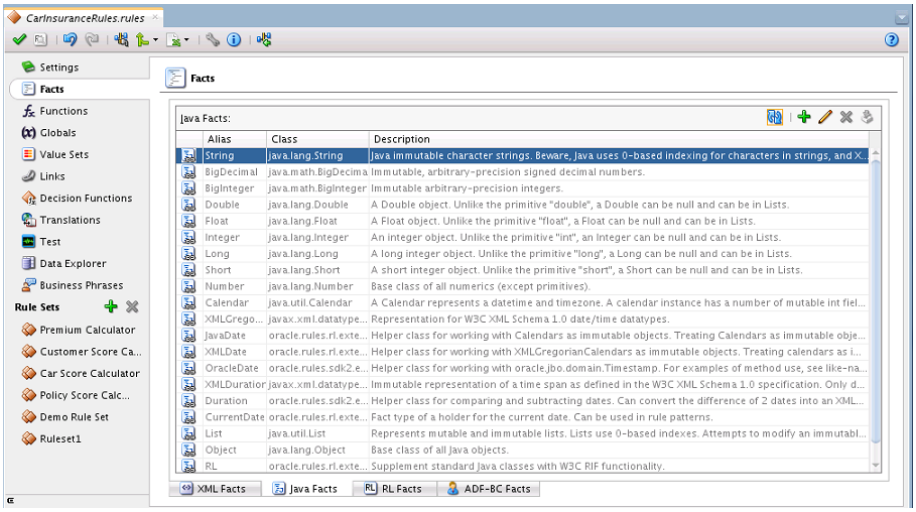
- Rule and Decision Table conditions do not use methods with side effects.
- Actions can use all methods. Though side effects are not recommended, are permissible.

To import and define Java Facts:

1. In Rules Designer, select the **Facts** navigation tab.

2. Select the **Java Facts** tab on the **Facts** navigation tab as shown in Figure 3-3.

Figure 3-3 The Java Facts Table in the Facts Navigation Tab



3. In the **Java Facts** tab, click **Create...** to open the Create Java Fact dialog.

4. In the Create Java Fact dialog, if the classpath that contains the classes you want to import is not shown in the **Classpath** area, then click **Add to Classpath**. This displays the Choose Directory/Jar dialog.

   The default Rules Designer classpath includes three packages, `java`, `javax`, and `org`. These packages contain classes that Rules Designer lets you import from the Java runtime library (rt.jar). Rules Designer does not let you remove these classes from the **Classes** area (and the associated classpaths are not shown in the **Classpaths** area).

5. In the Choose Directory/Jar dialog, browse to select the classpath or jar file to add. By default, the output directory for the project is on the import classpath and any Java classes in the project should appear in the Classes importer. If they do not appear, execute the Build action for the project.

6. Click **Open**. This adds the classpath or jar file and updates the **Classes** area.

7. In the Create Java Fact dialog, in the **Classes** area select the packages and classes to import.

8. Click **OK**. This updates the Java Facts table in the **Java Facts** tab.

How to Display and Edit Java Facts

To display or edit Java Facts after you import the Java Facts, use the Edit Java Fact dialog.

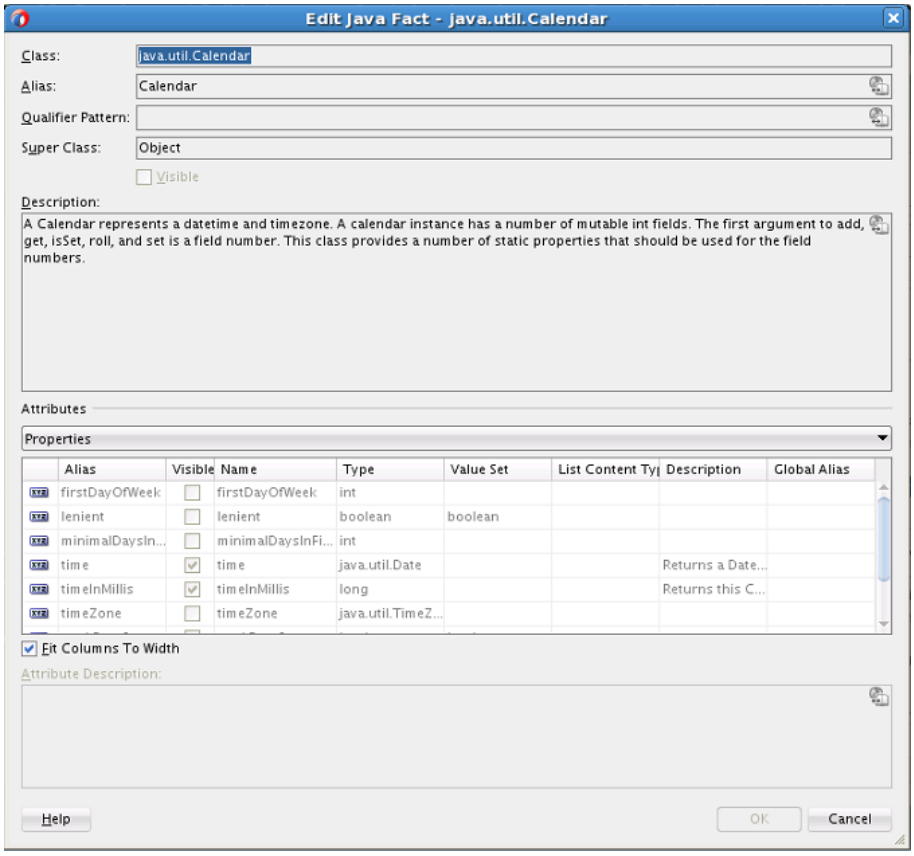
To display and edit Java facts:

1. In Rules Designer, click the **Facts** navigation tab.

2. Select the **Java Facts** tab in the **Facts** navigation tab.

3. In the Java Facts table, double-click the Java Fact you want to edit. This displays the Edit Java Fact dialog as shown in Figure 3-4.

Figure 3-4 Edit Java Fact Dialog



The Edit Java Fact dialog includes the fields shown in Table 3-2.

Table 3-2 Edit Java Fact Dialog Fields

Field	Description
Class	Displays the Java Fact class for the source associated with the Java Fact.
Alias	Enter the Java Fact alias.
Qualifier Pattern	This field is used for verbal rules. If nothing is specified here, then the system uses the global qualifier. If a custom qualifier pattern should be specified for a fact, it has to contain two parts in the pattern: {member}, {fact}. Specify patterns as follows: • {fact}'s {member} • {member} in {fact} • {member} of {fact} (This is the default). For more information about using verbal rules, see Introduction to Verbal Rules and Business Phrases.
Super Class	Displays Java super class associated with this fact.
Visible	Select to show the Java Fact in lists in Rules Designer.

Table 3-2 (Cont.) Edit Java Fact Dialog Fields

Field	Description
Description	Enter the Java Fact description.
Attributes area	Select the available class properties, constructors, methods, or fields associated with the Java class for the Java Fact act to display or edit.
Properties	Some java objects have fields to help define that object. For example a Calendar has properties for defining the first day of the week, the time zone, and so on.
Fit Columns to Width	Select this check box to adjust column width.
Attribute Description	The Attribute Description is the description of the property.

What You Need to Know About Java Facts

When you define Java Facts you need to know the following:

- On Windows systems, you can use a backslash (\) or a slash (/) to specify the classpath in the **Classpath** area. Rules Designer accepts either path separator.
- Classes and interfaces that you use in Rules Designer must adhere to the following rules: If you are using a class or interface, only its superclass or one of its implemented interfaces may be made visible.
- When you specify the classpath you can specify a JAR file, a ZIP file, or a full path for a directory.
- When you specify a directory name for the classpath, the directory specifies the classpath that ends with the directory that contains the "root" package (the first package in the full package name). Thus, if the classpath specifies a directory, Rules Designer looks in that tree for directory names matching the package name structure.

  For example, to import a class `cool.example.Test1` located in `c:\myprj\cool\example\Test1.class`, specify the classpath value, `c:\myprj`.
- You should reimport facts if you change the classes. After the reimport operation you may see validation warnings due to class changes. You should correct any validation warnings that may be caused by incompatible changes (for example, removing a property that is referenced by a rule).

Working with RL Facts

RL Facts are the only kind of facts that you can create directly and that do not have an external source. All other types of Oracle Business Rules facts are imported. An RL Fact is similar to a relational database row or a JavaBean without methods. An RL Fact contains a list of properties of types available in the data model, either RL Fact, Java Fact, or primitive types.

You can use an RL Fact to extend a Java application object model by providing virtual dynamic types.

For example:

```
R1: if monthly spend = Customer.monthlySpend
then assert new Temp(three month spend: monthly spend[0] + monthly spend[1] +
monthly spend[2])
R2: if Temp.three month spend > 500
then modify Temp(status: GOLD)
R3: if Temp.status == GOLD
then assert new Result(discount: 0.10).
```

For testing and prototyping with Rules Designer you can create RL Facts and use the RL Facts to write and test rules before you import a schema and switch to XML Facts (you might need to wait for an approved XML schema to be created or to be made available). Switching from RL Facts to corresponding XML Facts involves the following steps:

1. Delete the RL Facts (this action shows validation warnings in the rules or Decision Tables you created that use these RL Facts).
2. Import the XML Facts and give the facts and their properties aliases that match the names of the RL Facts and properties you deleted in step 1.
3. This process should remove the validation warnings if the XML Fact and property aliases and types match those of the RL Facts that you remove.

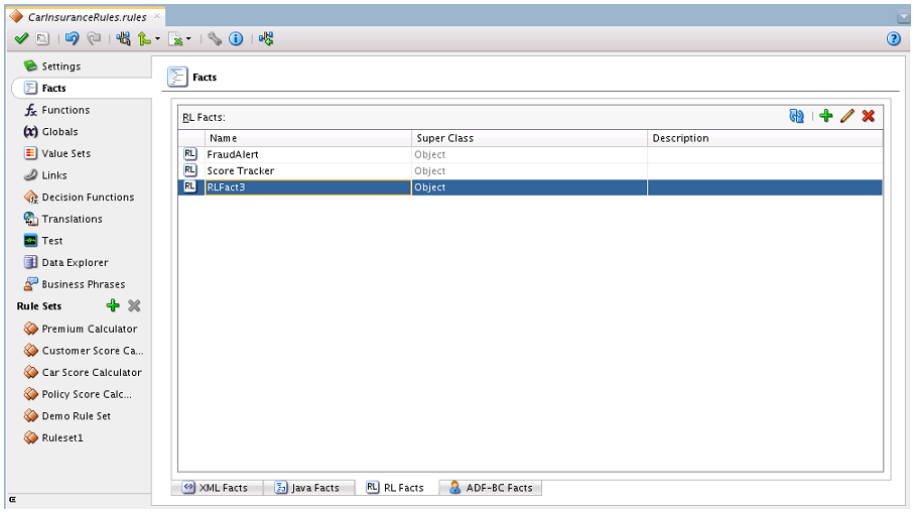
How to Define RL Facts

You add RL Facts from the **Facts** navigation tab.

To define RL facts:

1. In Rules Designer, select the **Facts** navigation tab.
2. Select the **RL Facts** tab in the **Facts** navigation tab as shown in Figure 3-5.

Figure 3-5 RL Facts Tab in Rules Designer



3. In the **RL Facts** tab, click **Create**.
4. In the RL Facts table, in the **Name** field, enter the name for the RL Fact or accept the default name.

5. In the RL Facts table, in the **Description** field, enter a description or accept the default, no description.

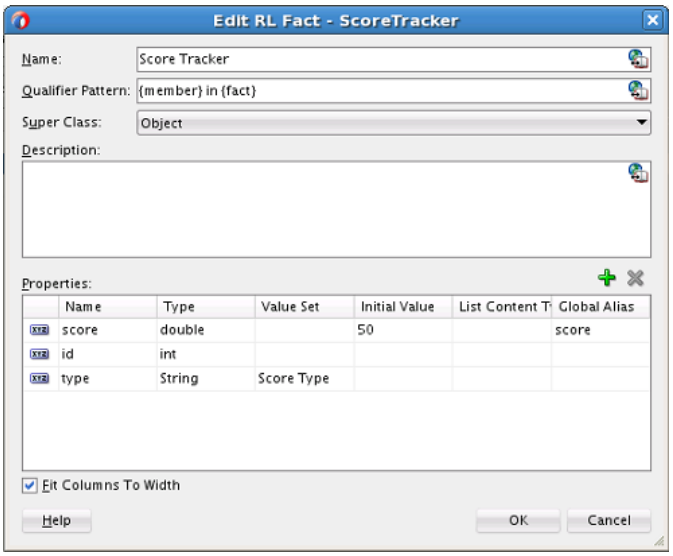
How to Display and Edit RL Facts and Add RL Fact Properties

You add properties to RL Facts using the Edit RL Facts dialog.

To display and edit RL facts and add RL fact properties:

1. In Rules Designer, select the **Facts** navigation tab.
2. In the **RL Facts** tab, double-click the icon for the RL Fact to display or edit the fact. This displays the Edit RL Fact dialog, as shown in Figure 3-6.

Figure 3-6 Edit RL Fact Dialog

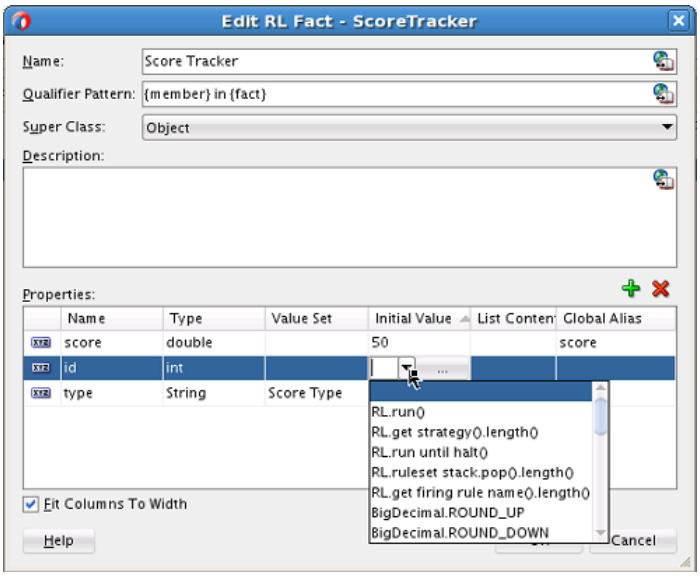


3. To add RL Fact properties, on the Edit RL Fact dialog in the **Properties** area, click **Create**.
   a. In the **Name** field, enter the property name.
   b. In the **Type** field, select a type from the list or enter a property type.
   c. To associate a value set with the property, from the list in the **Value Set** field, select a value set.
   d. To associate an initial value with the property enter a value in the **Initial Value** field.
4. Add additional properties by repeating these steps, as required.
5. Click **OK**.

What You Need to Know About RL Facts

When you add properties to RL Facts using the Edit RL Facts dialog, in the **Properties** area the **Initial Value** field provides a list of possible values as shown in Figure 3-7.

Figure 3-7 Setting RL Fact Property Initial Value



When you are working with some fields in Rules Designer, the initial values list or other lists may be empty. In this case the list is an empty box. Thus, when Rules Designer does not find options to assist you in entering values, you must supply a value directly in the text entry area or click the **Expression Builder** button to display the expression builder dialog.

Working with ADF Business Components Facts

ADF Business Components Facts enable you to use ADF Business Components as Facts in rules and in Decision Tables. By using ADF Business Components Facts you can assert view object graphs representing the business objects upon which rules should be based, and let Oracle Business Rules deal with the complexities of managing the relationships between the various related view objects in the view object graph.

For more information, see Working with Oracle Business Rules and ADF Business Components .

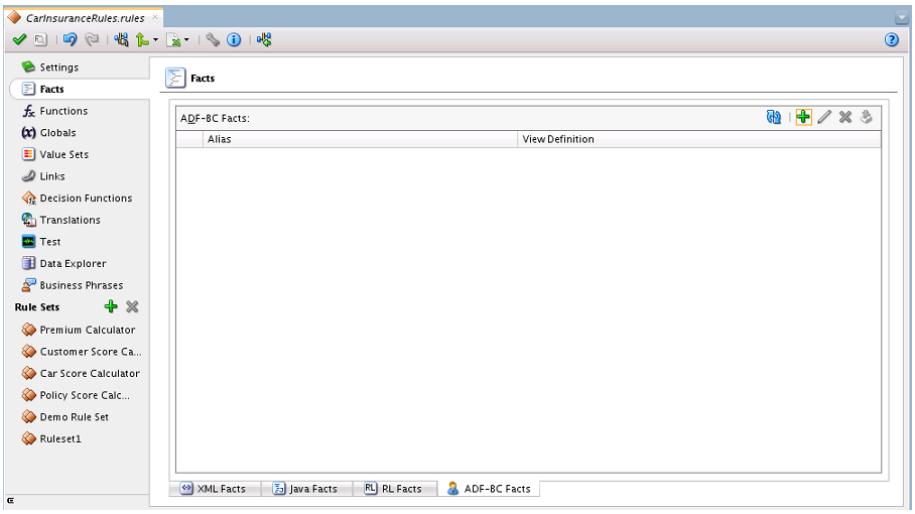
How to Import and Define ADF Business Components Facts

When an ADF Business Components view object is imported, an ADF Business Components fact type is created which has a property corresponding to each attribute of the view object.

To add ADF Business Components facts:

1. Click the **Facts** navigation tab and select the **ADF-BC Facts** tab. This displays the **ADF-BC Facts** table, as shown in Figure 3-8.

Figure 3-8 ADF Business Components Facts Tab



2. Click **Create...**. This opens the Create ADF-BC Fact dialog.
3. In the **Connection** field, from the list, select the connection which your ADF Business Components objects use. The **Search Classpath** area shows a list of classpaths. For more information, see What You Need to Know About ADF Business Components Fact Classpaths.
4. In the **View Definition** field, select the name of the view object to import.
5. Click **OK**. This displays the Facts navigation tab. Note that the imported fact includes a validation warning. For information on how to remove this validation warning, see What You Need to Know About ADF Business Components Circular References.

What You Need to Know About ADF Business Components Fact Classpaths

In the classpath list shown in the **Search Classpath** area in the Create ADF Business Components Fact dialog one of the listed classpaths allows you to see the view object definitions available in your project. In this dialog you only need to click **Add to Classpath** when you need to use a classpath that is not available to your project (this case should be very rare).

What You Need to Know About ADF Business Components Circular References

ADF Business Components Facts can include a circular reference. When this warning is shown in the Business Rule validation log you need to manually resolve the circular reference. To do this you must clear the **Visible** check box for one of the properties that is involved in the circular reference.

What You Need to Know About ADF Business Components Facts

Each ADF Business Components fact type contains a property named `ViewRowImpl` that references the `oracle.jbo.Row` instance that the fact instance represents and a property named `key_values` which points to an `oracle.rules.sdk2.decisionpoint.KeyChain` object that may be used to retrieve the set of key-values for this row and its parent rows.

When working with ADF Business Components Facts you should know the following:

- Relationships between view object definitions are determined by introspection of attributes on the View Definition, specifically, those attributes which are View Link Accessors.

  The ADF Business Components fact type importer correctly determines which relationships are 1-to-1 and which are 1-to-many, and generates definitions in the dictionary accordingly. For 1-to-many relationships the type of the property generated is a `List`, which contains facts of the indicated type at runtime.

- It is not possible to use ADF Business Components fact types which have cyclic type dependencies. These cycles must be broken by the clearing the **Visible** check box for at least one property involved in the cycle.

- ADF Business Components fact types are not Java fact types and do not allow invoking methods on any explicitly created implementation classes for the view object.

  If you need to call such methods then add the view object implementation to the dictionary as a Java fact type instead of as an ADF Business Components fact type. In this case, all getters and setters and other methods become available but the trade-off is that related view objects become inaccessible and, should related view object access be required, these relationships must be explicitly managed.

- Internally, ADF Business Components fact types are instances of RL fact types.

  Thus, you cannot assert ADF Business Components view object instances directly to a Rule Session, but must instead use the helper methods provided in the `MetadataHelper` and `ADFBCFactTypeHelper` classes. For more information, see *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*.

Working with Value Sets

You can create a value set to define a list of values or a list of value ranges to limit the acceptable set of values for a fact or a property of a fact in Oracle Business Rules. You can define a value set as a **Global Value Set** that allows reuse, where a value set is named and stored in the data model, or as a **Local Value Set** that is specified when you define a Decision Table and only applies to one condition expression.

For more information on using a local value set, see How to Add Condition Rows to a Decision Table.

You can use value sets for the following:

- You can associate fact type properties with value sets. This allows you to limit the acceptable set of values for a property of a fact. For more information, see How to Associate a Value Set with a Fact Property.

- A value set defines a list of values or value ranges for some primitive value (number, string, date, boolean, or enumeration). A value set may be associated with a fact type property in order to provide a fixed set of choices for the value of that property, for example, male or female. A value set must be associated with a decision table condition to provide a fixed set of choices for the value of the condition's expression. These choices (values or value ranges) are entered into the condition cells of the decision table.

  The value set values or ranges determine, for each condition expression in a Decision Table, that it has two or more possibilities. Using a value set, each possibility in a condition expression is divided into values or ranges where a cell specifies one value or range from the value set (or possibly multiple values or ranges per cell). For example, if a value set is defined for colors, then the values or

ranges could include a list of strings: "blue", "red", and "orange". A value set that includes integers could have three ranges could have three ranges, less than 1, 1 to 10, and greater than 10. For more information, see How to Add Condition Rows to a Decision Table.

- You can associate globals, functions, and function arguments with value sets. Associating a value set with a global allows for design-time validation that an assigned value is limited to the values specified in the value set. Associating a value set with a function argument validates that the function is only called with values in the value set. Using value sets in this manner allows Rules Designer to report validation warnings for global values and function arguments that are assigned or passed a constant argument value that is not allowed. Associating a value set with a function automatically sets a Decision Table condition row to use that value set when the function is used as the expression for that condition row. Only constant expression values are validated. To ensure that global initial expression values and function parameter expression values are validated against the associated value set, check the 'constant' check box associated with the expression. No runtime checks are applied based on the globals or function arguments associated with value sets. For more information, see How to Associate a Value Set with Functions or Function Arguments.
- In addition to design-time validation you can use an LOV value set to provide options that are displayed in lists when entering expressions in IF/THEN rule tests and actions. For more information, see How to Use Value Sets to Provide Options for Test Expressions.

There are three forms for value sets:

- LOV: Defined by a list of values (see How to Define a List of Values Global Value Set).
- Range: Defined by a list of value ranges, defined by the range endpoints (see How to Define a List of Ranges Global Value Set).
- Enum: Defined by a list of enumerated types that is imported from either of:
  - XML types (see How to Define an Enumerated Type (Enum) Value Set from XML Types).
  - Java facts (see How to Define an Enumerated Type (Enum) Value Set from Java Types).

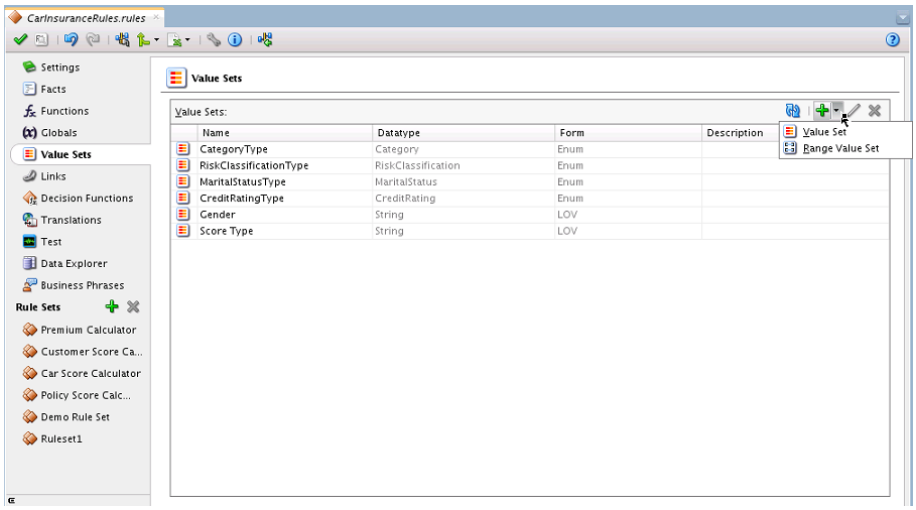
How to Define a List of Values Global Value Set

A list of values value set lets you specify the type and the list of values or ranges for the value set. For more information, see What You Need to Know About List of Values Value Sets.

To define a list of values (LOV) global value set:

1. From Rules Designer select the **Value Sets** navigation tab.
2. From the list next to the **Create Value Set** icon, select **Value Set**, as shown in Figure 3-9.

Figure 3-9 Adding a List of Values Global Value Set



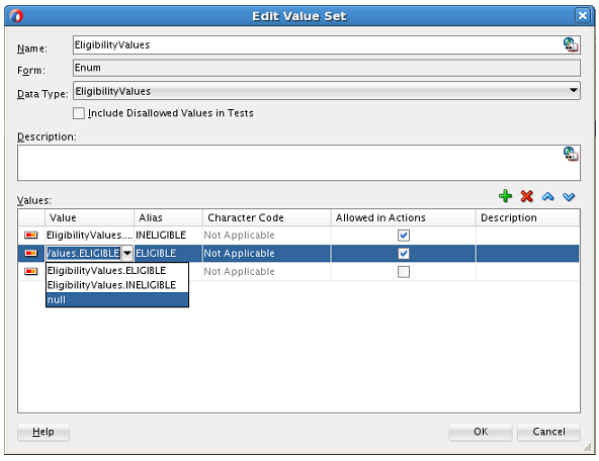
3. Click the Edit button for the value set to display the **Edit Value Set** dialog.
4. In the Edit Value Set dialog, enter the name in the **Name** column.

   Ensure that the value set name is not the same as any fact aliases. This will cause a validation error similar to the following:

   ```
   RUL-05006: The fact type "Rating" has the same alias as an unrelated value set.
   ```

5. In the **Datatype** column, select a data type from list.
6. Enter a **Description.**
7. Click the **Edit** button to add a value, as shown in Figure 3-10.

Figure 3-10 Edit Value Set Dialog



8. For each value that you add, do the following:

   - In the **Value** field, enter a value. Note that for String type values in an LOV value, you can select the entire string with three clicks. This allows you to enter the string. Rules Designer adds the same alias without quotation marks.

You can change the order of values in the list of Value set by editing the Value set dialog for a value set. Click the **Move up** or **Move Down** button to change the order.

- In the **Alias** field, enter an alias.

  For more information on specifying aliases, see How to Define a List of Ranges Global Value Set.

- In the **Allowed in Actions** field, select this if the value is an allowable value.

  For more information on the **Allowed in Actions** field and the **Include Disallowed Values in Tests** field, see What You Need to Know About the Value Set Allowed in Actions Option.

- In the **Description** field, enter a description.

9. Add additional values by clicking the **Create** button as needed for the value set.
10. On the Edit Value Set window, click **OK**.

You can control rule ordering in a Decision Table by changing the relative position of the values or ranges in an LOV value set associated with a condition expression in a Decision Table.

How to Define a List of Ranges Global Value Set

A list of ranges value set lets you specify the type and the endpoints for values or ranges in the value set. For more information, see What You Need to Know About Range Value Sets.

To define a list of ranges (range) global value set:

1. From Rules Designer select the **Value Sets** navigation tab.
2. From the list next to the **Create Value Set** button, select **Range Value Set**.
3. Double-click in the **Data Type** field. This displays the Edit Value Set dialog, enter the values set name in the Name field.
4. In the **Data Type** field, from the list, select the appropriate data type for the value set.
5. Click the **Create** repeatedly to add the number of values or ranges that you need in the value set.

   In these steps you add three values. You start with the default values. After changing the defaults, they should have the following values:

   - greater than 1000
   - between 500 and 1000, inclusive
   - less than 500

   Rules Designer added the values with the default values of 50 and 0 and a negative Infinity (-Infinity) value.

6. Starting at the first or top value, in the **Endpoint** field, double-click the default value and enter the top value endpoint, and press **Enter**.

In this example, enter 1000 for the first value.

7. In the **Included Endpoint** field, select the check box as appropriate to include or exclude the value endpoint.

   In this example, you can leave this check box checked to include the value endpoint.

8. In the **Allowed in Actions** field select the check box as appropriate to include the value in the value set allowable values.

   For more information on the **Allowed in Actions** field and the **Include Disallowed Values in Tests** field, see What You Need to Know About the Value Set Allowed in Actions Option.

9. Optionally, in the **Alias** field double-click the default value and enter the desired value alias, and press **Enter**.

   The alias appears in Decision Tables that use this value set. Use an alias to give a more meaningful name to the value than the default value (the range-based Range value).

   Note that most names and aliases in Oracle Business Rules allow only letters, numbers, embedded single spaces, and the characters $, _, ', ., -, /, and :. However, value aliases allow additional characters, such as [0..1]. If an alias contains such additional characters, then you cannot refer to the value by the alias in the action cells in a Decision Table. In these cases, you can use the value name, which is also known as the value.

   The Range field is read-only: it clearly identifies the actual range associated with the value regardless of the Alias value. For more information, see What You Need to Know About Range Value Sets).

10. Moving down the list of values, for each subsequent vale, repeat from Step 6. For the second value, enter the endpoint value 500.

11. In the **Edit Value Set** dialog, click **OK**.

How to Define an Enumerated Type (Enum) Value Set from XML Types

When you import an XML schema, if the XSD contains enumeration values Rules Designer automatically creates an enumerated type value set for each enumeration. Although enumerated type value sets are read-only, you can change the order of values.

For more information, see What You Need to Know About XML Facts.

To define an enumerated type (enum) value set from XML types:

1. Obtain an XSD with the desired enumerations.

   The following example shows the `order.xsd` schema file which contains the enumeration `Status`.

```
<?xml version="1.0" ?>
<schema attributeFormDefault="qualified" elementFormDefault="qualified"
        targetNamespace="http://example.com/ns/customerorder"
        xmlns:tns="http://example.com/ns/customerorder"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="CustomerOrder">
    <complexType>
```

```
      <sequence>
        <element name="name" type="string" />
        <element name="creditScore" type="int" />
        <element name="annualSpending" type="double" />
        <element name="value" type="string" />
        <element name="order" type="double" />
      </sequence>
    </complexType>
  </element>
  <element name="OrderApproval">
    <complexType>
      <sequence>
        <element name="status" type="tns:Status"/>
      </sequence>
    </complexType>
  </element>
  <simpleType name="Status">
    <restriction base="string">
        <enumeration value="manual"/>
        <enumeration value="approved"/>
        <enumeration value="rejected"/>
    </restriction>
  </simpleType>
</schema>
```

2. Open a dictionary in Rules Designer and create XML facts using the specified schema that contains the enumeration. For more information, see Working with XML Facts.

3. Click the **Value Sets** navigation tab and select the Enum value to see the value set. Notice that the imported `Status` enumeration values are imported as values with the XSD-specified values.

   You can change the order of values in an Enumerated Value set by editing the Value set dialog for a value set. Click the **Move up** or **Move Down** button to change the order.

You can control rule ordering in a Decision Table by changing the relative position of the values in an enum value set associated with a condition expression in a Decision Table.

How to Define an Enumerated Type (Enum) Value Set from Java Types

When you import a Java enum, Rules Designer automatically creates an enumerated type value set for each Java enum. Although enumerated type value sets are read-only, you can change the order of values.

To define an enumerated type (enum) valueset from Java facts:

1. Create or obtain the Java class with the desired enumerations.

   The following code example shows the `RejectPurchaseItem.java` class which contains enumeration `OrderSize`.

```
package com.example;


public class Class1

{

    public enum OrderSize { SMALL, MEDIUM, LARGE };
```

```
    public Class1()

    {
    }
}
```

2. In Rules Designer open a dictionary and create a Java Fact using the Java class. For more information, see Working with Java Facts.

3. In Rules Designer click the **Value Sets** navigation tab and select the Enum value set.

   You can change the order of values or ranges in an enumerated type valueset by editing the Value set dialog for a value set. Click the **Move up** or **Move Down** button to change the order.

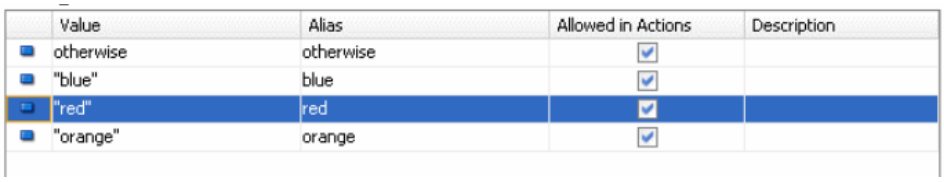
You can control rule ordering in a Decision Table by changing the relative position of the values or ranges in an enum value set associated with a condition expression in a Decision Table.

What You Need to Know About List of Values Value Sets

In a Decision Table, the order of the values in a value set associated with a condition expression determines the order of the condition cells, and thus the order of the rules. You can control rule ordering in a Decision Table by changing the relative position of the values in a list of values value set associated with a condition expression; however, you cannot reorder ranges.

Figure 3-11 shows a value set definition in Rules Designer for a value set named colors using a list of values.

Figure 3-11 Value Set Definition Using List of Values



As shown in Figure 3-11, by default with a List of Values value set there is a value `otherwise` included with the list of values (LOV). This value, `otherwise`, is distinct from all other values and matches all values of the type that have no other value or range. Thus, with `otherwise` in the list of values a condition expression that uses the value set can handle every value and provides a match for every value of the specified type, where a match is either a defined value or the `otherwise` value. The `otherwise` value cannot be removed from an LOV value set but it can be excluded by clearing the **Allowed in Actions** check box (when `otherwise` is excluded an attempt to assign any value that is not in the list of values in the value set causes a validation warning).

Table 3-3 shows the value set values that Rules Designer supports for LOV value sets.

Table 3-3 Supported Types for LOV Value Sets

Type	Description
Java primitive types	This includes `int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, and `float`

Table 3-3 (Cont.) Supported Types for LOV Value Sets

Type	Description
`String`	Contains `String` types
`Time, DateTime, and Date`	Contains `Time, DateTime, and Date` types in the current locale

Note:

You are not required to specify an LOV value set when you use a boolean type in a Decision Table. For boolean types, Oracle Business Rules provides built-in values or ranges for the possible values (`true` and `false`).

What You Need to Know About Range Value Sets

When you add a value or range to a List of Ranges value set, the value is calculated based on the currently selected value and the next highest value. When you change the endpoint value the value is automatically sorted in the value set; thus, it does not matter where a new range is added. However, it is possible for Rules Designer to not have values between the current value set endpoint value and the endpoint value. In this case, Rules Designer shows a validation warning of the following form:

```
RUL-05849: Valueset has duplicate bucket value "4999"
```

To correct this problem you must modify value endpoints to remove the duplicate value.

Table 3-4 shows the types Rules Designer supports for Range values.

Table 3-4 Supported Types for Range Values

Type	Description
Selected primitive types	This includes: `int`, `double`, `short`, `long`, and `float`
`Time, DateTime, and Date`	Contains `Time, DateTime, and Date` types in the current locale

Note the following conventions for the Range field:

- Logical operator: specifies a range with respect to positive or negative infinity. For example, `">=25"` means "from 25 to positive infinity" and `"<18"` means from negative infinity up to but not including 18.
- Square bracket `"["`: specifies a range that includes this end point value. For example, `"[18..25)"` means "from 18 up to but not including 25".
- Round bracket `")"`: specifies a range that excludes this end point value. For example, `"(18..25]"` means "over 18, not including 18, up to and including 25".

What You Need to Know About the Value Set Allowed in Actions Option

When you define values or ranges in a value set you might define some ranges or values corresponding to non-permissible values. For example, in a value set for driver

ages you would typically not allow a value that contains values less than 0. Thus, when a fact with driver data includes an age property associated with a driver ages value set, then you should not be able to create or modify a fact that has the age property set to a value such as -1. In a value set you select **Allowed in Actions** for valid values and clear this option for invalid values.

The value set option **Include Disallowed Values in Tests** allows you to include all the values, whether **Allowed in Actions** is selected or not, in Decision Table conditions and in rule tests. By including all values or ranges you can explicitly test for illegal values. Using the option **Include Disallowed Values in Tests** you can handle two possible cases:

1. The input data for the Oracle Business Rules Engine is clean and does not contain invalid data (such as a negative age). In this case, you should clear the **Include Disallowed Values in Tests**. Note: the reason you do not want to make `age < 0` an **Allowed in Actions** is this provides design time validation warnings if you try to create an action that uses an invalid value, such as the following: `modify(driver, age: -1))`. For more information, see Using Value Sets as Constraints for Options Values in Rules.

2. You want to consider excluded values in rule tests and in Decision Tables. In this case, you should select **Include Disallowed Values in Tests**. This is useful when the input data for the Oracle Business Rules Engine may not be clean and may contain invalid data (for example an invalid negative age). A Decision Table that provides actions for all value sets could include cases for excluded values and provide an appropriate action, such as asserting an error fact. To handle this you could either select the **Allowed in Actions** field for every value in the value set, or, leave the **Allowed in Actions** field configured as is and select the **Include Disallowed Values in Tests** field. Using the **Include Disallowed Values in Tests** field is not only convenient, you do not need to reconfigure every value, it also preserves the configuration of **Allowed in Actions** so that you can easily reuse this value set to handle the first case (when you clear **Include Disallowed Values in Tests**).

What You Need to Know About Values

When you enter a value in a value set, the value you supply must be valid for the type specified for the value set. If the value you enter is not valid for the value set type, Rules Designer makes the value you supply a string by adding quotation marks. Adding quotation marks is the only way to make a legal literal when the user provided data is not appropriate for the specified type. For example, if you add an int type LOV value set, and then supply a value 2.2, Rules Designer shows a warning such as the following:

```
RUL-05833: Invalid characters "2.2" in value
```

To fix this problem either enter a valid value for the value, for example in this case the value 2, or change the type of the value set.

For an additional example, when you enter a value for a value, for example if you enter a value with value set with data type short and add a value with the value 999999, Rules Designer assigns this the value "999999". The maximum value for a short is 32767. In this case you see a warning related to the value, similar to the previous example, because a String is not a valid value for a value set with data type short. The solution to this is to enter appropriate values for all values (in this example, enter a value less than or equal to 32767).

Associating a Value Set with Business Terms

After you define a global value set, you can associate parts of the data model with the global value set (if their types are compatible). In this way, condition cells in the **Conditions** area can automatically be assigned a value set when you define a Decision Table. Also, when a value set is associated with a business term, Oracle Business Rules uses the values or ranges that you define as constraints for the values for expressions for the business terms in rules.

You can associate the following four kinds of business terms with a value set:

- Fact Property
- Function Result
- Function Argument
- Global Value

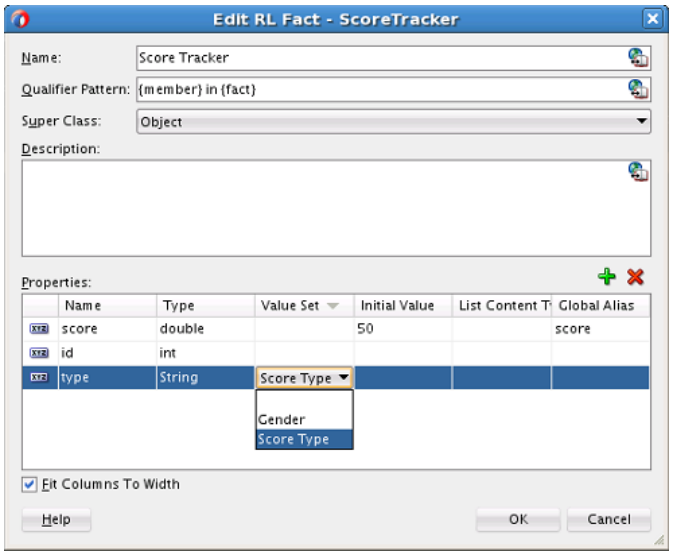
How to Associate a Value Set with a Fact Property

To prepare for creating Decision Tables, you can associate a global value set with fact properties in the data model.

To associate a value set with a fact property:

1. From Rules Designer, select the **Facts** navigation tab.
2. Select the fact type to edit and click the **Edit** button. This displays the appropriate Edit Fact dialog for the fact type you select.
3. In the Properties table, under **Value Set**, select the cell for the appropriate fact property and from the list select the value set you want to associate with the property. For example, see Figure 3-12.

Figure 3-12 Defining a Value Set for a Property



4. On the Edit Fact page, click **OK**.

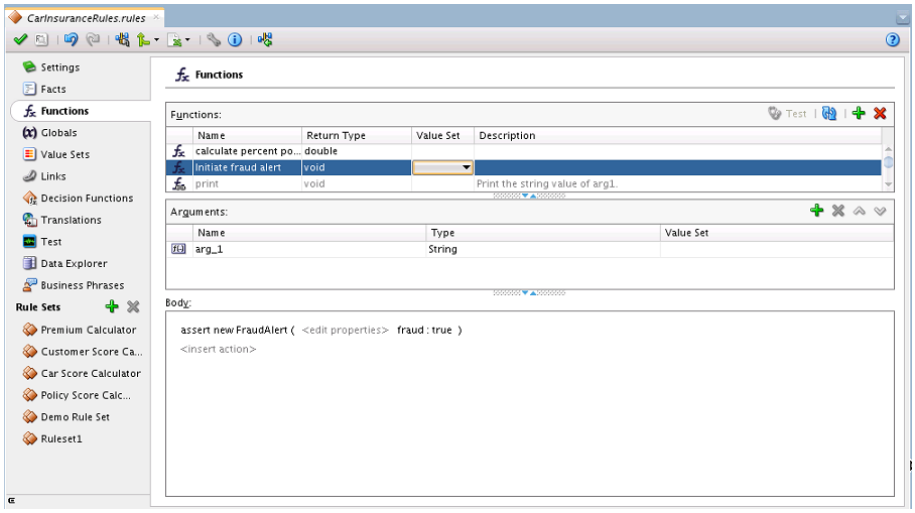
How to Associate a Value Set with Functions or Function Arguments

To prepare for creating Decision Tables you can associate a global value set with functions in the data model.

To associate a value set with a function return value:

1. From Rules Designer, select the **Functions** tab.
2. Select the function to edit. This shows the function arguments and the function body for the specified function.
3. In the Functions table, under **Value Set**, select the cell and from the list select the value set you want to use. For example, see Figure 3-13.

Figure 3-13 Defining a Value Set for a Function Return Value



How to Associate a Value Set with a Function Argument

To associate a value set with a Function argument:

1. From Rules Designer, select the **Functions** navigation tab.
2. Select the function to edit. This shows the function arguments and the function body for the specified function.
3. In the Functions table, in the **Arguments** area select the appropriate argument.
4. For the specified argument, under **Value Set**, select the cell and from the list select the value set you want to use.

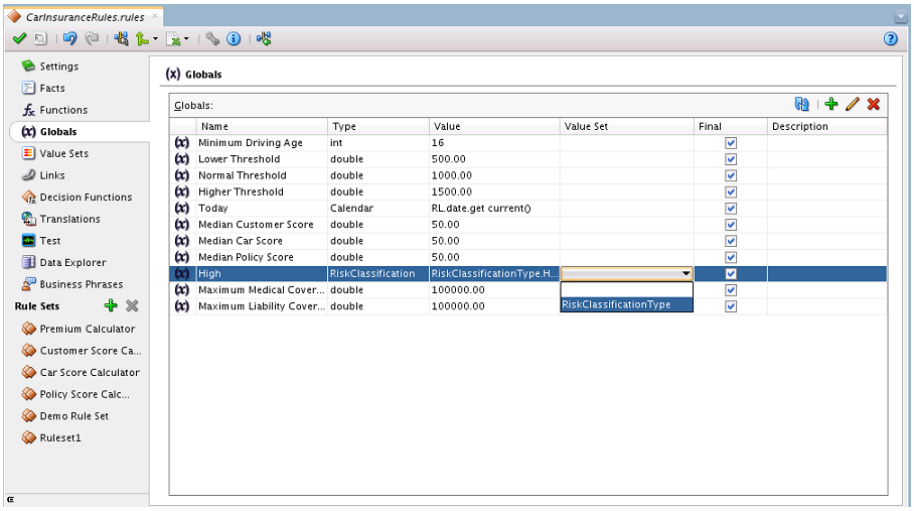
How to Associate a Value Set with a Global Value

To prepare for creating Decision Tables, you can associate a global value set with global values in the data model.

To associate a value set with a global value:

1. From Rules Designer, select the **Globals** navigation tab.
2. Select the global value to edit.
3. In the Globals table, under **Value Set**, select the cell for the appropriate global value, and from the list, select the value set that you want to associate with the global value. For example, see Figure 3-14.

Figure 3-14 Defining a Value Set for a Global Value



4

Working with Rulesets and Rules

This chapter describes the Oracle Business Rules data model element called a ruleset that you use to group one or more rules or Decision Tables. It also discusses how to work with dictionaries, nested tests, and simple and tree mode rules, and Expression Builder.

The chapter includes the following sections:

For more information, see What Are Rulesets?.

Introduction to Working with Rulesets, Rules, and Business Phrases

Use business rules to define key decisions and policies for a business.

Some of these decisions and policies include:

- Business Policies: for example spending policies and approval matrices
- Constraints: for example valid configurations or regulatory requirements
- Computations: for example discounts, premiums, or scores

- Reasoning Capabilities: for example offers based on customer value

Oracle Business Rules provides multiple approaches to writing rules:

- IF/THEN rules - rules are expressed as IF/THEN statements. There are two ways of modeling IF/THEN rules. General rules use a pseudo-code language to express rule logic. Verbal rules use natural language statements to express rule logic.
- Decision Tables, which display multiple related rules in a single spreadsheet-style view.

Business phrases are used to provide a natural language vocabulary for the construction of verbal rules' tests and actions. They are not used in general rules.

This chapter includes details for working with IF/THEN rules. For information on working with Decision Tables, see Working with Decision Tables.

Working with Rulesets

A ruleset provides a unit of execution for rules and Decision Tables. In addition, rulesets provide a unit of sharing for rules; rules belong to a ruleset. Multiple rulesets can be executed in order. This is called rule flow. The ruleset stack determines the order. The order can be manipulated by rule actions that push and pop rulesets on and off the stack. In rulesets, rule priority specifies the order in which the rules should be fired.

Rulesets also include an effective date specification that controls when a ruleset is active. A ruleset can be:

- always active
- active during a time range
- active during a date range
- active during a time and date range

How to Create a Ruleset

All rules and Decision Tables are created in a ruleset. A ruleset organizes rules and Decision Tables into a unit of execution.

To create a rule set:

1. In Rules Designer, go to the **Rule Sets** tab.
2. Click the Create rule set button. This displays the Create Rule Set dialog.
3. Enter a name in the **Name** field.

> **Note:** The names of ruleset and ruleset alias, business rules, and any other rule objects must begin with a letter and can contain only letters and numbers. They must not contain spaces or special characters like `.`, `-`, `_`, `:`, `,`, `""`.

4. Enter a description in the **Description** field.
5. Click **OK**.

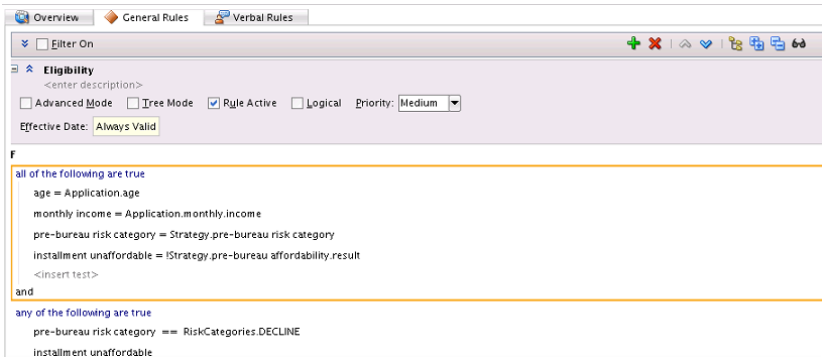
How to Set the Effective Date for a Rule Set

Effective date support provides the ability to specify a start date and an end date for a ruleset, a rule or a Decision Table. For a ruleset the effective date defines the date range in which the rules and Decision Tables within the ruleset are effective. For more information on effective dates, see Using Date Facts_ Date Functions_ and Specifying Effective Dates.

To set the effective date for a ruleset:

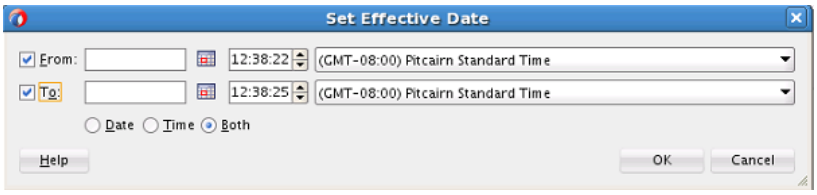
1. Select the ruleset name from the **Rule Sets** tab.
2. Click the navigation button next to the ruleset name to expand the ruleset information to show the ruleset **Name**, **Description**, and **Effective Date** fields, as shown in Figure 4-1.

Figure 4-1 Ruleset Showing Effective Date Field



3. Select the **Effective Date** entry. This displays the Set Effective Date dialog, as shown in Figure 4-2.

Figure 4-2 Using the Set Effective Date Dialog



4. Use the Set Effective Date dialog to specify the effective dates for the ruleset. Clicking the **Set Date** button displays a calendar to assist you in entering the **From** and **To** field data.

You can specify an effective start date and or an effective end date for a ruleset, a rule, or a Decision Table. For information on specifying the effective date for a ruleset, see How to Set the Effective Date for a Rule Set.

How to Set the Effective Date for a Rule

You can specify an effective start date and or an effective end date for a rule.

To set the effective date for a rule:

1. Select the ruleset name from the **Rulesets** navigation tab.
2. Select a rule within the ruleset.
3. Next to the rule name click **Show Advanced Settings**.
4. Select the **Effective Date** field. This displays the Set Effective Date dialog.
5. Use the Set Effective Date dialog to specify the effective dates for the rule. Clicking the **Set Date** button displays a calendar to assist you in entering the **From** and **To** field data.
6. In the Set Effective Date dialog, click **OK**.

How to Use a Filter to Display Matching Rules in a Ruleset

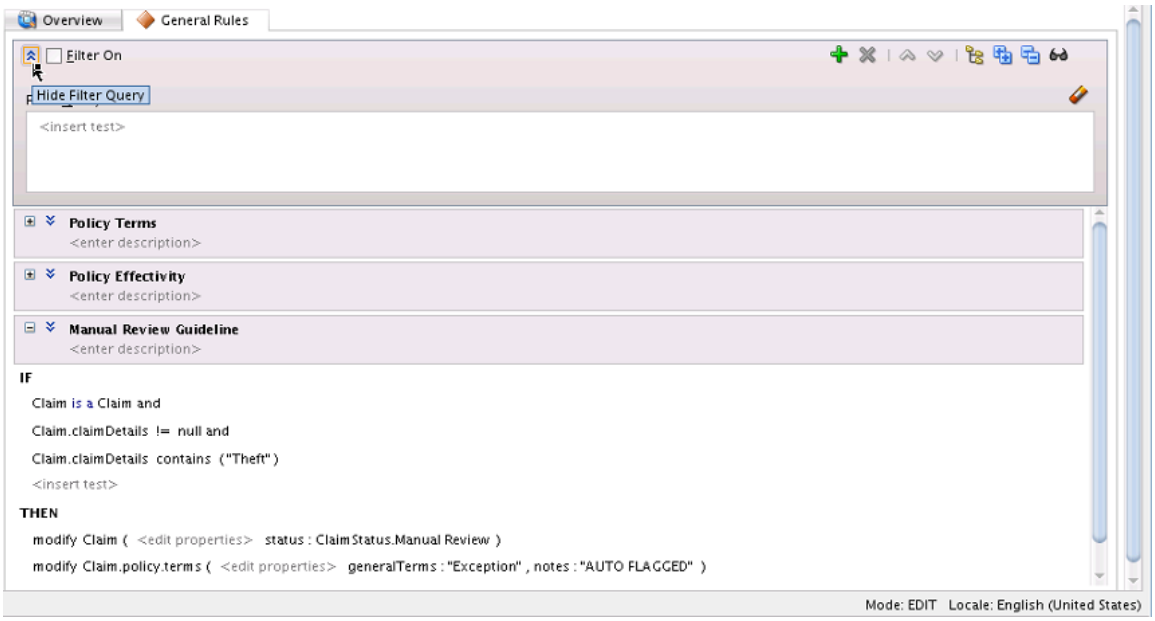
As the number of rules in a ruleset increases, it can be difficult to navigate the list of rules. You can instruct Rules Designer to filter the list of rules, to display only rules of interest. For example, you can display only active rules or only rules that have validation warnings.

For more information on creating rules, see Working with Rules.

To use a filter to display matching rules in a ruleset:

1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. To show the rule filter settings, next to the ruleset name, click **Show Filter Query** as Figure 4-3 shows.

Figure 4-3 Showing or Hiding a Filter Query in a Rule Set



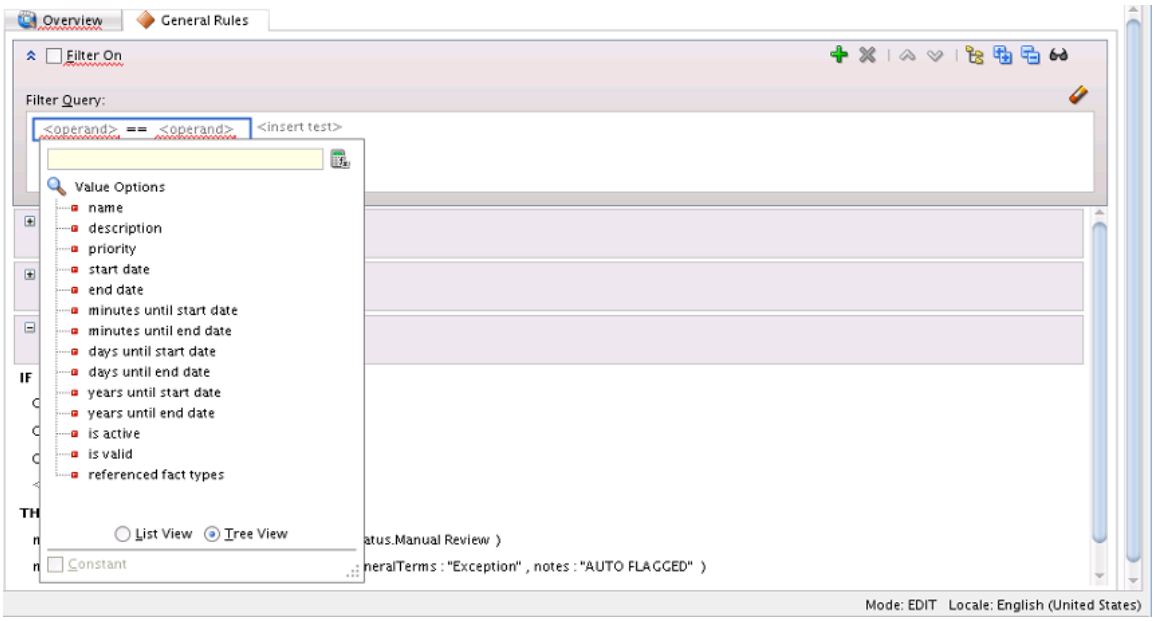
3. In the **Filter Query** field, click **<insert test>** to insert a default test.
4. Configure the default test.

In this case, as shown in Figure 4-4, when you click an **<operand>** you can choose from the rule-specific options shown in Table 4-1.

Table 4-1 Rule Filter Query Operands

Operand	Description
`name`	Matches against the rule name.
`description`	Matches against the rule description.
`priority`	Matches against the rule priority. For more information, see How to Set a Priority for a Rule.
`start date`	Matches against the rule start date. For more information, see What You Need to Know About Effective Dates.
`end date`	Matches against the rule end date. For more information, see What You Need to Know About Effective Dates.
`minutes until start date`	Matches against a specified number of minutes until the rule start date. For more information, see What You Need to Know About Effective Dates.
`minutes until end date`	Matches against a specified number of minutes until the rule end date. For more information, see What You Need to Know About Effective Dates.
`days until start date`	Matches against a specified number of days until the rule start date. For more information, see What You Need to Know About Effective Dates
`days until end date`	Matches against a specified number of days until the rule end date. For more information, see What You Need to Know About Effective Dates
`years until start date`	Matches against a specified number of years until the rule start date. For more information, see What You Need to Know About Effective Dates
`years until end date`	Matches against a specified number of years until the rule end date. For more information, see What You Need to Know About Effective Dates
`is active`	Matches against whether the rule is active. For more information, see How to Select the Active Option.
`is valid`	Matches against whether the rule has validation warnings. For more information, see Understanding Rule Validation.
`referenced fact types`	Matches against one or more fact types.

Figure 4-4 Filter Query Operands



For more information, see How to Define a Test in a Verbal Rule.

5. Select the operator to choose an operator for the comparison. For example, for the name you can select **contains** from the operand list.

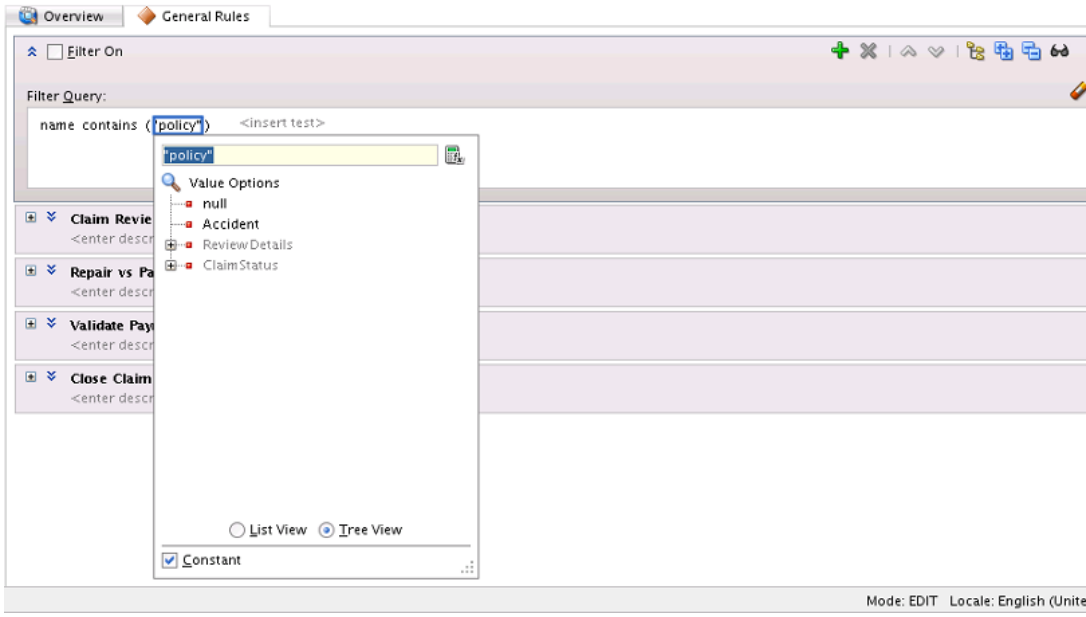
6. Enter a comparison operand for the right-hand-side of the filter test. For example, enter the string `Policy`.

7. When the filter query is complete you can apply the filter to the rules in the ruleset:

   a. To apply the filter, select the **Filter On** check box.

      Rules Designer displays only the rules that match the filter query as Figure 4-5 shows.

Figure 4-5 Enable Filter Query in a Rule Set



b. To disable the filter query, clear the **Filter On** check box.

Rules Designer displays all the rules in the ruleset.

c. To delete the filter query, select it and press **Delete** or click **Clear Filter**.

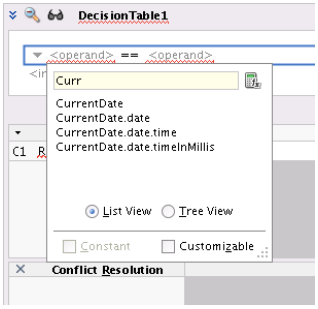
Using Auto Complete when Selecting Component Values from a List

The Rules Designer enables you to easily set values for the following components of a business rule:

- Expressions
- Conditions
- Operands
- Actions

You can edit these components by clicking them in the Rules Editor and selecting the desired value from a drop down list or tree. You can also enter the name of the desired value in the text area at the top of the list. When you begin entering text, the list of options are filtered as shown in Figure 4-6.

Figure 4-6 Using the Auto Complete Function



In this figure, only the options beginning with the text entered are displayed.

Working with Rules

You create business rules to process facts and to obtain intermediate conclusions that Oracle Business Rules can process. You create rules in a ruleset, so before working with rules you must create a ruleset (or use the default ruleset).

For more information on creating a ruleset, see Working with Rulesets.

You can test rules as you design them without having to deploy your application. For more information, see Testing Decision Functions Using a Rules Function.

Rules Designer rule validation can assist you when you work with rules by showing warnings for incorrect or incomplete rules. To show the validation log window, click the **Validate** button or select **View**>**Log** and select the **Business Rule Validation** tab. Note that you must correct all warnings before you can test or deploy rules. For more information on rule validation, see Understanding Rule Validation.

As the number of rules in a ruleset increases, you can configure Rules Designer to filter the list of rules to show only rules of interest. For more information, see How to Use a Filter to Display Matching Rules in a Ruleset.

How to Add General Rules

To create a general rule, first add the rule to a ruleset, and then insert tests and actions. Actions are associated with pattern matches. At runtime, when a test in the **IF** area of a rule matches, the Rules Engine activates the **THEN** action and prepares to run the actions associated with the rule.

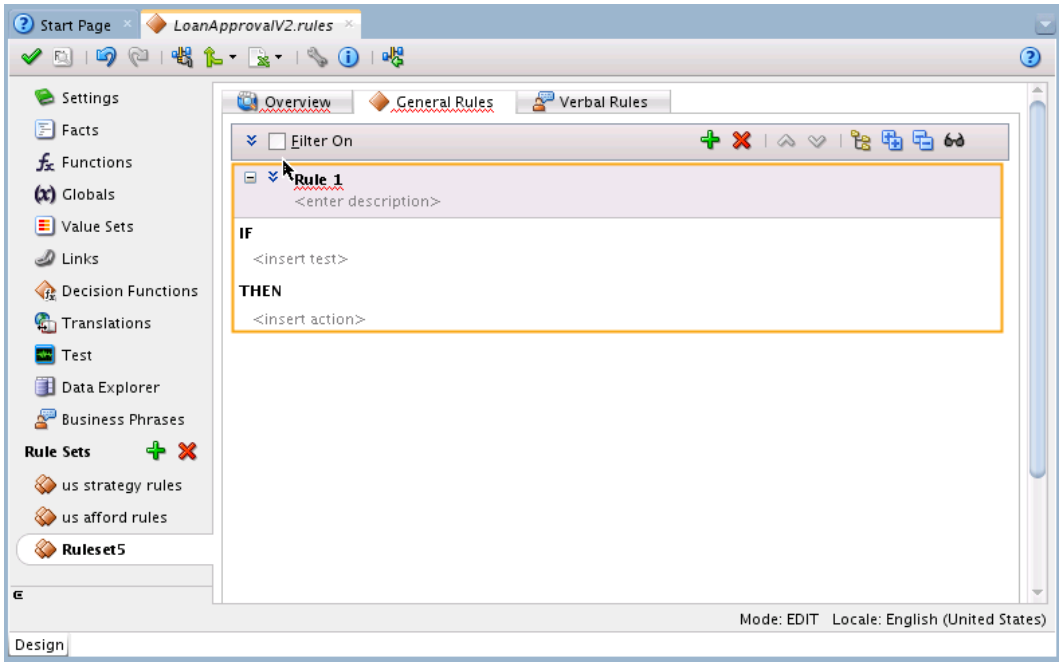
By default, Rules Designer creates rules which fire for each matching fact. Select **Advanced Mode** to enable other options, such as a rule in which the same fact type matches more than once, or never. For more information on advanced mode and showing advanced settings, see Using Advanced Settings with Rules and Decision Tables.

To add a general rule to a ruleset:

1. In Rules Designer, select the **Rule Sets** tab and click **+.**
2. In the **Overview** tab, in the **General Rules** panel, click **+**. Alternatively, in the **General Rules** tab, click **Create Rule** or **Create** (**+**).

   For example, click **Create Rule** to add a rule named `Rule_1`, as shown in Figure 4-8.

Figure 4-7 Adding a Rule to a Rule Set



Note: Delete rules only from the rule editor region by clicking the delete button. Rules do not get deleted cleanly when you delete them from left tree navigation.

How to Add Verbal Rules

Verbal rules are created and executed in a similar fashion to general rules. However, there are some differences.

To create a verbal rule, first add the rule to a ruleset, and then insert tests and actions. As you define a verbal rule, you add business phrases which can either be automatically derived by the system, or defined by you. You can define business phrases before writing a rule, or after.

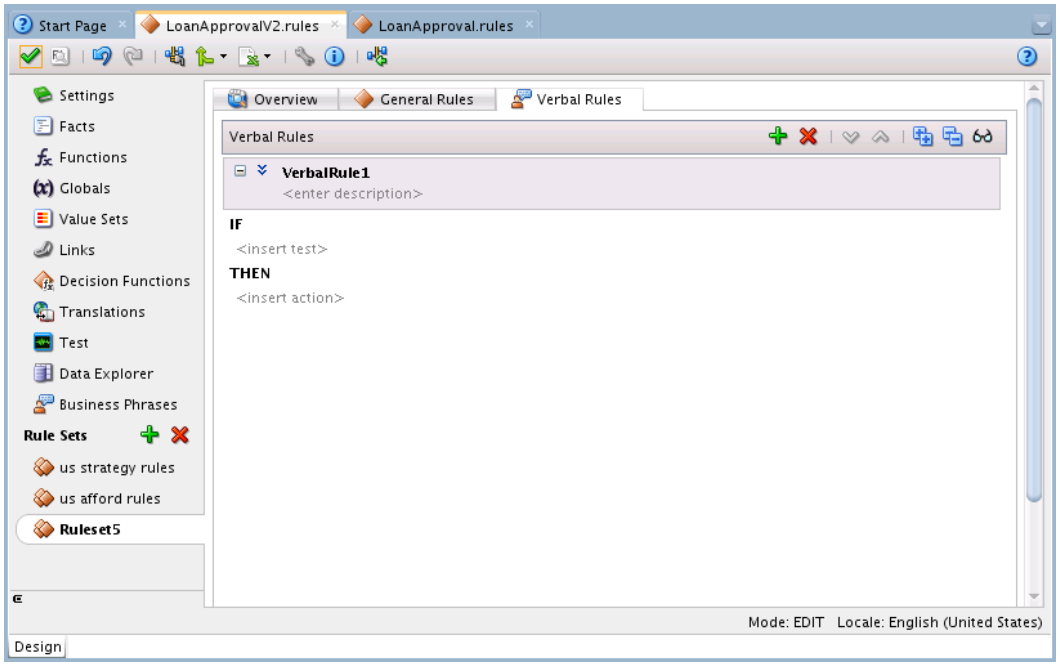
Verbal rules do not support Advanced Mode or Tree Mode.

To add a verbal rule to a ruleset:

1. In Rules Designer, select the **Rule Sets** tab and click **+.**
2. In the **Overview** tab, in the **Verbal Rules** panel, click **+**. Alternatively, in the **Verbal Rules** tab, click **Create Verbal Rule** or **Create** (**+**).

   For example, click **Create Verbal Rule** to add a rule named `VerbalRule1`, as shown in Figure 4-8.

Figure 4-8 Adding a Verbal Rule to a Rule Set

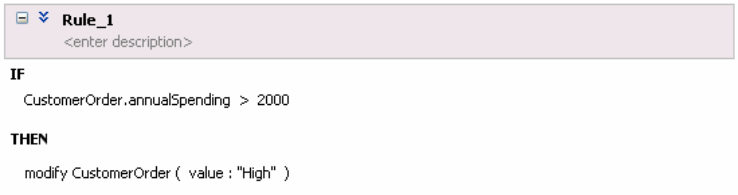


How to Define a Test in a Rule

To create a test in a rule you add conditions for facts. For example, with a sample `CustomerOrder` fact with an annual spending property, you can add a test to determine if a customer order is associated with a high value of spending, based on the annual spending for the customer. Note that you can use value sets to limit the values for tests and actions in rules. For more information, see Using Value Sets as Constraints for Options Values in Rules.

Figure 4-9 shows this sample rule.

Figure 4-9 Adding a Test to a Rule



At runtime, when this rule is processed the Rules Engine checks the facts against rule pattern tests that you define to find matching facts. For this sample rule, `Rule_1`, when a fact matches the Rules Engine modifies the fact and then modifies the value property to "High."

To define tests in rules:

1. In Rules Designer, click **+** from the **Rule Sets** tab, add or select the rule you want to use, for example, select **Rule_1**.
2. The **IF** area of the rule includes a left-hand-side `<operand>` and a right-hand-side `<operand>`, as shown in Figure 4-14.

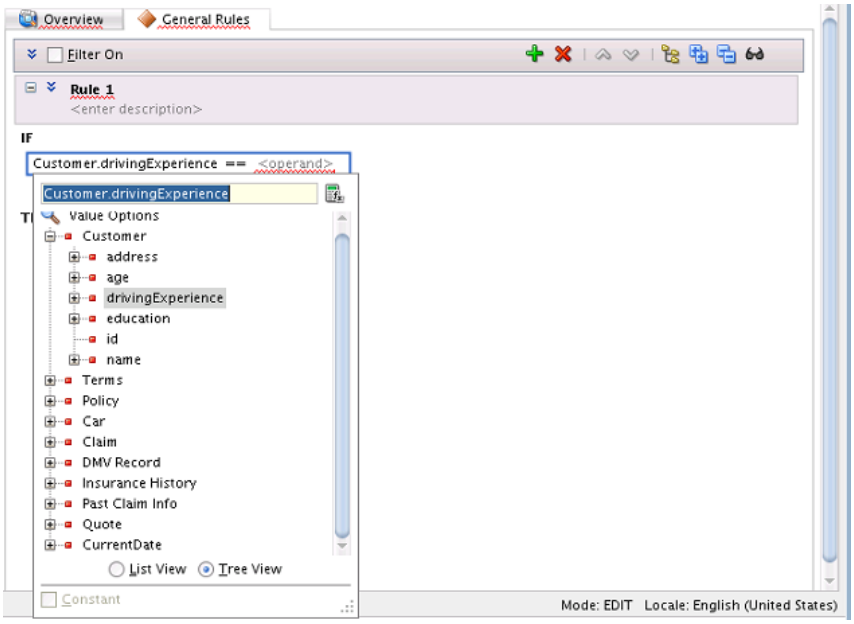
Figure 4-10 Rule Test with Left-hand-side operand and Right-hand-side operand

3. In the rule, click **<insert test>** and choose **simple test**, for example.

4. In a test, you replace the left-hand-side operand with a value.

   To do this, select the left-hand-side **<operand>**. This displays a text entry area and a list, as shown in Figure 4-15:

Figure 4-11 Configuring the Left-hand-side Operand of a Test in a Rule



   a. To enter a value use the list to select an item from the value options.

      You can view the options using a single list, by selecting **List View**, or using a navigator by selecting **Tree View**.

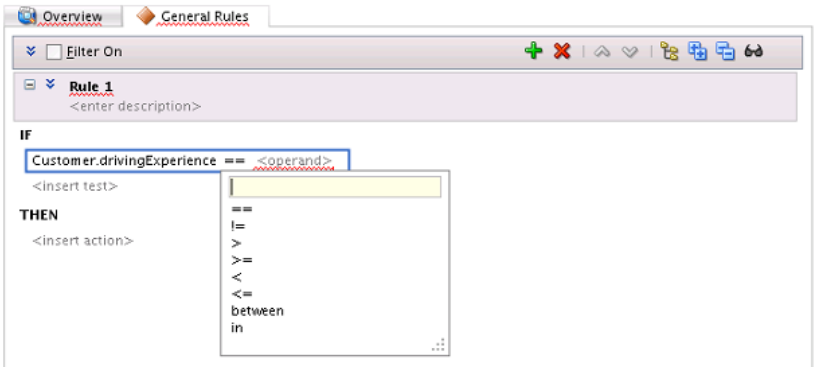
   b. To enter a literal value, type the value into the text entry area and press **Enter**.

      The value you enter must agree with the type of the corresponding operand. For example, in the test **IF** `CustomerOrder.annualSpending` > **<operand>**, valid values for **<operand>** must agree with the type of `CustomerOrder` field `annualSpending`.

5. In a test, you replace the operator with the desired logical operator or accept the default (==). To do this, select the default == operator. This displays a field and a list. The list may contain additional operators, depending on the datatype of the left operand. For example, to test strings, if you select a String operand on the left

hand side, then additional String operators, such as startsWith and equalsIgnoreCase are available as shown in Figure 4-16.

Figure 4-12 Configuring String Operators in a Rule



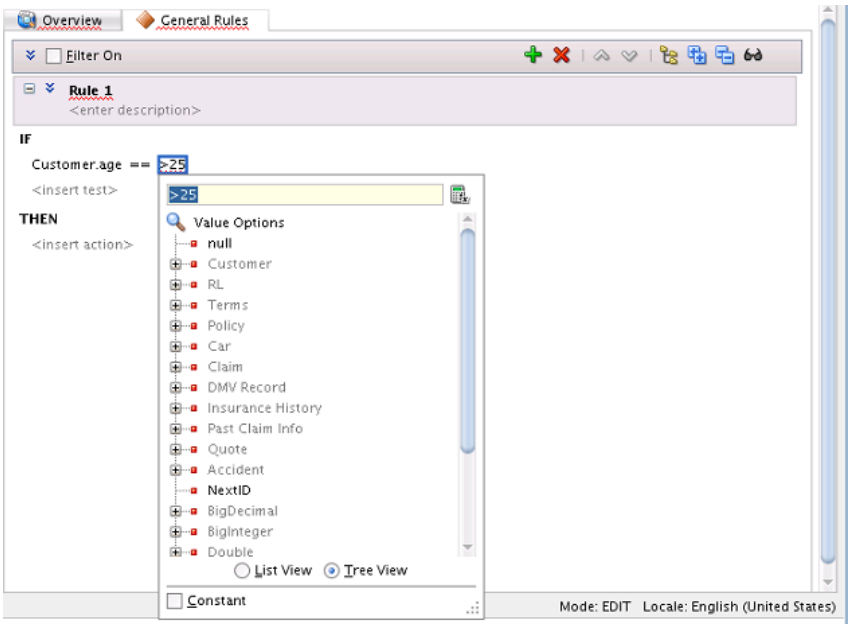
Similarly, to test a logical condition between the left-hand and right-hand operands, select one of the logical operators: == (equality), != (not equal), > (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to). For more information on the operators, see Oracle Business Rules Built-in Classes and Functions.

6. In a test, you replace the right-hand-side operand with a value.

   Configure the **<operand>** placeholder as you would for any operand.

   For example, enter >25 into the text entry area and press **Enter** or **Return**, as shown in Figure 4-13.

Figure 4-13 Configuring the Right-hand-side Operand of a Test in a Rule



How to Define a Test in a Verbal Rule

To create a test in a verbal rule you select a derived or user-defined business phrase, or write a new user-defined business phrase, for which you supply details later.

As you enter text in a verbal rule test, the Rules Editor displays a drop down list of related business phrases.

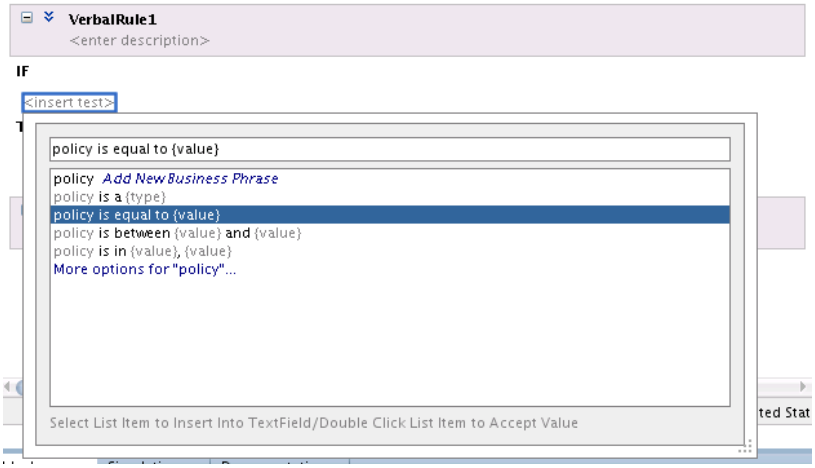
To define tests in verbal rules:

1. In Rules Designer, add a new verbal rule, or select the verbal rule you want to use.
2. The **IF** area of the rule includes a placeholder `<insert test>`, as shown in Figure 4-14.

Figure 4-14 Rule Test with Left-hand-side operand and Right-hand-side operand

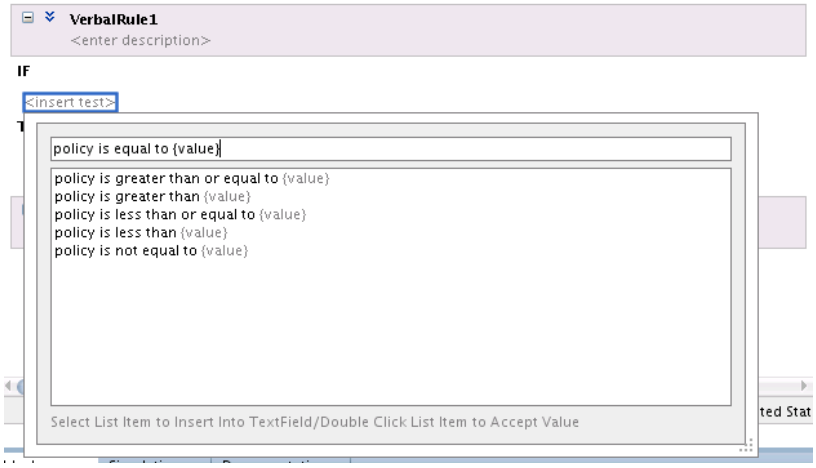
3. In the rule, click **<insert test>** and begin to type a test in text entry box that appears.
4. As you type text (such as 'policy', for example), a list of related business phrases are displayed as shown in Figure 4-15. Select the one you want.

Figure 4-15 Configuring a Test in a Verbal Rule



5. You can refine the list if needed. To display more choices, select a business phrase and press the Tab key. The list is populated with business phrases related to the one you selected, as shown below.

Figure 4-16 Refining Suggested Business Phrases in a Verbal Rule



6. If there are parameters in your business rule, such as '{value}', click on them and specify their details.

7. If you have written a new business phrase, the rule is put into draft mode. Define the business phrase in the Business Phrases tab. For more information, see How to Create Business Phrases.

What You Need to Know About Oracle Business Rules Test Variables

Oracle Business Rules test variables provide a way to shorten lengthy expressions that occur in rule and decision table conditions and actions. The variable and its value can be represented as an inline business term definition. The test variables are also called as inline aliases.

The option to insert test variables appears as a list next to **<insert test>** in the rules condition section. As part of the definition of rule condition, you can define a variable to represent a complex expression, a mathematical expression, or callouts to functions.

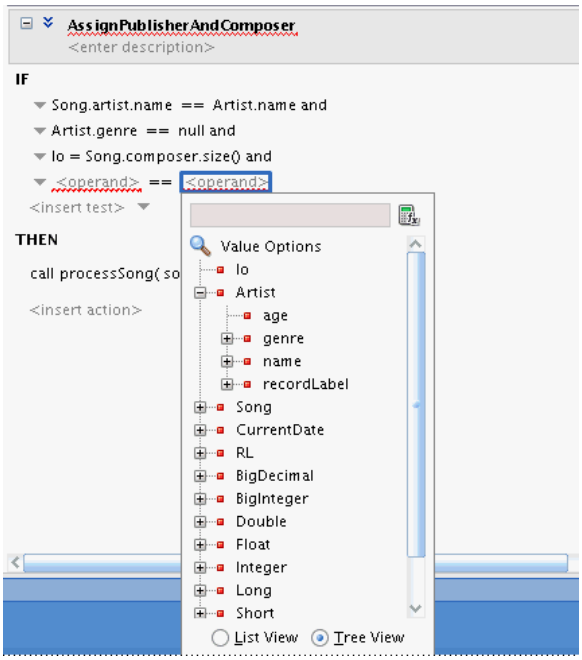
For example you have an XML fact called `Song` that has an attribute as `composer` having a function called `size`. When referring to the attribute, instead of using `Song.composer.size()` every time, you can just define a variable as the following:

```
lo = Song.composer.size()
```

Subsequently, in tests, you can use `lo` as part of your expressions. The expression can be anything from a simple to a complex expression. For example, in the body of a function, if you click **<insert action>**, you can see expression as a part of the available options.

Figure 4-17 displays a test variable.

Figure 4-17 Rules Test Variable



Once you define an inline alias, for subsequent test conditions, the inline alias is available in the list of the operands. The scope of an inline alias is restricted to the subsequent tests in a particular rule, in which the inline alias is defined. In case of a nested test, you can still use the inline alias, because the nested test is a part of the base test where you have defined the alias. This is true even for any test that you define even within the nested test. The scope of the inline alias is not just restricted to the test conditions of the base and its nested test, but also to the actions of that rule. If the inline alias is defined as a part of a nested test condition and not as a part of the main test condition, even then the alias will be available to all the subsequent test conditions and actions within or outside the main nested test.

However, if you define an inline alias inside a not nested test, then the scope of the inline alias is restricted only to the subsequent tests inside the not nested test and not to any tests that are outside the not nested test.

The inline aliases can be used both in If-Then rules as well as Decision Tables. In a Decision Table, in Advanced Mode, you can show or hide patterns as well as enter a pattern by clicking **<insert pattern>**. After you insert a pattern, you can insert tests. In normal mode, you can show or hide tests as well as enter a test by clicking **<insert test>**.

Note:

Advanced Mode capability has been maintained for backward compatibility only. We recommend that you use extended tests in simple mode to create any kind of condition that you need.

Everything that can be done in Advanced Mode can be done in simple mode. Advanced mode rules can be converted to equivalent simple mode rules simply by clearing the Advanced Mode check box.

For more information, see Working with Extended Tests

How to Define Range Tests in Rules

To create a range test in a rule, you add conditions for facts. For example, with a sample `CustomerOrder` fact with an annual spending property, you can add a test to determine if the value of a customer order falls between an upper and lower range.

The following summarizes this sample rule:

```
IF
     CustomerOrder.annualSpending between 100 and 2000
THEN
     Modify CustomerOrder.value = "Normal"
```

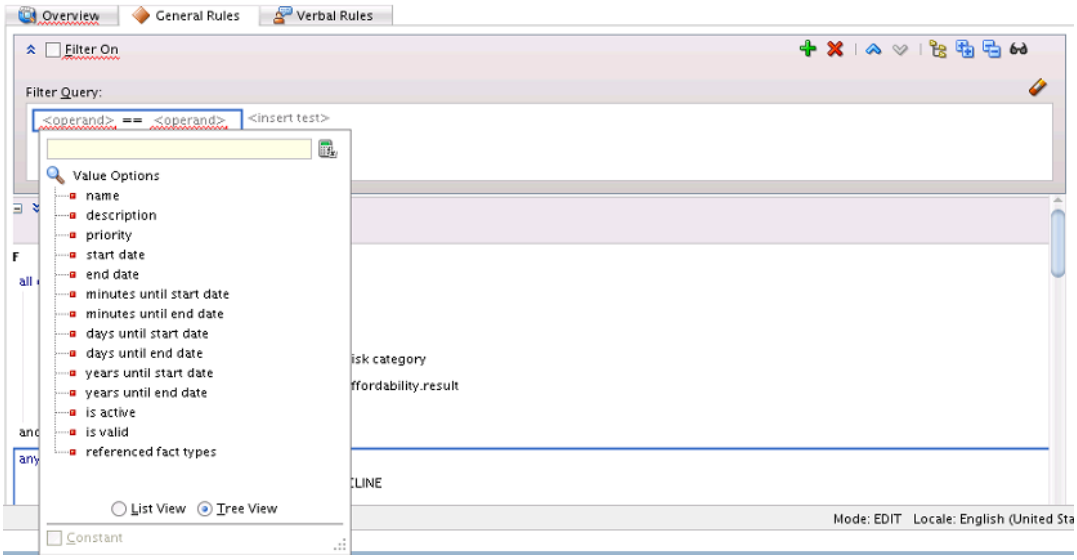
At runtime, when this rule is processed the Rules Engine checks the facts against rule pattern tests that you define to find matching facts.

To define range tests in rules:

1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. In the **View** field, select **IF/THEN Rules** (this is the Rules Designer default).
3. Add or select the rule you want to use, for example, select **Rule_1**.
4. In **Rule_1**, in the **IF** area, select **<insert test>**.
5. The test in the **IF** area of a rule includes a left-hand side **<operand>** and a right-hand-side **<operand>**.
6. In a range test, you replace the left-hand-side operand with a value.

   To do this, select the left-hand-side **<operand>**. This displays a text entry area and a list, as shown in Figure 4-18:

Figure 4-18 Adding a Test Left hand-side Operand to a Rule



   a. To enter a value, use the list to select an item from the value options.

      You can view the options using a single list, by selecting **List View**, or using a navigator by selecting **Tree View**.

b. To enter a literal value, type the value into the text entry area and press **Enter**. The value you enter must agree with the type of the corresponding operand.

   For example, in the test **IF** `CustomerOrder.annualSpending` **> <operand>**, valid values for **<operand>** must agree with the type of `CustomerOrder` field `annualSpending`.

7. In a range test, you choose the `between` operator. To do this, select the default **==** operator. This displays a text entry area and a list. Select **between** as shown in Figure 4-19.

Figure 4-19 Configuring the Operator of a Range Test in a Rule

This adds two more **<operand>** placeholders.

8. Configure the **<operand>** placeholders as you would for any operand.

   For this example, the test is true when the left-most operand (`CustomerOrder.annualSpending`) is between the values `100` and `2000`.

How to Define Set Tests in Rules

To create a set test in a rule, you add conditions for facts. For example, with a sample `CustomerOrder` fact with a line item property you can add a test to determine if the line item belongs to an arbitrary set of products.

The following summarizes this sample rule:

```
IF
     CustomerOrder.lineItem.sku in 12345, 43255, 76348
THEN
     Modify CustomerOrder.value = "High"
```

At runtime, when this rule is processed the Rules Engine checks the facts against rule pattern tests that you define to find matching facts.

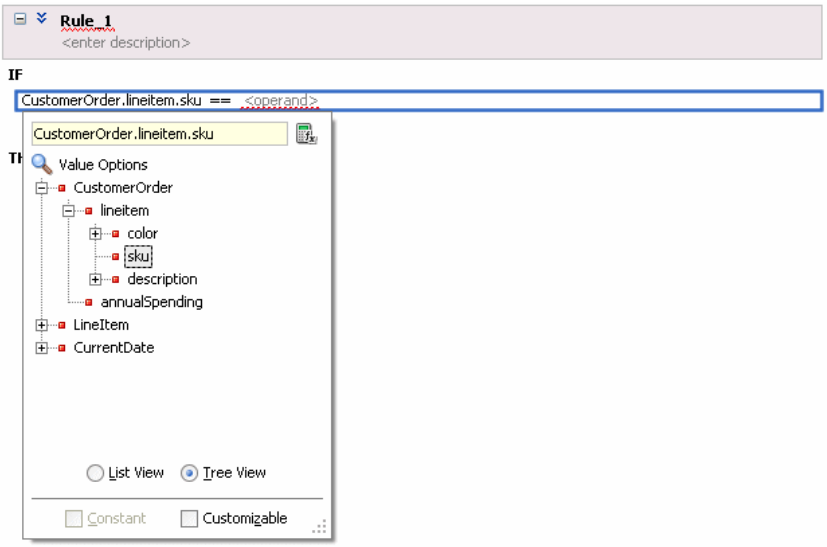
To define set tests in rules:

1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. In the **View** field, select **IF/THEN Rules** (this is the Rules Designer default).
3. Add or select the rule you want to use, for example select **Rule_1**.
4. In **Rule_1**, in the **IF** area select **<insert test>**.
5. The test in the **IF** area of a rule includes a left-hand side **<operand>** and a right-hand-side **<operand>**.

6. In a set test, you replace the left-hand-side operand with a value.

   To do this, select the left-hand-side **<operand>**. This displays a text entry area and a list as shown in Figure 4-20:

Figure 4-20 Adding a Test Left-hand-side Operand to a Rule



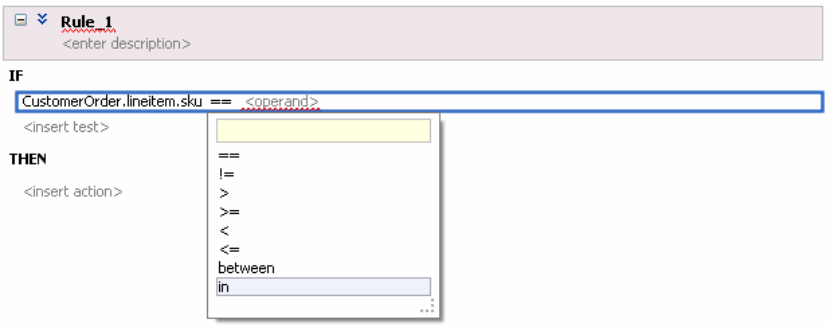
   a. To enter a value use the list to select an item from the value options.

      You can view the options using a single list, by selecting **List View**, or using a navigator by selecting **Tree View**.

   b. To enter a literal value, type the value into the text entry area and press **Enter**.

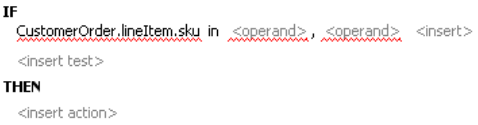
7. In a set test, you use the `in` operator. To do this, select the default **==** operator. This displays a text entry area and a list. Select **in** as shown in Figure 4-21.

Figure 4-21 Configuring the Operator of a Set Test in a Rule



This adds two more `<operand>` placeholders in a comma separated list and an `<insert>` placeholder as shown in Figure 4-22.

Figure 4-22 In Operator in a Set Test

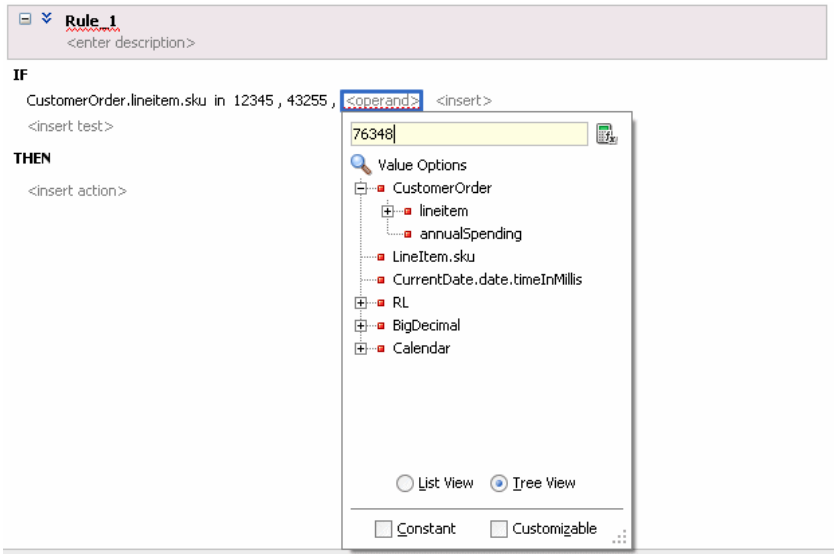


To add another operand to the list, click **<insert>**.

To delete an operand from the list, right-click the operand and select **Delete Test Expression**.

8. Configure the `<operand>` placeholders as you would for any operand as shown in Figure 4-23.

Figure 4-23 Configuring the Operands of a Set Test in a Rule



The test is true when the value of the left-most operand (`CustomerOrder.lineItem.sku`) is any of 12345, 43255, or 76348.

How to Define an Action in a General Rule

To create a rule you insert tests and you insert actions. The actions are associated with pattern matches. When a test in the **IF** area of a rule matches, the Rules Engine activates the **THEN** action and prepares to run the actions associated with the rule.

When you add an action, you use one of the forms of actions shown in Table 4-2. For each form shown in Table 4-2 the options that Rules Designer presents are context sensitive, so the lists and the number of items you work with may be different, depending on which action you add and the choices you make while you enter the action. Table 4-2 shows the basic actions; additional actions are available with Advanced Mode. For more information on advanced mode see Using Advanced Settings with Rules and Decision Tables.

To define actions in general rules:

1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. In a general rule, in the **THEN** area, select **<insert action>**. This displays the add action list as shown in Figure 4-27.

Figure 4-24 Adding a Modify Action to a Rule

3. In the add action list, select the type of action you want to add. For example, select **modify**. You can also enter the name of the action in the text area. As you begin entering a name, the list of available choices is automatically filters. This is useful when there are a large number of options available.

   You can add any required action ranging from `assert`, `call`, `modify` to even conditional actions such as `if`, `else`, `elseif`, `while`, `for`, `if (advanced)`, and `while (advanced)`.

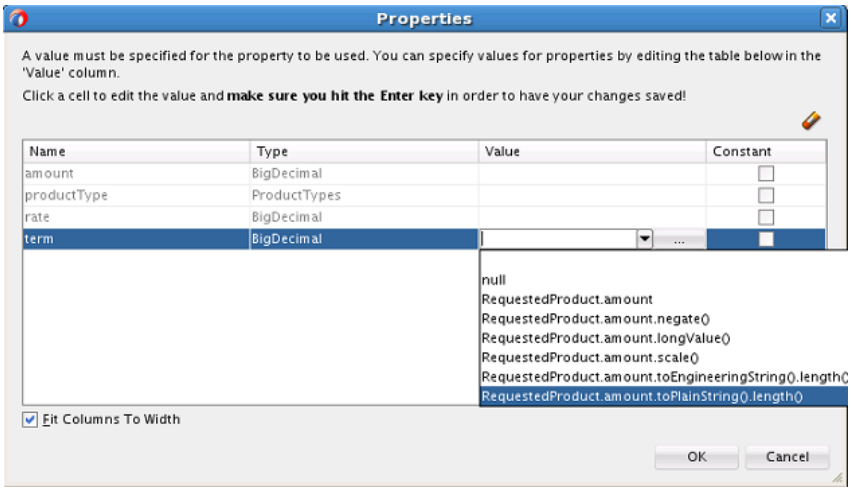
4. In the **THEN** area, select **<target>** to display the option list. For example, select `RequestedProduct` as shown in Figure 4-25.

Figure 4-25 Adding Modify Action to a Rule and Selecting the Target

5. Select **<edit property>**. This displays the Properties dialog.

6. In the Properties dialog, in the **Value** column, enter `"High"` (include the double quotation marks) and press **Enter** or **Return** as shown in Figure 4-26.

Figure 4-26 Adding Modify Action Property and Value to a Rule



7. In the Properties dialog, click **Close**. This displays the rule.

Basic Actions in a General Rule

Table 4-2 Rule Action Choices

Action Form	Description
assert New	Assert a new fact.
`assign,`	Assert a new fact.
`call`	Call a function.
`modify`	Modify a data value associated with a matched fact.
`retract`	Retract a fact.
`assert`	Assert a fact.
`asset tree`	Asserts a tree of facts given the root.
`assign new`	Assign a new fact.
`expression`	Perform expression.
`return`	The return action returns from the action block of a function or a rule. A return action in a rule pops the ruleset stack, so that execution continues with the activations on the agenda that are from the ruleset that is currently at the top of the ruleset stack.
`RL`	Use an Oracle RL expression that you supply.
`synchronized`	The synchronized action is useful for synchronizing the actions of multiple threads. The synchronized action block lets you acquire the specified object's lock, then execute the action-block, then release the lock.
`throw`	Throw an exception, which must be a Java object that implements java.lang.Throwable. A thrown exception may be caught by a catch in a try action block.

Table 4-2 (Cont.) Rule Action Choices

Action Form	Description
`try`	The try, catch, and finally in Oracle RL is like Java both in syntax and in semantics. There must be at least one catch or finally clause.
`if`, `else`, `elseif`, `for`, `while`	Conditional actions.

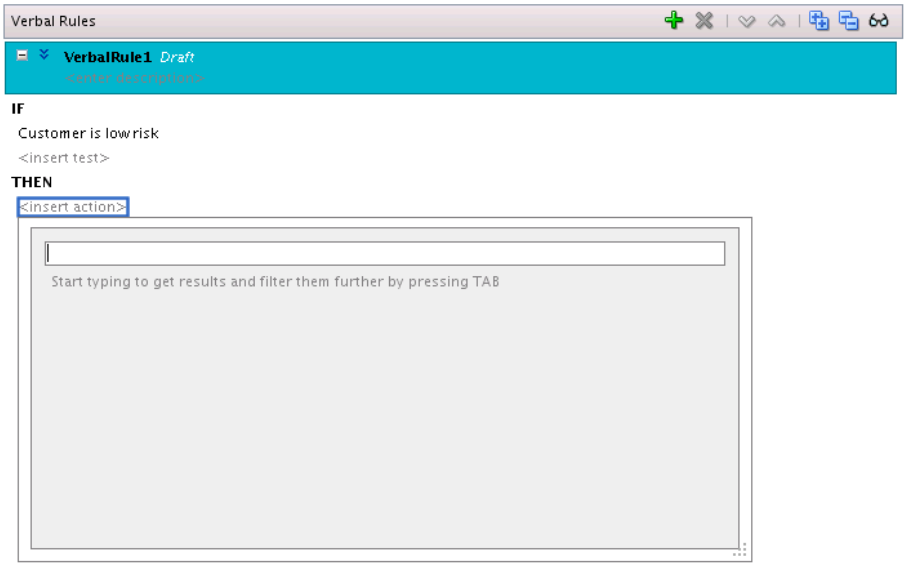
How to Define an Action in a Verbal Rule

Like general rules, to create a verbal rule you insert tests and actions. Verbal rules tests and actions are composed primarily from business phrases.

To define actions in verbal rules:

1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. In a verbal rule, in the **THEN** area, select **<insert action>**. This displays the add business phrase list as shown in Figure 4-27.

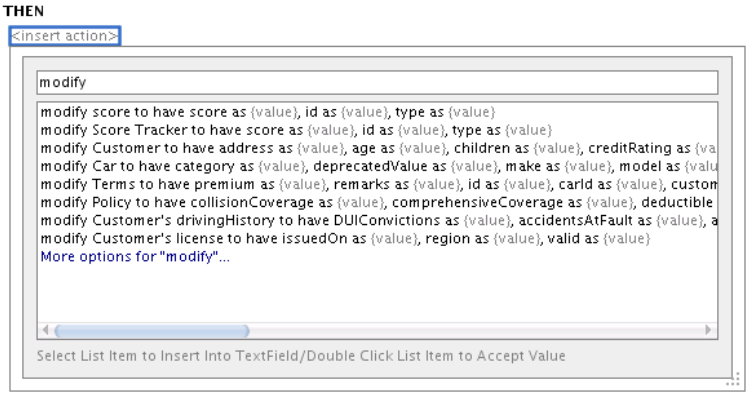
Figure 4-27 Adding an Action to a Verbal Rule



3. In the business phrases list, start to type the action you want to display a list of suggested business phrases.

   You can also type keywords if you aren't certain of how to phrase your action. For example, if you know you want to calculate a premium in a particular way, you might type 'calculate Premium' to see related business phrases.

Figure 4-28 Adding an Action to a Verbal Rule

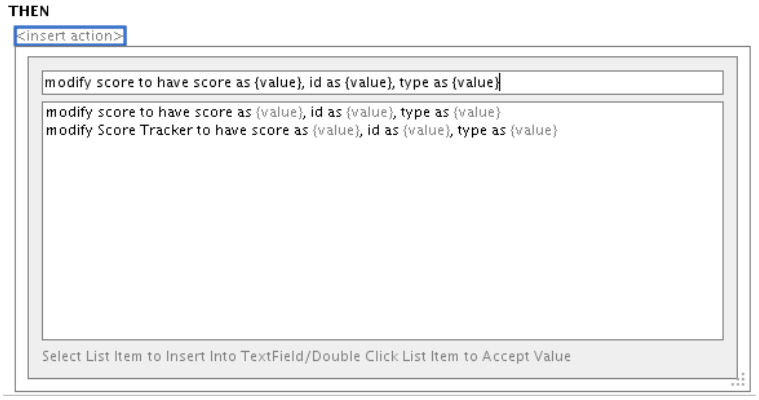


4. Select a business phrase if one is available that meets your needs.

5. To refine the list of business phrases further, select one related to what you want to use and press the Tab key.

   The list is displayed with a refined set of business phrases. Select the phrase you want.

Figure 4-29 Adding an Action to a Verbal Rule



6. If no business phrases in the list meet your needs, type a business phrase and select Add New Business Phrase to instantiate a new business phrase. Complete the definition of the business phrase in the Business Phrases tab.

What You Need to Know About Rule Actions

A rule loop occurs when the value for a condition is changed by an action. Loops can occur across rules in a single rule, spread over several Decision Tables, or spread over rules and Decision Tables in the same ruleset. You need to avoid creating rule actions that modify fact properties that are used in rule conditions. At runtime, such rules could cause an infinite loop.

What You Need to Know About Oracle Business Rules Performance Tuning

In most cases, writing of rules should not require a focus on performance. However, there are tips that can that help you to enhance and maximize rule performance.

For more information on Oracle Business Rules performance tuning, see Oracle Business Rules Performance Tuning in *Tuning Performance*.

Introduction to Verbal Rules and Business Phrases

Verbal rules work hand in hand with business phrases to provide a flexible way author rules using natural language statements to express rule logic in domain specific sentences that are similar to spoken language.

Business phrases provide the logic behind conditions that are used in the composition of the verbal rule.

You can write verbal rule tests and actions using derived business phrases as well as user-defined business phrases. Derived business phrases are automatically created using facts, globals and other information in the dictionary while user-defined phrases can be explicitly authored to augment derived phrases. Further, user-defined phrases can either be pre-created or created as needed while composing the verbal rule.

As you write a verbal rule, you can use suggested business phrases, or instantiate your own on the fly and provide their implementation details later. Alternatively, you can create the business phrases you need for your verbal rule first, and then complete the verbal rule.

Working with Business Phrases

You create business phrases in the Business Phrases tab of the Rules Designer.

Business phrases comprise three parts:

- Parameters - parameters defining the types of variables that can be passed to the business phrase
- Value - the business phrase expression, including placeholders for parameters if any
- Mapping - definitions of the logic defining the business phrase conditions

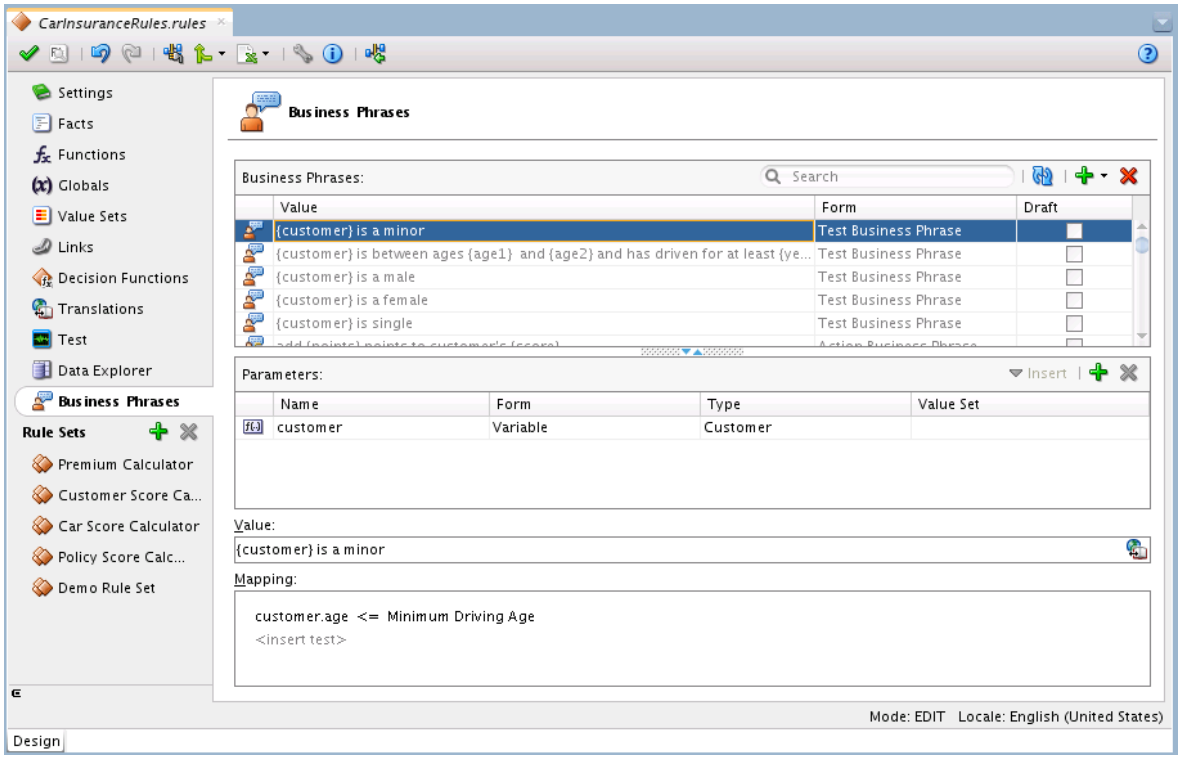
There are two types of business phrases:

- Test business phrases - define conditions. These provide the same types of logic as the IF part of a general rule. For more information, see How to Define a Test in a Rule.
- Action business phrases - define the actions to perform if the conditions defined by the test business phrases in a verbal rule are met. These provide the same types of logic as the THEN part of a general rule. For more information, see How to Define an Action in a Verbal Rule.

Business Phrases Tab

You create both test and action business phrases in the Business Phrases tab of the Rules Designer, as shown in Figure 4-30.

Figure 4-30 Business Phrases Tab



The tab includes the following sections:

- Business Phrases list
- Parameters
- Value
- Mapping

Business Phrases list

The Business Phrases list displays the business phrases included in the dictionary.

Use the toolbar controls to filter the list by searching, to refresh the list, to add new test or action business phrases, and to delete the currently selected business phrase.

The list displays business phrases and their attributes: **Value**, **Form** and **Draft**.

Mark a business phrase as draft by checking the Draft check box directly in the list.

Business phrases containing validation errors are marked with a red squiggly underscore. Hover over them to see the error in a pop-up.

Parameters

Use the Parameters panel to view and edit parameters.

Click **Insert** to add a parameter to the value of the business phrase. Click **Add** and **Delete** to create and remove parameters.

The Form attribute defines the type of parameter. Choices include:

- Value - ad hoc value. When selected, the Type can be chosen from boolean, byte, char, double, float, int, long, short or String

- Variable - a variable which is already defined within the scope of the business phrase. When selected, the Type can be chosen from one of the defined fact types in the dictionary.
- Expression - enter an expression

Value

Edit the definition of the business phrase in the Value panel. The value is also used as the display name of the business phrase in the Business Phrases list, and in business phrases displayed in choice lists when authoring a verbal rule.

Mapping

Edit the logical definition of conditions for the business phrase in the Mapping panel. A business phrase mapping contains similar logical constructs to what you would see if the business phrase logic were authored as a general rule. See the discussions of tests and actions in Working with Rules for principles and procedures which also apply to the creation of business phrase mappings.

Draft Business Phrases and Verbal Rules

Business phrases can be marked as being in draft status.

You can set or override the draft status of a business phrase by checking or unchecking **Draft** in the Business Phrases list.

The draft status of a verbal rule is derived from the business phrases it references and can not be manipulated directly. If a verbal rule contains business phrases marked draft, the rule is also marked draft. The verbal rule description panel is changed to a solid blue color, and the word 'Draft' appears next to the rule name, as shown in Figure 4-31. When all business phrases referenced by the verbal rule are no longer marked draft, the verbal rule is taken out of draft status.

Figure 4-31 Verbal Rule Marked Draft

Draft business phrases and verbal rules are not validated and are not included in the dictionary for execution. This allows you to continue to use or test a dictionary as you refine your business phrases and verbal rules.

As you write a verbal rule you can compose business phrases that do not yet exist in the dictionary. These are automatically added to the list of business phrases and marked draft, and the verbal rule is marked draft as well.

How to Create Business Phrases

You create business phrases in the Business Phrases Tab. You can also specify a business phrase while writing a verbal rule and then complete its definition later in the Business Phrases tab.

Use the Business Phrases tab to add, modify, and delete business phrases.

To Create a New Business Phrase

1. In Rules Designer, select the **Business Phrases** tab and click **Create** (+) to create a test business phrase. Select either **Test Business Phrase** or **Action Business Phrase**.

A new business phrase is created.

2. Enter the definition of the business phrase in the **Value** panel. Placeholders for parameters that have not yet been defined can be included by typing their name wrapped in curly braces. For example:

   ```
   {customer} is single
   ```

3. Define parameters in the **Parameters** panel. Click Create (+) to add a new parameter. Double-click its name to edit it. Specify its Form, and Type. Optionally, specify a Value Set.

4. To add a parameter to the business phrase value, click **Insert**.

5. Define the mapping for the business phrase in the **Mapping** panel. Begin by clicking **<insert test>**. Select tests and specify operands. Add additional tests if needed.

Example Business Phrase Creation Scenario

For this example, assume you have an Insurance Quote project with most of the project definitions complete. Perhaps you want add a business phrase that tests to see if a customer is a minor, and to invalidate the policy.

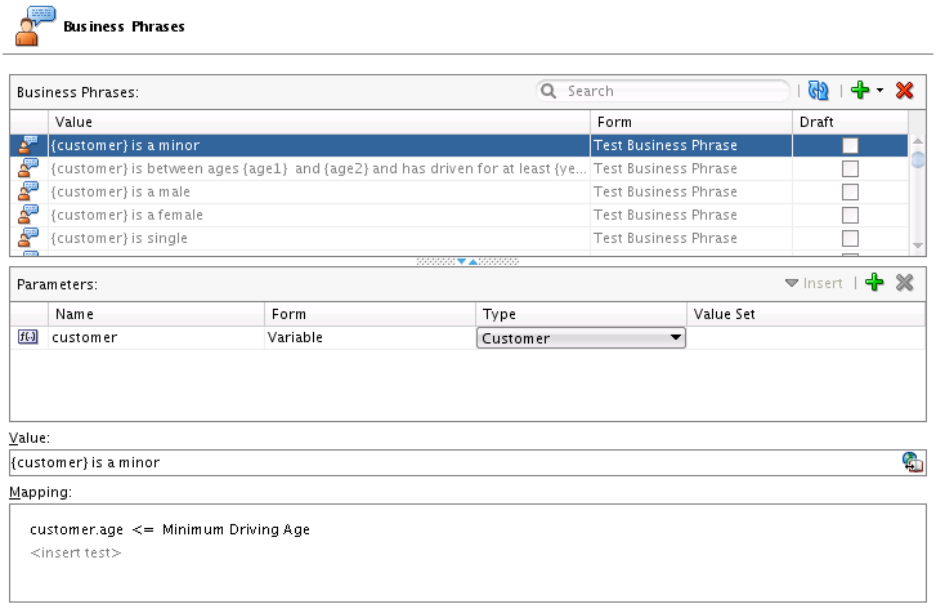
You create a new test business phrase and provide the value **{customer} is a minor**.

Next, you define the parameter customer, and map it to your previously defined Customer fact.

Now you provide the mapping that specifies the condition. You click on <insert test> and select simple test. You click on the left <operand>, expand customer and select customer.age. You click on the right operand and specify the value 21.

The test business phrase looks like Figure 4-32 below.

Figure 4-32 Test Business Phrase Example



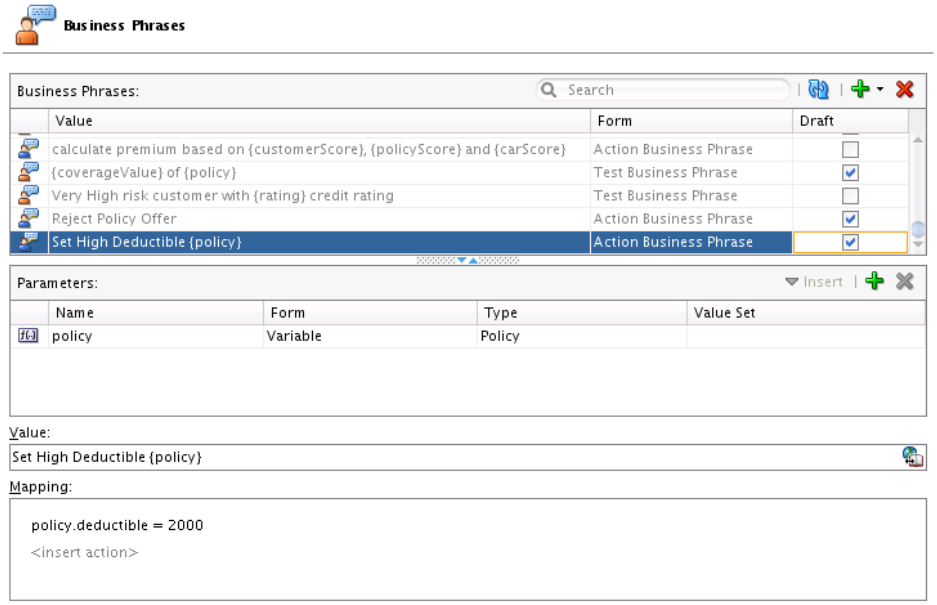
Now you create an action business phrase to set the deductible to a high value and give it the value **Set High Deductible**.

You create a variable called **policy** and map it to your previously defined Policy fact.

You click **<insert test>** in the **Mapping** panel and select **Expression**. You click **<expression>** and the **Expression Editor** displays. You navigate to **policy.deductible**, select it, and click **Insert into Expression**. You complete the expression with '= 2000' and click OK.

Your action business phrase looks like the Figure 4-33 below.

Figure 4-33 Action Business Phrase Example



Translating Business Phrases

The Value attribute of Business phrases that have been added to the dictionary can be translated, regardless of whether they originated as derived business phrases or as user-defined business phrases.

In the Business Phrases tab, select the business phrase to translate and click **Edit Translation Bundles** in the Value panel. Edit translations in the **Bundle Editor** dialog that appears.

Choosing or Adding Business Phrases in Verbal Rules

Verbal rules use business phrases to specify the IF and THEN tests and actions.

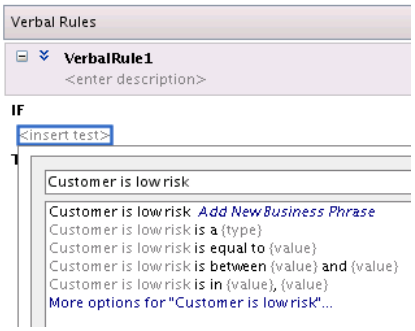
When defining a test or action in a verbal rule, you enter text which triggers a drop-down list of choices. From the list, you can select existing business phrases from the dictionary, automatically generated business phrases, or you can instantiate your a new business phrase based on what you typed, provide its implementation details later in the Business Phrases tab.

Instantiating New Business Phrases While Authoring a Verbal Rule

You can instantiate new business phrases while authoring a new verbal rule simply by typing them into a test or action instead of selecting one from the drop down list. These business phrases are marked draft, and the verbal rules which use them are also marked as draft.

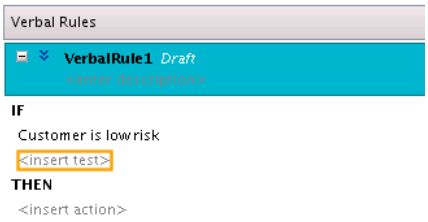
In the example below, the desired business phrase **Customer is low risk** is entered as a test. The business phrase is not shown in the drop down list. It was not automatically generated, and has not previously been defined in the dictionary.

Figure 4-34 Adding a New Business Phrase to a Rule



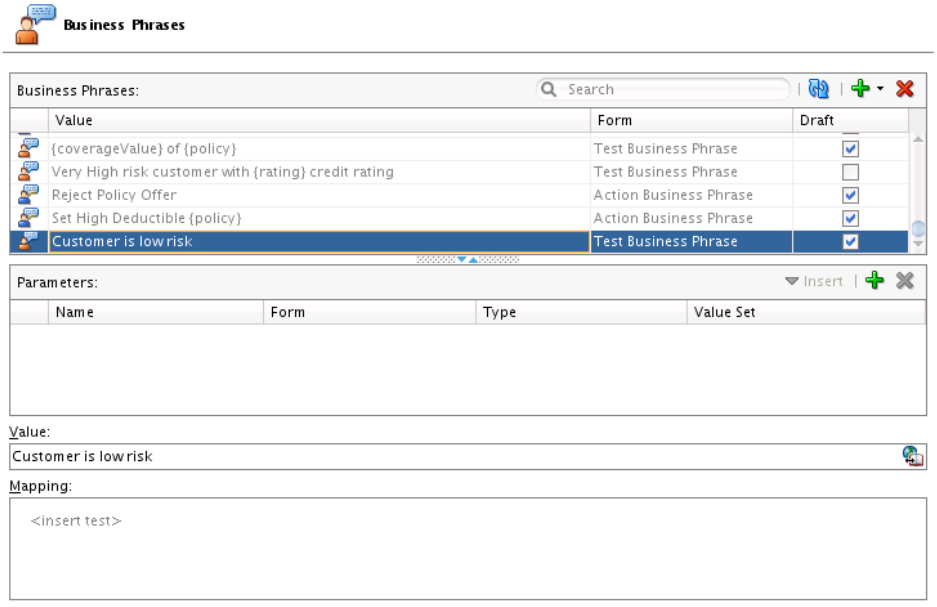
The verbal rule is marked as a draft.

Figure 4-35 New Business Phrase Added and Rule is Marked Draft



The business phrase marked as a draft and is added to the Business Rules list. The parameters (if any) and mapping information must still be specified.

Figure 4-36 Business Phrases Added From Verbal Rule Marked Draft



Choosing Business Phrases While Creating a Verbal Rule

A robust list including both previously user-defined and auto-generated derived business phrases, sorted by relevancy, is automatically provided as you author a test or action.

User-defined and derived business phrases are not visually distinguished from one another in the drop down list

Derived Business Phrases

Derived business phrases are automatically created and are based on business objects and data model as defined in the dictionary based. These are created on the fly and based on what the system calculates you intend to author, based on your typed input. These are not persisted if they are not added to the verbal rule.

Derived business phrases, once added to a verbal rule, are just normal business phrases and support parameters and translation.

Choosing Which Business Phrases to See in the List

Use the Settings tab > Dictionary Settings > Phrase Suggestions > Value drop down to control the types of business phrases seen in the drop down pick lists shown while authoring verbal rules' tests and actions.

Choices include:

- All - display both user-defined business phrases in the dictionary, and derived business phrases
- Auto Suggestions - display only derived business phrases
- Business Phrases - display only user-defined business phrases from the dictionary

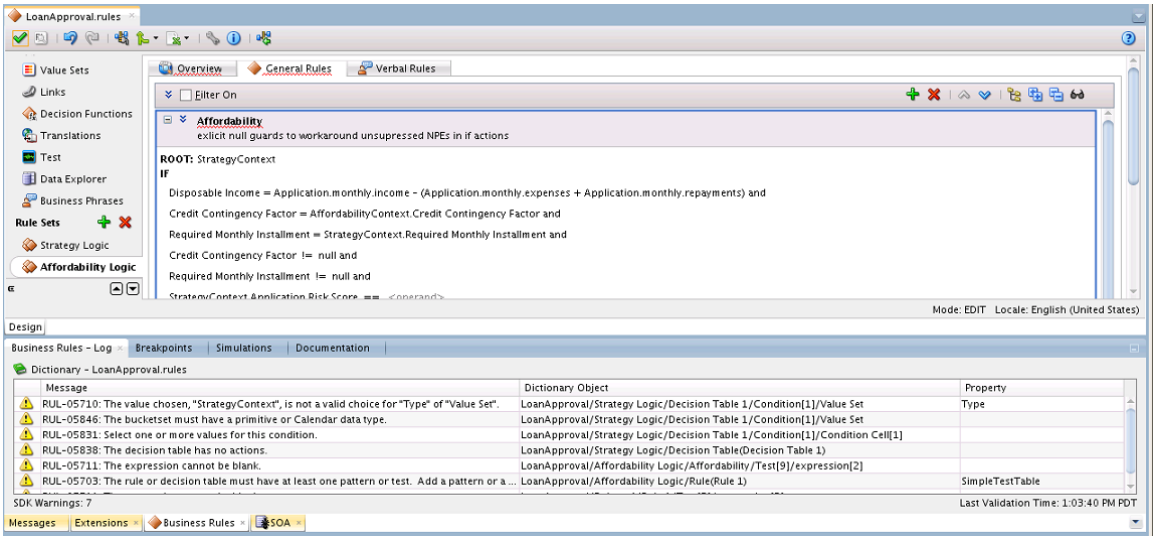
Validating Dictionaries

Rules Designer performs dictionary validation when you make any change to the dictionary. Rules Designer validation can assist you when you work with rules or Decision Tables.

To show the validation log window, click the **Validate** button or select **View>Log** and select the **Business Rule Validation** tab. This displays warnings for incorrect or incomplete rules. Note that you must correct all warnings before you can test or deploy rules.

When a dictionary is invalid, Rules Designer produces a list of warning messages and lists the associated dictionary objects. You can use the validation message information to locate the dictionary object and to correct problems. In addition, Rules Designer flags objects with validation warnings with a validation indicator (a red, wavy underline), as shown in Figure 4-37.

Figure 4-37 Validation Warnings Shown in Log and On Screen with Wavy Underline



If a dictionary is invalid, you can save the dictionary. However, you can only generate RL Language for a dictionary that is valid and does not display warnings in the Rules Designer validation log.

In the validation log, each validation message includes the following:

- Message: The message provides details on the Oracle Business Rules exception that describes the problem.
- Dictionary Object: This field displays a path that indicates details that should allow you to identify a component in the dictionary.
- Property: provides information on a property of the object associated with the warning message.

When you are viewing the validation log, if you select an item and then right-click and select from the list **Select and Highlight Object in Editor**, Rules Designer moves the cursor to select the dictionary object. Note that for some validation warnings this functionality is not possible.

Understanding Data Model Validation

Rules Designer performs dictionary validation when you make any change to the dictionary. When Rules Designer displays a warning message, the validation log includes a message that should assist you in locating the dictionary object that caused the validation warning. For example, the following string indicates that the warning originates from the data model object named `RLFact_1`. In addition, the problem is in the property named `test_int`:

```
CarRental/Data Model/RLFact_1/test_int/Expression
```

Table 4-3 specifies the parts of the dictionary object name specified in a validation message.

Table 4-3 Data Model Dictionary Property in Validation Log

Name	Description
`CarRental`	Dictionary Name

Table 4-3 (Cont.) Data Model Dictionary Property in Validation Log

Name	Description
`Data Model`	Data Model component in dictionary.
`RLFact_1`	Element name in data model
`test_int`	Property name in the specified element.
`Expression`	Expression part of property.

For more information, see:

- Understanding Rule Validation
- Understanding Decision Table Validation
- How to Validate a Dictionary

Understanding Rule Validation

When you click the **Validate** button Rules Designer displays the validation log. When you first add a rule you see validation warnings similar to those shown in Figure 4-38.

Figure 4-38 Business Rules - Log Validation Messages for a New Rule

The dictionary object name part of a validation message for a rule includes details that help you to identify the ruleset, the rule, and an area in the rule that is associated with the validation warning. For example, the following dictionary object specification indicates a problem:

```
OracleRules1/Ruleset_2/Rules_1/Pattern[1]
```

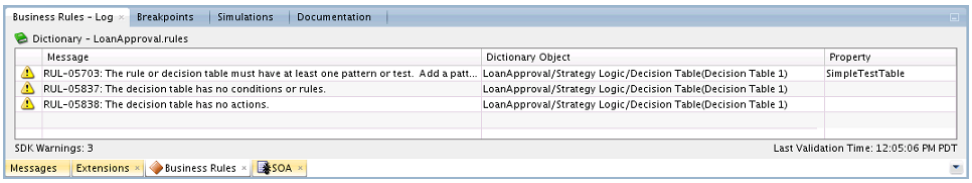
In validation messages, the dictionary object name for a rule uses indexes that start at 1. Thus, the first pattern is `Pattern[1]`.

In addition to validating rules, you can also test them in Rules Designer as you are designing them. For more information, see Testing Decision Functions Using a Rules Function.

Understanding Decision Table Validation

When you click the **Validate** button Rules Designer displays the validation log. When you first add a Decision Table you see validation warnings similar to those shown for a new rule, as in Figure 4-38.

Figure 4-39 Business Rules - Log Validation Messages for a New Decision Table



The dictionary object name part of a validation message for a Decision Table includes details that help you to identify the area of the Decision Table that is associated with the validation warning. For example, the following dictionary object specification indicates a problem in the first action row, and the first action cell of the Decision Table:

```
OR1/Ruleset_1/DecisionTable_1/Action[1]/Action Cell[1]
```

In validation messages the dictionary object name for a Decision Table object uses indexes that start at 1. For example, to indicate the first condition cell in the first row in the **Conditions** area, the message is as follows:

```
OracleRules1/Ruleset_1/DecisionTable_2/Condition[1]/Condition Cell[1]
```

This specification indicates the condition cell for the rule with the label R1 in the first row of the **Conditions** area in a Decision Table.

How to Validate a Dictionary

Rules Designer performs dictionary validation when you make any change to the dictionary.

To validate a dictionary:

1. In Rules Designer, click the **Validate** button (a checkmark).
2. Select the Business Rules - Log from the messages area.
3. When you are viewing the validation log, if you select an item and then right-click and select from the list **Select and Highlight Object in Editor**, Rules Designer moves the cursor to select the dictionary object. Note that for some validation warnings this functionality is not possible.

Using Advanced Settings with Rules and Decision Tables

Advanced settings for rules and Decision Tables allow you to work with features that provide advanced options that not all Oracle Business Rules users need.

Advanced settings features are shown in Figure 4-40:

Figure 4-40 Show/Hide Advanced Settings

These features include:

- **Advanced Mode:** allows additional pattern matching options and nested tests in rules. Only use Advanced Mode if you have used it before. We recommend that you use extended tests in simple mode to create any kind of condition that you need.

  For more information, see:

  – How to Show and Hide Advanced Settings in a Rule or Decision Table

  – How to Select the Advanced Mode Option

  – Working with Advanced Mode Rules

- **Simple Mode:** has been updated and should be used when building complex rules. Only use Advanced Mode if you have used it before. Advanced Mode capability has been maintained for backward compatibility only.

  For more information, see Working with Extended Tests.

- **Tree Mode:** makes it easier to work with master detail hierarchy, nested elements that map to a parent child relationship. These parent child relationships among facts are common with XML and ADF Business Components fact types. You can use this option with the **Advanced Mode** option.

  For more information, see How to Create Simple Tree Mode Rules.

- **Rule Active:** specifies that a rule or Decision Table is active or inactive. When **Rule Active** is cleared, Rules Designer does not validate the specified rule or Decision Table.

  For more information, see How to Select the Active Option.

- **Logical:** allows you to enable or disable logical dependence between the facts that trigger a rule and the facts asserted by a rule.

  For more information, see How to Select the Logical Option.

- **Allow Gaps** (available only with Decision Table advanced settings). This check box determines if validation messages are reported when gaps are detected in a Decision Table. The specific validation message is:

  ```
  RUL-05852: Decision Table has gaps
  ```

  For more information, see Understanding Decision Table Gap Checking and How to Perform Decision Table Gap Checking.

- **Priority:** specifies the priority for a rule or a Decision Table. Higher priority rules run before lower priority rules, within a ruleset.

  For more information, see How to Set a Priority for a Rule.

- **Conflict Policy:** (available only with Decision Table advanced settings). Specifies the Decision Table conflict policy, one of the following:

  – **manual** conflicts are resolved by manually specifying a conflict resolution for each conflicting rule.

  – **auto override** conflicts are resolved automatically using an override conflict resolution when this is possible, using the automatic conflict resolution policies.

  – **ignore** conflicts are ignored.

For more information, see Understanding Decision Table Conflict Analysis.

- **Effective Date:** specifies effective dates for a rule or a Decision Table.

  For more information, see How to Specify Effective Dates.

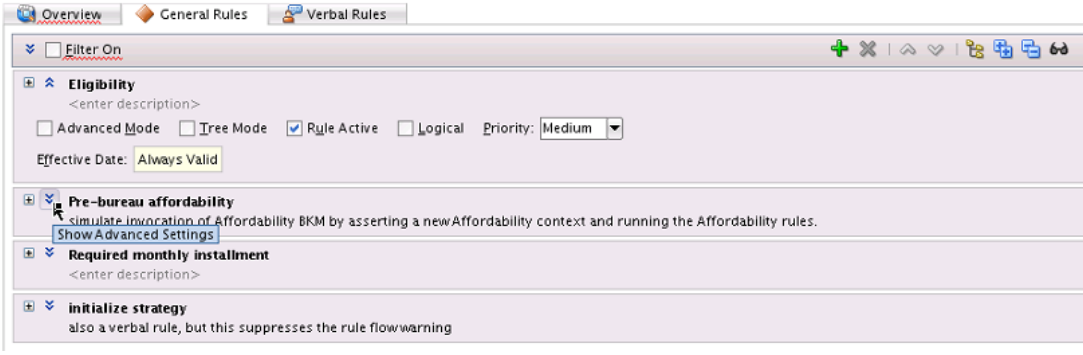
How to Show and Hide Advanced Settings in a Rule or Decision Table

In Rules Designer, next to each rule name and Decision Table name, the show or hide advanced settings button lets you show and hide advanced settings.

To show and hide advanced settings in a rule or decision table:

1. Select the ruleset where you want to show advanced settings.
2. In the **View** field, from the list, select either **IF/THEN Rules** or select a Decision Table.
3. **a.** To show the advanced settings, next to the rule name click **Show Advanced Settings**, as shown in Figure 4-41 (there is a highlighted button shown next to the rule name).

   Figure 4-41 Show or Hide Advance Settings

   b. To hide the advanced settings, next to the rule name click **Hide Advanced Settings**.

How to Select the Advanced Mode Option

Select Advanced Mode to use Rule or Decision Table features that provide additional pattern matching options and additional actions. For more information, see Working with Advanced Mode Rules.

To select the advanced mode option:

1. Select the rule or Decision Table where you want to set Advanced Mode.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
3. Select the **Advanced Mode**.

Figure 4-42 and Figure 4-43 are examples of a rule displayed in Advanced versus Simple Mode.

The same forms look different in Advanced Mode and Simple Mode due to the presence and absence of "Patterns."

Figure 4-42 Advanced Mode Checked

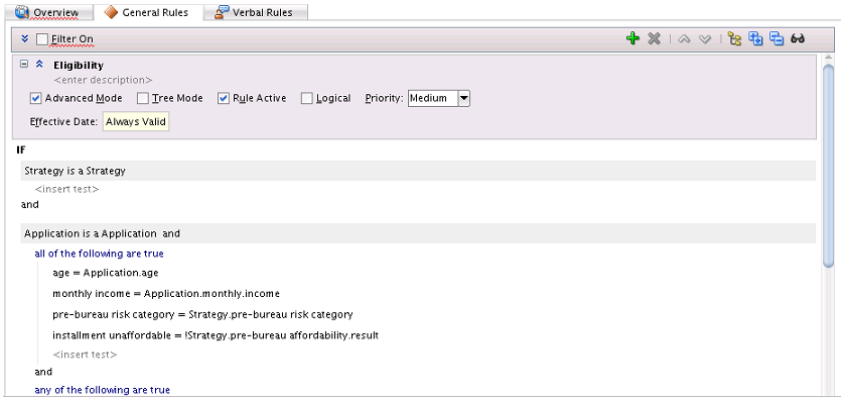
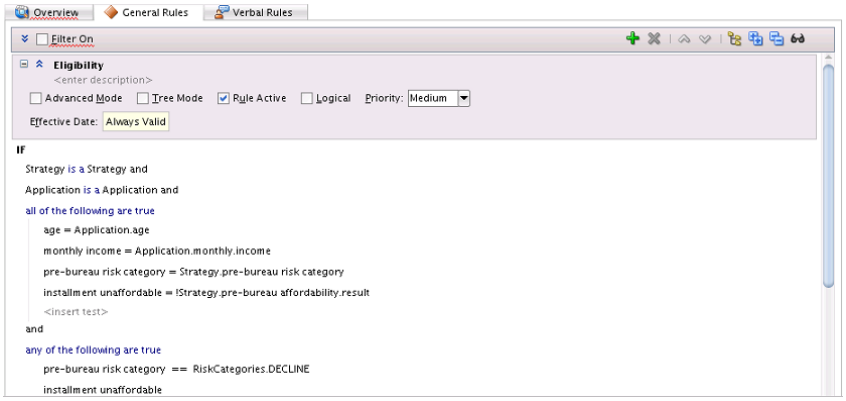


Figure 4-43 shows the same rule with Advanced Mode cleared:

Figure 4-43 Advanced Mode Cleared



How to Select the Active Option

Oracle Business Rules includes the ability to specify that a rule or a Decision Table is active or inactive. The active option is set independent of the effective dates and may be set without changing or removing previously specified effective dates. When **Rule Active** is cleared, Rules Designer does not validate the rule.

To select the active option:

1. Select the rule or Decision Table where you want to set the **Rule Active** option.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
3. Select **Rule Active**.

How to Select the Logical Option

A ruleset or Decision Table with the **Logical** option selected specifies that rules in the generated RL Language use the logical property. The logical property allows you to enable or disable logical dependence between the facts that trigger a rule and the facts asserted by a rule.

A rule with the logical property enabled makes all facts that are asserted by an action block in the rule dependent on facts matched in the rule condition. Anytime a fact

referenced in the rule condition changes, such that the rule's conditions no longer apply, the facts asserted by the rule condition are automatically retracted. For more information, see Rule Definitions in the *Rules Language Reference forOracle Business Process Management*.

Using the ruleset and Decision Table **Logical** option you can enable or disable the logical property for the generated RL Language associated with the rules in the ruleset or the Decision Table. By default, the **Logical** option is not selected.

To select the logical option:

1. Select the rule or Decision Table where you want to set the **Logical** option.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
3. Select **Logical**.

How to Set a Priority for a Rule

You can set the priority for a rule or a Decision Table. You can select from a predefined named priority list as shown in Table 4-4, or enter a positive or negative integer to specify your own priority level. Higher priority rules run before lower priority rules, within a ruleset. The default priority is `medium` (with the integer value 0).

Table 4-4 Priority String Value Mapping

Named Priority	Integer Value
highest	3000
higher	2000
high	1000
medium (Default Priority)	0
low	-1000
lower	-2000
lowest	-3000

To set a priority for a rule:

1. Select the rule or Decision Table where you want to set the priority.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
3. In the **Priority** field, specify the priority value:
   a. To specify a named priority, select a named priority from the **Priority** list.
   b. To specify an integer priority, click in the Priority field and enter a positive or negative integer value and press **Enter**.

How to Specify Effective Dates

You can specify effective dates for a ruleset, a rule, or a Decision Table.

To specify effective dates:

1. Select the rule or Decision Table where you want to set the effective date.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
3. Select the **Effective Date** field. This displays the Set Effective Date dialog.
4. Use the Set Effective Date dialog to set the effective date.

For more information on using effective dates, see Using Date Facts_ Date Functions_ and Specifying Effective Dates and How to Set the Effective Date for a Rule Set.

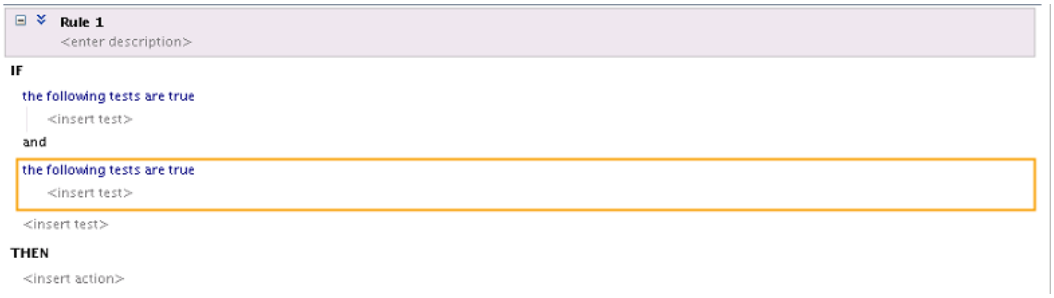
Working with Nested Tests

In a rule or a Decision Table you can create more complicated tests using the nested tests feature.

To use nested tests:

1. Select the rule where you want to use a nested test.
2. In the **IF** area, click and select Nested Test.
3. With a test selected right-click to display the list, as shown in Figure 4-44.

Figure 4-44 Adding a Nested Test to a Rule



4. Complete the test as necessary.

Working with Advanced Mode Rules

Oracle Business Rules provides features that allow you to create advanced rules that add support for the Oracle Business Rules feature.

Note:

Advanced Mode capability has been maintained for backward compatibility only. We recommend that you use extended tests in simple mode to create any kind of condition that you need.

Everything that can be done in Advanced Mode can be done in simple mode. Advanced mode rules can be converted to equivalent simple mode rules simply by clearing the Advanced Mode check box.

For more information, see Working with Extended Tests.

Oracle Business Rules provides features that allow you to create advanced rules that add support for the following Oracle Business Rules features:

- Additional Pattern Match options (see How to Use Advanced Mode Pattern Matching Options)
- Additional Matched Fact Naming options (see How to Use Advanced Mode Matched Fact Naming)
- Additional Supported Action forms (see How to Use Advanced Mode Action Forms)
- Pattern Match Aggregate Function options (see How to Use Advanced Mode Aggregate Conditions)

For more information, see What You Need to Know About Advanced Mode Rules.

How to Use Advanced Mode Pattern Matching Options

The advanced mode pattern matching options specify when a rule should fire. Table 4-5 shows the available options.

Table 4-5 Advanced Mode Pattern Matching Options

Option	Description
`for each case where`	This is the default pattern matching option. A rule should fire each time there is a match (for all matching cases).
`there is a case where`	This option selects one firing of the rule if there is at least one match.
`there is no case where`	The value specifies that the rule fires once if there are no such matches.
`aggregate`	This specifies an aggregate function is applied to all matches. For more information, see How to Use Advanced Mode Aggregate Conditions.

To use advanced mode pattern matching options:

1. Select the rule or Decision Table where you want to use pattern matching options.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).

3. Select **Advanced Mode**.
4. Right-click a test pattern and select **Surround With...** as shown in Figure 4-45.

Figure 4-45 Surrounding with an Option

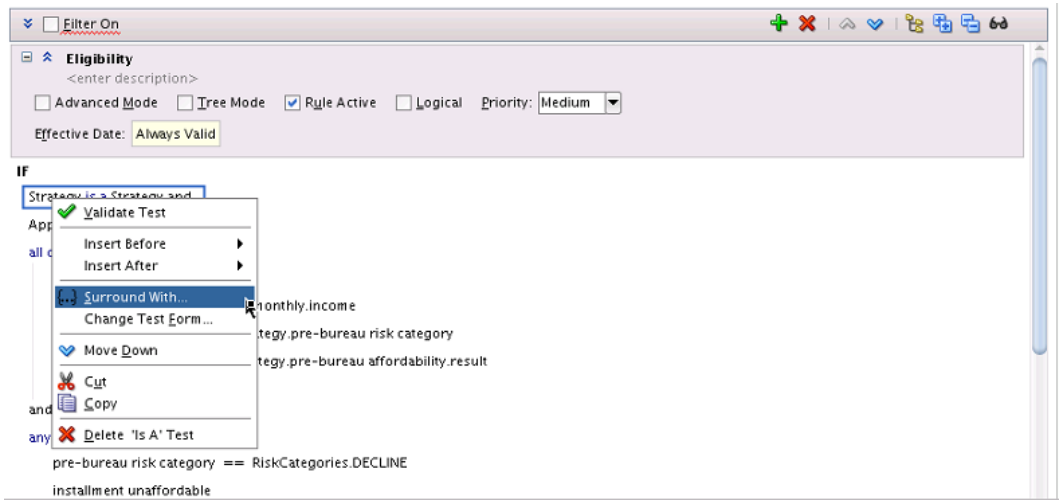
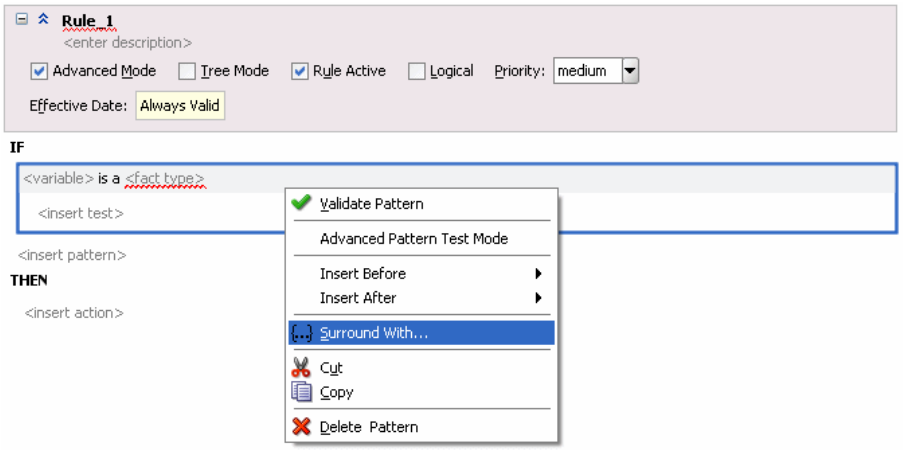


Figure 4-46 Surrounding With Option

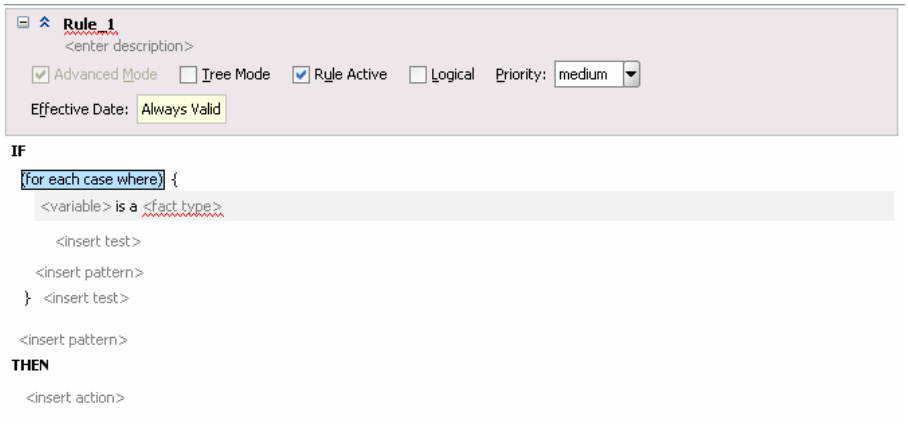


The Surround With dialog appears.

5. Choose the **Pattern Block** option from the Surround With dialog and click **OK**.

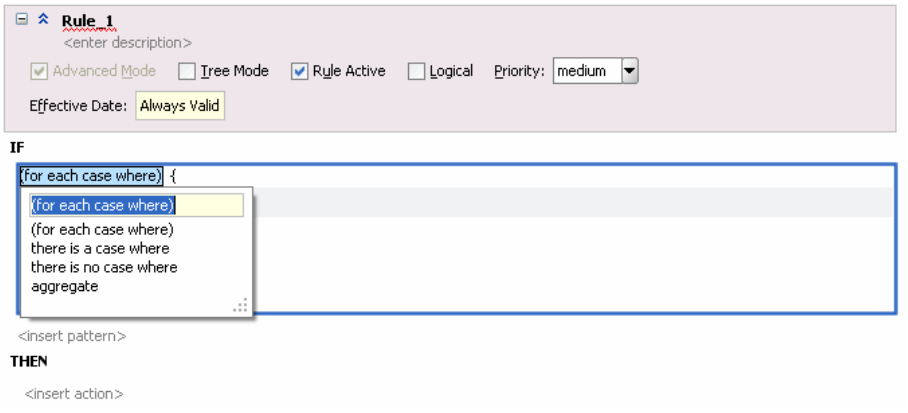
The pattern is surrounded by a nested pattern with the default **(for each case where)** as shown in Figure 4-47.

Figure 4-47 Default Pattern Matching Option: for each case where



6. Select the default **(for each case where)** option and select the desired pattern matching option from the list as shown in Figure 4-48.

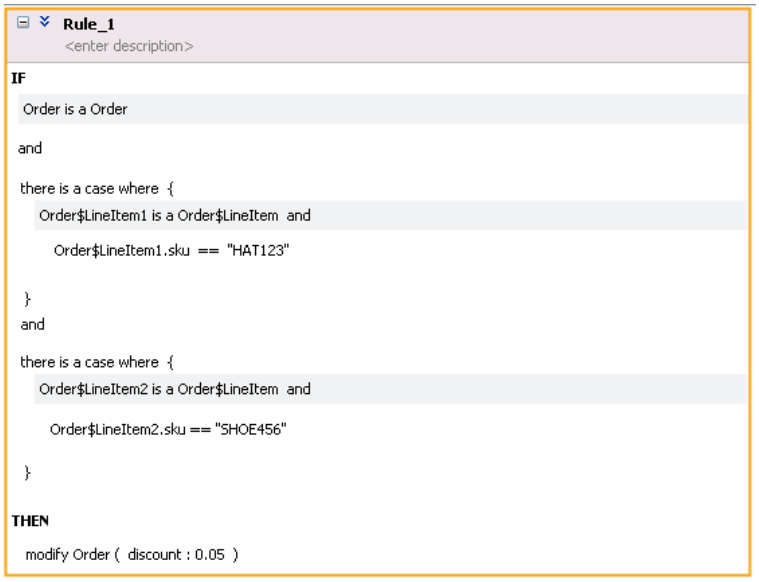
Figure 4-48 Adding an Advanced Pattern Match Option



How to Use Advanced Mode Matched Fact Naming

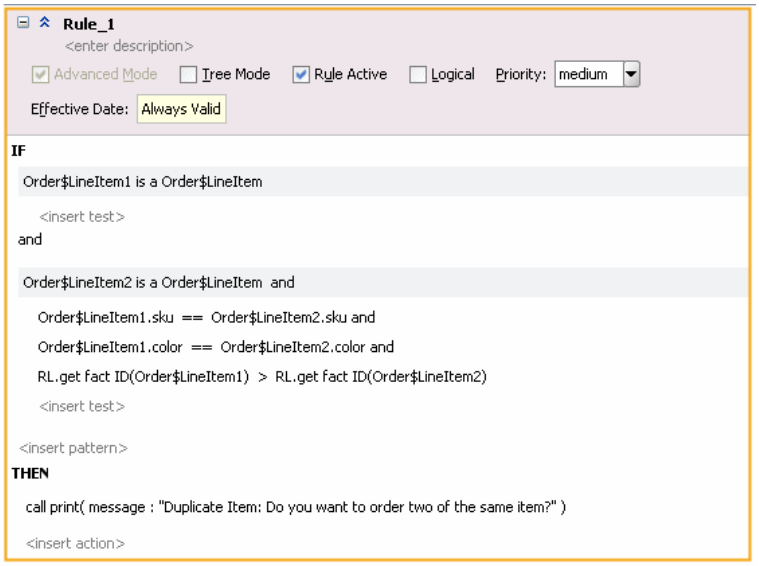
The matched fact name field, pattern binding variable, in a rule or a Decision Table lets you test multiple instances of the same type in a single rule. The matched fact name lets you enter a temporary name for the matched fact to use in a test. For example, the rules shown in Figure 4-49 show the use of pattern binding variables in a rule that applies a discount on a shoe item when an order includes at least one "matching" hat item.

Figure 4-49 Rule Using a Pattern Binding Variable



For example, you can create the rule, as shown in Figure 4-50 to find duplicate items in a customer order. This example shows the use of matched in a rule.

Figure 4-50 Rule to Find Duplicate Items in an Order

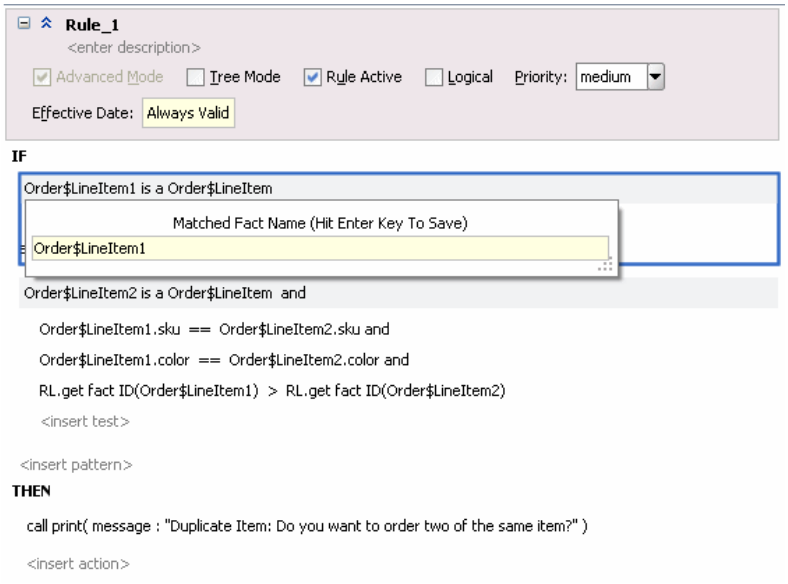


To use advanced mode matched fact naming:

1. Select the rule or Decision Table where you want to add a matched fact name.
2. Click the **Show Advanced Settings** button next to the rule name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
3. Select **Advanced Mode**.
4. Select the **<fact type>** and enter a fact type from the list.

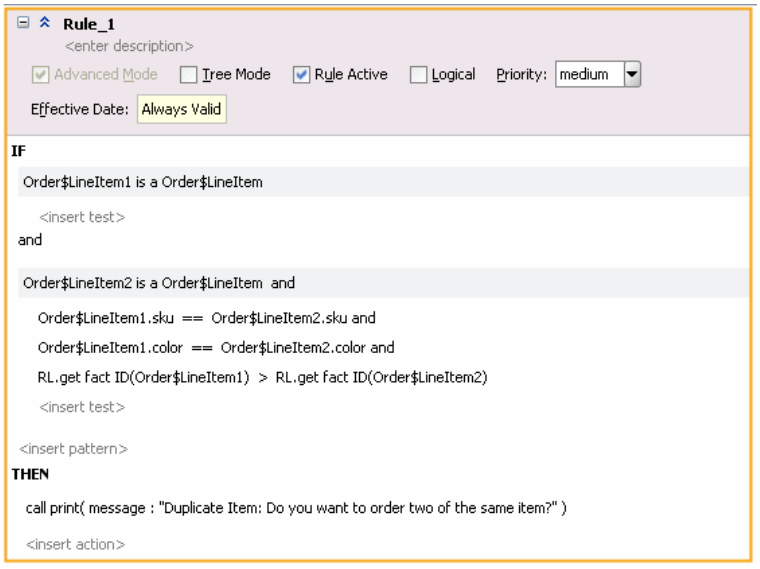
5. Select the supplied matched fact name and modify it as needed, as shown in Figure 4-51. For example, enter the matched fact name `Order$LineItem1` and then press **Enter**.

Figure 4-51 Adding a Matched Fact Variable Name



6. Create the rule as Figure 4-52 shows. Note that you can choose a matched fact name as an operand. Choose the **LineItem1** and **LineItem2** operands as needed to create the rule.

Figure 4-52 Choosing a Matched Fact Variable Name as an Operand



Note in Figure 4-52 that the test checking:

`RL.get fact ID(Order$LineItem1) > RL.get fact ID(Order$LineItem2)`

Prevents a single instance of an `Order$LineItem` from matching both patterns that match the `Order$LineItem` fact type. The ">" is required so that the rule does not fire for different permutations of different instances. For more information, see How Do I Correctly Express a Self-Join?.

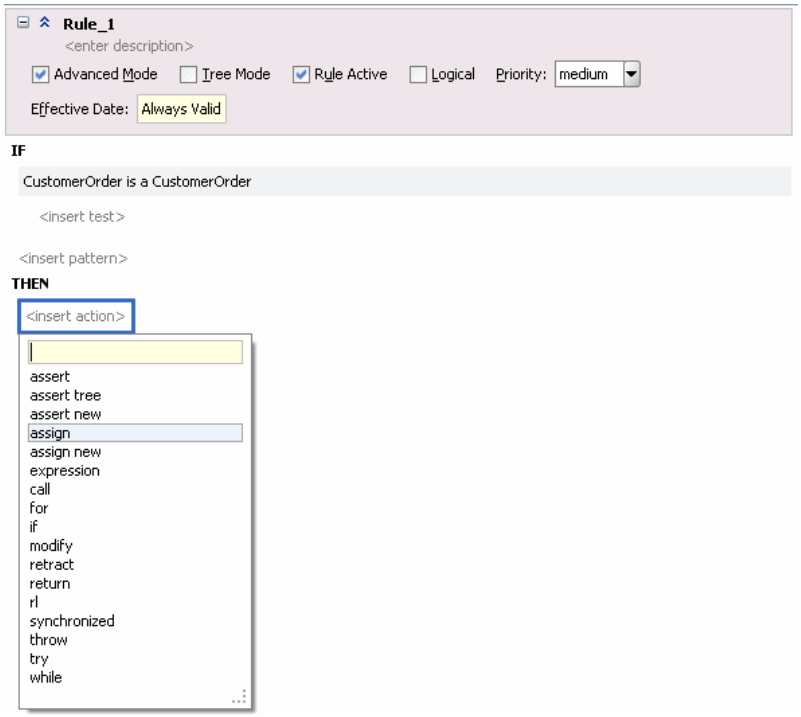
How to Use Advanced Mode Action Forms

When you create a rule with **Advanced Mode**, Rules Designer presents a list with the available actions shown in Table 4-6. For each form shown in Table 4-6, the options that Rules Designer presents are context sensitive. Thus, the lists and the number of items you see when you work with the action types are context sensitive, depending on which action you add and the choices you make while you enter the action.

To use advanced mode action forms:

1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. Select or add a rule or a Decision Table.
3. In the rule or Decision Table click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
4. Select **Advanced Mode**.
5. With the insertion areas showing, in a rule in the **THEN** area select **<insert action>**. This displays the action list, as shown in Figure 4-53.

Figure 4-53 Adding an Action to a Rule in Advanced Mode



6. In the list select the action you want to add.

   For example, select **assign new**.

7. In the **THEN** area, select the context sensitive parameters for the action and enter appropriate values.

Advanced Mode Action Options in Rule Designer

Table 4-6 Advanced Mode Action Options

Action Form	Description
`Assert`	Assert a fact
`Assert Tree`	Asserts a tree of facts given the root.
`Assert New`	Assert a new fact.
`Assign`	Assign a value to a variable.
`Assign New`	Assign a value to a new variable.
`Expression`	Perform expression.
`Call`	Call a function.
`For`	Oracle RL, like Java, has a for loop. A for loop includes a type with a variable and a collection. The type and variable defines the loop variable that holds the collection value used within the loop. Collection is an expression that evaluates to a collection of the correct type for the loop variable. You can use a for loop to iterate through any collection. A return, throw, or halt may exit the action block.
`If`	Using the if else action, if the test is true, execute the first action block, and if the test is false, execute the optional else part, which may be another if action or an action block. Oracle RL, unlike Java, requires action blocks and does not allow a single semicolon terminated action.
`Modify`	Modify a data value associated with a matched fact.
`Retract`	Retract a fact.
`Return`	The return action returns from the action block of a function or a rule. A return action in a rule pops the ruleset stack, so that execution continues with the activations on the agenda that are from the ruleset that is currently at the top of the ruleset stack.
`rl`	Use an Oracle RL expression that you supply.
`synchronized`	As in Java, the synchronized action is useful for synchronizing the actions of multiple threads. The synchronized action block lets you acquire the specified object's lock, then execute the action-block, then release the lock.
`throw`	Throw an exception, which must be a Java object that implements java.lang.Throwable. A thrown exception may be caught by a catch in a try action block.
`try`	The try, catch, and finally in Oracle RL is like Java both in syntax and in semantics. There must be at least one catch or finally clause.
`while`	While the test is true, execute the action block. A return, throw, or halt may exit the action block.

How to Use Advanced Mode Aggregate Conditions

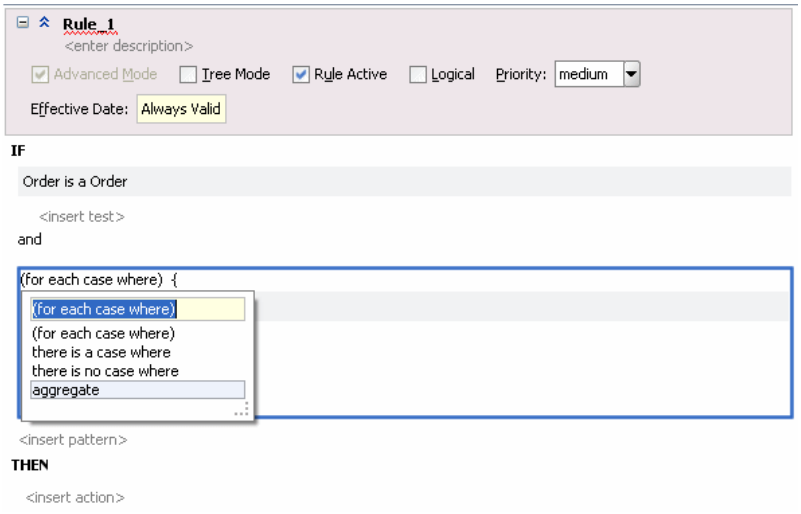
When you create a rule with **Advanced Mode**, Rules Designer supports the pattern matching aggregate option. When you write rule conditions that are based not only on one fact, but on many facts, you can use an aggregate. You use aggregate functions when the conditions have a view spanning multiple facts.

To use advanced mode aggregates:

1. Select or create the rule or Decision Table where you want to use an aggregate function.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
3. Select **Advanced Mode** and enter the fact type you want to work with.
4. Select **<insert pattern>** to add a pattern and select the pattern.
5. Right-click the pattern and select **Surround With...**. This displays the Surround With dialog.
6. In the Surround With dialog select **Pattern Block**. Click **OK**.

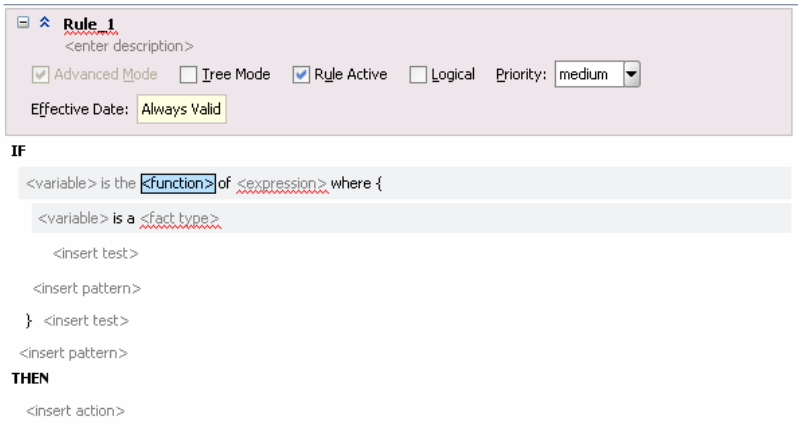
   For more information, see How to Use Advanced Mode Pattern Matching Options.
7. In the pattern select the first field. By default this field contains **(for each case where)**, as shown in Figure 4-54.

Figure 4-54 Adding an Advanced Pattern Match Option



8. Select the **aggregate** option. This adds the context sensitive fields for an aggregate, as shown in Figure 4-55.

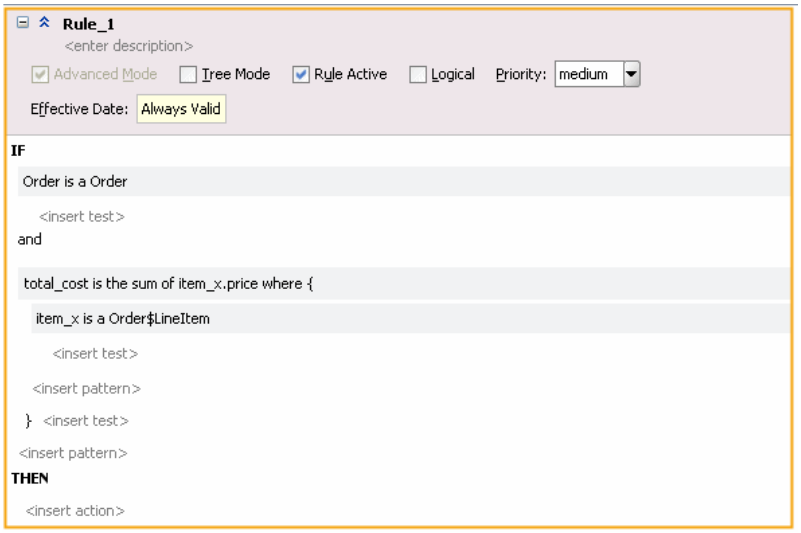
Figure 4-55 Using Aggregate Functions in a Rule



- Click **<function>** and select a function from the list.
- In the condition, click **<fact type>** and select a fact type from the list.
- Click **<expression>** and select an expression from the list.

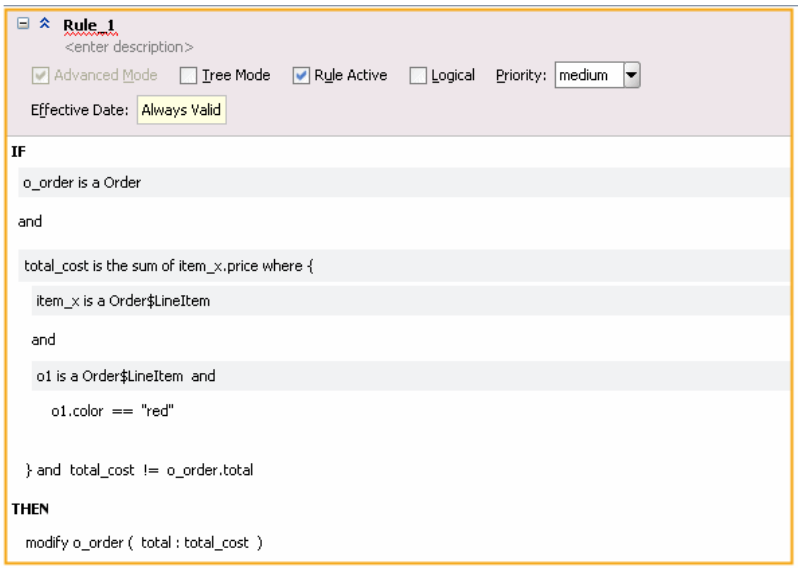
Rules Designer by default constructs variable names as you create the aggregate pattern. If needed for the rule you are constructing enter variable names to replace the default variable names. Figure 4-56 shows a completed rule using aggregate. In this example, for clarity the rule shows the variable names `total_cost` and `item_x`.

Figure 4-56 Completed Aggregate Function in a Rule



9. Enter additional tests as required. For this example you enter the test for items with color "red", as Figure 4-57 shows.

Figure 4-57 Using Aggregate Functions with Rules Red Color Total Cost Rule



Using Aggregate Functions

Table 4-7 shows the available aggregate functions.

Table 4-7 Aggregate Functions for Advanced Mode Rules

Function	Description
`count`	Count of matching facts.
`average`	Average of matching facts.
`maximum`	Maximum value of matching facts.
`minimum`	Minimum value of matching facts.
`sum`	Sum of matching facts.
`collection`	Builds a list of matching facts.

For example, to write a rule that specifies a special order as follows:

```
IF
   an order has more than 5 line items whose price is above a certain value
THEN
   the order requires manual approval
```

The five line items may span multiple facts. Thus, you can use the `count` aggregate function to write this sample special order rule.

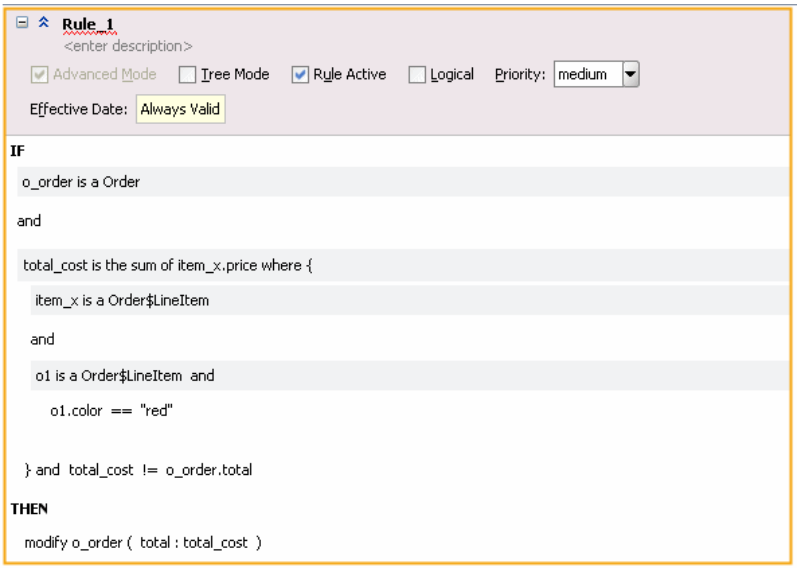
When you use an aggregate function, do the following:

- Select `aggregate` for the pattern.
- Enter a function from the list shown in Table 4-7
- Enter or select values from the context sensitive menus:
  - `<variable>` A name for the aggregate value.

– `<expression>` The value to aggregate, for example `driver.age`. When the aggregate function you select is the `count` function the `<expression>` is not used.

For example, you can compute the sum of the cost all the line items with color "red", as shown in Figure 4-58.

Figure 4-58 Using Aggregate Functions with Rules Red Color Total Cost Rule



What You Need to Know About Advanced Mode Rules

There are some special cases to keep in mind when you work with **Advanced Mode** rules, including the following:

- When you work with aggregates, in actions, you do not see pattern variables. The pattern variables are only shown for action lists when you use (foreach...) patterns. Thus, you cannot see pattern variables in aggregate, "there is a case", or "there is no case patterns".
- After you select **Advanced Mode** the **Advanced Mode** stays selected and inactive (gray), as long as your rule uses advanced options such as advanced pattern matching. To clear **Advanced Mode** you must remove or undo the advanced mode features (sometimes it is easier to start over by creating a non-advanced mode rule and then delete the advanced mode rule).

How to Clear Advanced Mode Option

1. Select the rule or Decision Table where you want to clear **Advanced Mode**.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name (see How to Show and Hide Advanced Settings in a Rule or Decision Table).
3. Consider the state of the rule:
   - If you can simplify the rule to enable the **Advanced Mode** option (such that the **Advanced Mode** button changes from gray to enabled). Then simplify the rule and when **Advanced Mode** is enabled, clear **Advanced Mode**.

- If you can use **Undo** to undo the steps you used to create the **Advanced Mode** rule, to get to a state where the rule is no longer in **Advanced Mode**, then use this technique to simplify the rule.
- If you cannot simplify the rule, then delete the rule and re-create it.

Working with Extended Tests

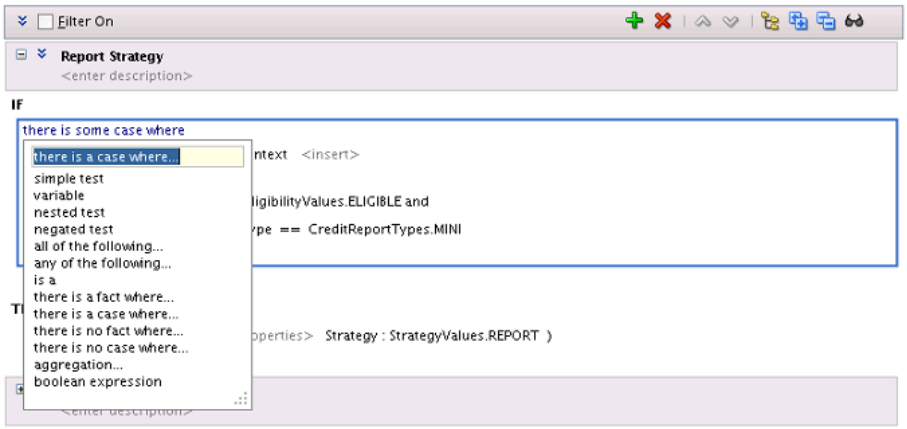
Extended tests should be used when building complex rules. Extended tests, or Simple Mode, replaces Advanced Mode rules.

Note:

Advanced Mode capability has been maintained for backward compatibility only. For more information about Advanced Mode, see Working with Advanced Mode Rules.

Everything that can be done in Advanced Mode can now be done in Simple Mode. The UI has been streamlined and improved to enable you to more easily create complex rules and tests, as shown Figure 4-59.

Figure 4-59 List of Extended Tests



Advanced mode rules can be converted to the equivalent simple mode rules by clearing the Advanced Mode check box.

Extended tests are only applicable to general rules, decision tables, and while defining business phrases. They are not visible in verbal rules.

Extended Test Forms

In addition to the original four tests (shown first in Table 4-8) there are new forms:

Table 4-8 Extended Tests

Forms	Description
simple test	This is the building block for conditions. Compares a value against another value, range or set. For example: Emp.salary > 1000

Table 4-8 (Cont.) Extended Tests

Forms	Description
variable	Initializes variables. For example: age = Duration.years between(Emp.birthdate,RL.date.get current())
nested test(...)	Encapsulates tests in a containing block. For example: (age > 50 or Emp.salary > 50000)
negated test(...)	Negates a test. For example: not(age > 50 and Emp.salary > 50000)
all of the following	all of the following are true. For example: (age > 50 and Emp.salary > 50000)

any of the following	some of the following are true.For example:
	`IF`
	`e is a Emp and there is no Emp where Emp.salary < e.salary <insert test> <insert test>THEN assign e.isLowestPaid = true`

is a	Defines a fact. For example: e **is a** Emp
boolean expression	Captures a boolean expression. For example: isEligible(Emp)

there is a case where	This test has 1 or more child tests that are ANDed.
	The child tests are all true for at least 1 case. A case is a binding of facts to contained **is a** tests.
	Must have **is a** descendant.
	Example:
	`There is a case where`
	`e is a Emp and`
	`d is a Dept and`
	`e.salary > 1000000 and`
	`d.name == "Marketing" and`
	`d.employees contains e`

Table 4-8 (Cont.) Extended Tests

Forms	Description

there is a <factType1>,...<factTypeN> where#*
This test has N or more child tests that are ANDed
	Hidden <factType> **is a** <factType> tests as first N children.
	The child tests are all true for at least 1 case.
	It is legal to have no visible child tests, in which case the where keyword should be suppressed.
	Example:
	`IF`
	`  there is a Emp, Dept where`
	`  Emp.salary > 1000000 and`
	`  Dept.name == "Marketing" and`
	`  Dept.employees contains Emp`
	`THEN`
	`  call print "there is a highly paid marketer!"`
	`IF`
	`there is a Emp`
	`THEN  call print "somebody works here!"`

there is no case where	This test has 1 or more child tests that are ANDed. The child tests are true for no case (no binding of facts to contained is a tests satisfy all the other tests). Must have **is a** descendant.
there is no <factType1>,...,<factTypeN> where	Hidden <factType> **is a** <factType> as first N children The child tests are true for no case

Table 4-8 (Cont.) Extended Tests

Forms	Description

aggregation	This test has 0 or more child tests that are ANDed.
	Must have **is a** child (may be hidden).
	v is the sum | average | minimum | maximum | count | collection of<expression> where
	Where clause omitted when there are no visible child tests.
	`IF`
	`  number of employees is the count of Emp`
	`THEN`
	`  call print "number of employees: " + number of employees`
	
	`IF`
	`  number of male employees is the count of Emp where`
	`  Emp.gender == "M"`
	`THEN`
	`  call print "number of male employees: " + number of male employees`
	
	Note that in both rules above, the SDK will create a hidden nested **is a** test for Emp.
	You can also use an explicit **is a**
	`IF`
	`  number of male employees is the count of e where`
	`  e is Emp and`
	`  e.gender == "M"`
	`THEN`
	`  call print "number of male employees: " + number of male employees`

Figure 4-60 is an example where "there is a case where" form is used:

Figure 4-60 Extended Test Example 1

Figure 4-61 is an example where "there is no case where" form is used:

Figure 4-61 Extended Test Example 2

For information about how to build complex rules, see Working with Rules.

Working with Tree Mode Rules

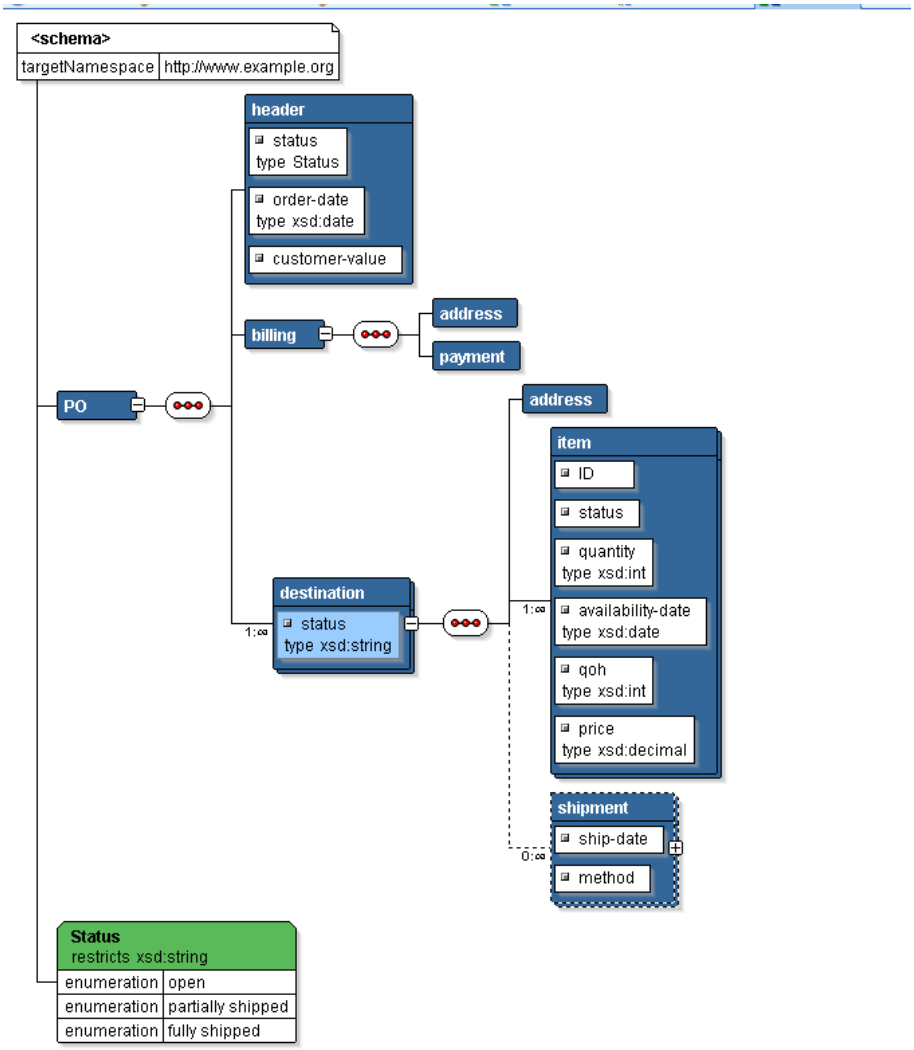
Tree Mode rules make it easier to work with a master detail hierarchy, where there are nested elements that map to a parent child relationship.

Consider the lifecycle of an application fragment that uses business processes and rules to process a retail purchase order (PO). The purchase order has a header with business terms that apply to the entire PO. The PO also contains a list of shipping destinations. Each destination has an address, a list of items to be shipped to the destination's address, and a list of shipments.

Consider the business rule: the status of a PO is "fully shipped" if the status of every item is either "shipped" or "canceled".

Figure 4-62 shows a sample XML schema representation for the PO example. The XML documents for the PO are tree structured. This allows a natural representation for the PO. For example, the PO itself is the top level document element and destinations are nested elements that contain item elements and shipment elements. Shipment elements also contain item elements that reference the ordered items. Status has a list of valid values.

Figure 4-62 PO Schema for Tree Mode Rules Sample



The following example of sample Purchase Order (PO) schema shows the sample purchase order XML schema as represented in Figure 4-62.

```
<?xml version= '1.0' encoding= 'UTF-8' ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://www.example.org"
targetNamespace="http://www.example.org"
     elementFormDefault="qualified">
    <xsd:element name="PO">
        <xsd:annotation>
            <xsd:documentation>A sample element</xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="header">
                    <xsd:complexType>
                        <xsd:attribute name="status" type="Status"/>
                        <xsd:attribute name="order-date" type="xsd:date"/>
                        <xsd:attribute name="customer-value"/>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element name="billing">
                    <xsd:complexType>
```

```
                    <xsd:sequence>
                        <xsd:element name="address"/>
                        <xsd:element name="payment"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="destination" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="address"/>
                        <xsd:element name="item" maxOccurs="unbounded">
                            <xsd:complexType>
                                <xsd:attribute name="ID"/>
                                <xsd:attribute name="status"/>
                                <xsd:attribute name="quantity" type="xsd:int"/>
                                <xsd:attribute name="availability-date" type="xsd:date"/>
                                <xsd:attribute name="qoh" type="xsd:int"/>
                                <xsd:attribute name="price"
                                               type="xsd:decimal"/>
                            </xsd:complexType>
                        </xsd:element>
                        <xsd:element name="shipment" minOccurs="0" maxOccurs="unbounded">
                            <xsd:complexType>
                                <xsd:sequence>
                                    <xsd:element name="item" maxOccurs="unbounded">
                                        <xsd:complexType>
                                            <xsd:attribute name="ID"/>
                                            <xsd:attribute name="quantity"/>
                                        </xsd:complexType>
                                    </xsd:element>
                                </xsd:sequence>
                                <xsd:attribute name="ship-date"/>
                                <xsd:attribute name="method"/>
                            </xsd:complexType>
                        </xsd:element>
                    </xsd:sequence>
                    <xsd:attribute name="status" type="xsd:string"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:simpleType name="Status">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="open"/>
        <xsd:enumeration value="partially shipped"/>
        <xsd:enumeration value="fully shipped"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Sample Abbreviated PO XML Instance

Example 4-1 shows part of the XML for an instance of the PO schema. To use tree mode rules you can create a rule that tests one or more business terms and if the tests pass, one or more business terms are added or changed. Oracle Business Rules has special support to enable error-free authoring of rules on fact trees like the sample PO instance.

For example, consider creating a rule for an instance of the PO schema that states:

```
IF the time between the order date and the date for availability of an item is more
than 30 days
THEN cancel the item
```

Example 4-1 Sample Abbreviated PO XML Instance

```
<PO xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.org ../../../../Temp/PO.xsd"
    xmlns="http://www.example.org">
  <header/>
  <billing>
    <address/>
    <payment/>
  </billing>
  <destination>
    <address/>
    <item ID="a01"/>
    <item ID="a02"/>
    <item ID="a03"/>
    <shipment>
      <item ID="a01"/>
      <item ID="a02"/>
    </shipment>
  </destination>
</PO>
```

Understanding Tree Mode Rules (Non-Advanced Mode)

You use non-advanced tree mode, or simple tree mode, when the **Advanced Mode** option is not selected and **Tree Mode** is selected. With this mode Rules Designer shows **ROOT**: **<fact type>** where you enter the root fact type.

When you create rules with **Tree Mode** selected and **Advanced Mode** cleared, you can reference properties in the tree using qualified names, for example:

- `PO/destination/item.quantity` that is similar to `item.quantity` but only items that are a `destination` of PO are matched.
- `PO$Destination$item.quantity` that refers to a `List<item>`. This reference is unchanged from non-tree mode.

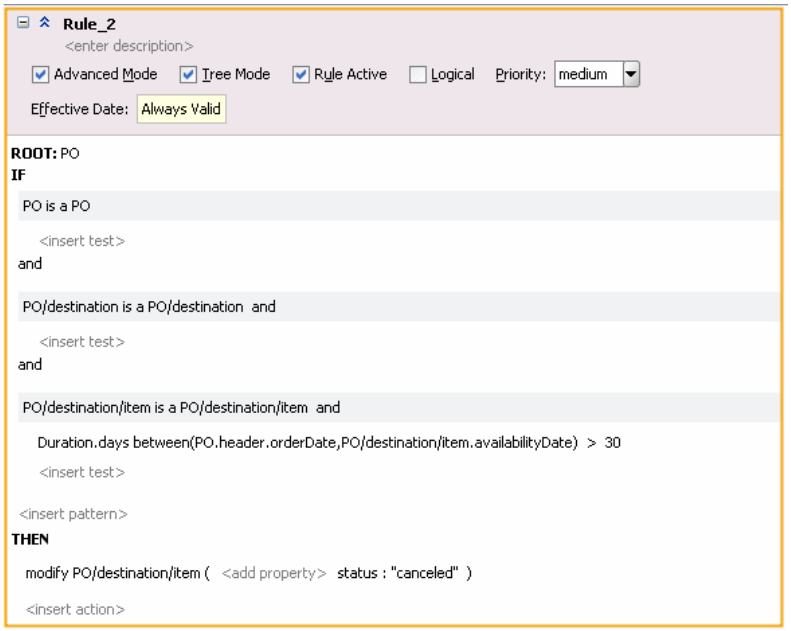
With Simple Tree Mode you can only choose terms that do not require many-to-many joins or aggregation.

For more information, see How to Create Simple Tree Mode Rules.

Understanding Advanced Tree Mode Rules

You use advanced tree mode when the **Advanced Mode** option is selected and the **Tree Mode** option is selected. With this mode Rules Designer shows **ROOT**: **<fact type>** where you enter the root fact type, as shown in Figure 4-63. Rules Designer shows patterns for the tree structured facts but the simple tests that join the parent and child facts are hidden.

Figure 4-63 Advanced Tree Mode



In advanced tree mode the tree mode patterns, except for the root, display as:

`<operator> <variable> is a <fact path>`

Where the `<fact path>` is an XPath-like path through the 1-to-1 and 1-to-many relationships starting at the root. For example, each fact path has a name like `PO/destination`, where `PO` is the root fact type and the destination is a property of type `List`. A 1-to-many relationship in a fact path is indicated with a "/", as in `PO/destination`.

A 1-to-1 relationship in a fact path is indicated with "`.`" This unchanged from non-tree mode. For example, `item.availabilityDate`.

Advanced mode exposes the concept of a pattern, the simplest of which is **is a** pattern. For example, `p is a PO` causes `p` to match, iterate over, all the `PO` facts, and `d is a p/destination` causes `d` to match all the destinations of `p`. The left side of **is a** is a variable, and the right side is a fact type or a fact path. By default, Oracle Business Rules sets the variable name equal to the fact type or path. For example, PO is a PO. A pattern can also be a pattern block. A pattern block has a logical quantifier, negation, or aggregation that applies to the patterns and tests nested inside the block.

For more information, see How to Create Advanced Tree Mode Rules.

When you work with advanced tree mode rules, Rules Designer expects you to use an aggregation pattern, including exists and not exists to combine terms from different child forests into the same rule while avoiding a Cartesian product.

How to Create Simple Tree Mode Rules

The following procedure creates the PO rule to cancel non 30-day availability items.

```
IF the time between the order date and the date for availability of an item is more
than 30 days
THEN cancel the item
```

To create simple tree mode rules:

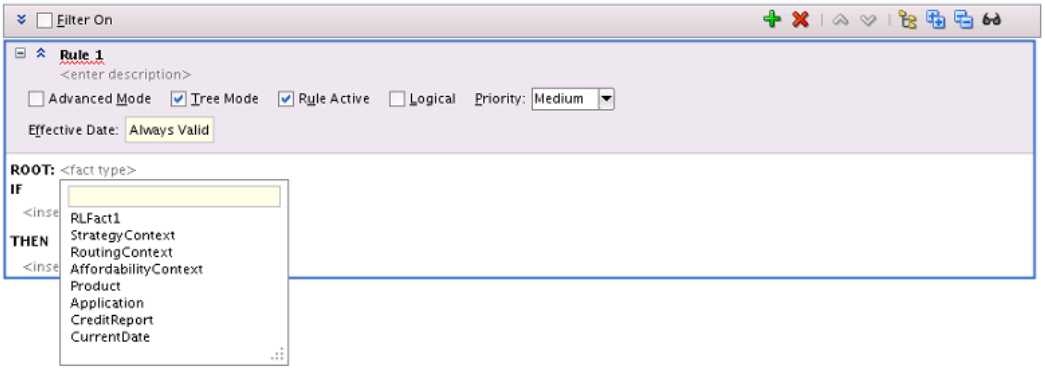
1. Create an IF/THEN rule in your ruleset and view the advanced settings.

   For more information on adding general rules, see How to Add General Rules.

   For more information on advanced settings, see How to Show and Hide Advanced Settings in a Rule or Decision Table.

2. Select **Tree Mode**. Next to ROOT:, click the <fact type> placeholder and select from the list.

Figure 4-64 Simple Tree Mode: Configuring the Root



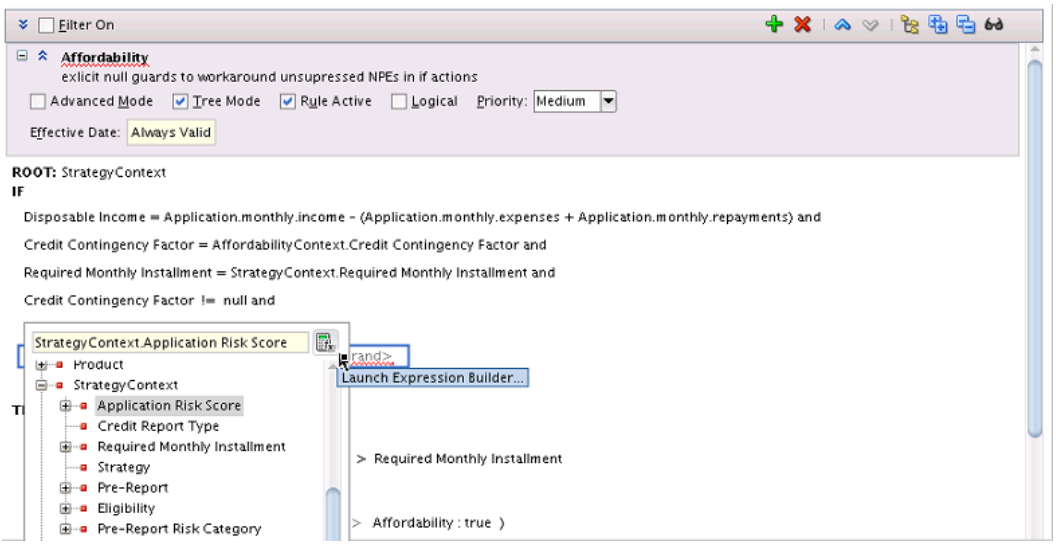
   - Select **<insert test>** and select from the list.

     The IF statement now reads `IF <operand> == <operand>`.

   - Select the left-hand **<operand>** and select an option from the list.

3. Select the **Expression Builder** button, as shown in Figure 4-65.

Figure 4-65 Adding a Simple Tree Mode Expression



   - In the Expression Builder dialog, copy and delete the item shown in the **Expression** area.

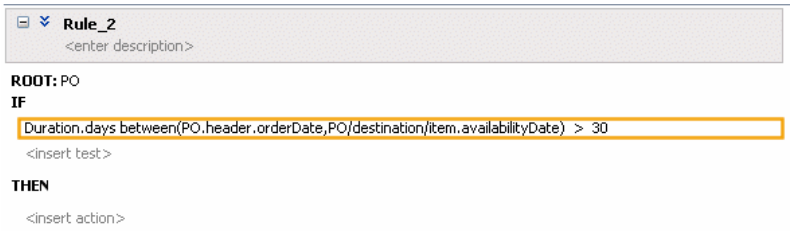
   - In the Expression Builder, select the **Functions** tab.

- In the navigator, expand **Duration** and double-click the **daysbetween** function.
- Remove the **daysbetween** argument templates.
- In the **daysbetween** function, paste the value you previously cut as the second argument.
- In the Expression Builder dialog, select the **Variables** tab.
- For the **daysbetween** function first argument, use the navigator to expand **PO** and expand **header**, and double-click **orderDate**.
- In the Expression Builder dialog, click **OK**.

  For more information, see Introduction to Expression Builder.

4. In the list, in the expression area and press **Enter**. Select the operator and enter >.
5. Select the right-hand **<operand>** and enter the value 30 and press **Enter**, as shown in Figure 4-66.

Figure 4-66 Simple Tree Mode: Right-Hand Operand with Value 30



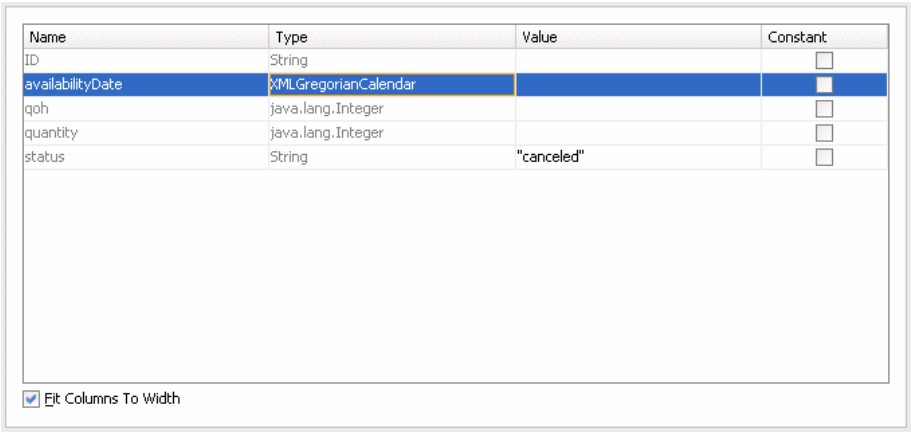
- Click **<insert action>** and from the list select **modify**.

  The THEN statement now reads: `THEN modify <target>`.
- Click **<target>** and from the list select PO/destination/item. The THEN statement now reads:

  ```
  THEN modify PO/destination/item ( <add property> )
  ```
- Click **<add property>**. This displays the properties dialog.

  In the properties dialog for the status name, enter the value "canceled", as Figure 4-67 shows.

Figure 4-67 Simple Tree Mode: Action



6. In the Properties dialog, click **Close**.

   This displays the finished rule, as shown in Figure 4-68.

Figure 4-68 Simple Tree Mode Rule Final Rule

Note that in the `modify` statement, `PO/destination/item` refers to the particular `item` instance member.

How to Create Advanced Tree Mode Rules

The following procedure creates a free shipping rule that can be summarized as:

```
IF the total cost of "free shipping eligible" items to a given destination is
greater than $40
THEN shipping of those items is free
```

To create advanced tree mode rules:

1. Create an IF/THEN rule in your ruleset.

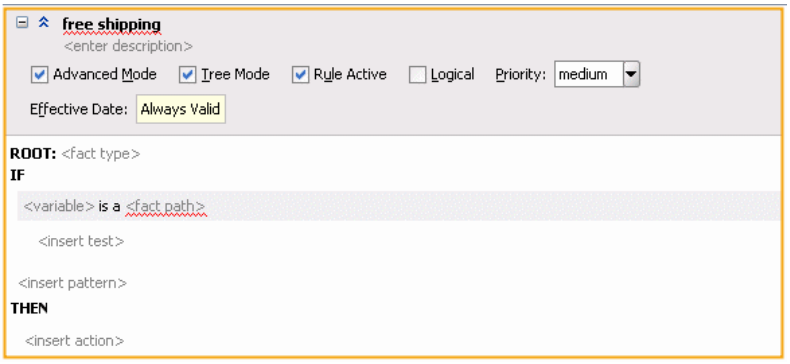
   For more information, see How to Add General Rules.

2. View advanced settings.

   For more information, see How to Show and Hide Advanced Settings in a Rule or Decision Table.

3. Select **Advanced Mode** and select **Tree Mode** as Figure 4-69 shows.

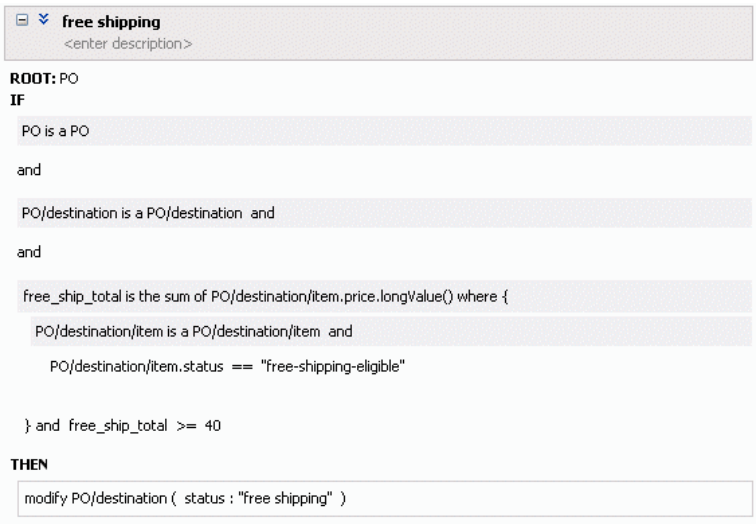
Figure 4-69 Advanced Tree Mode Rule for Free Shipping



4. Select the **<fact type>** place holder and from the list, select **PO**.

5. Complete the free shipping rule, as shown in Figure 4-70.

Figure 4-70 Advanced Tree Mode Free Shipping Rule



What You Need to Know About Tree Mode Rules

When you select **Tree Mode** and select a root fact type, the options lists show all available fact types (not just the children of the root fact type). This allows you to view all available fact types as well as the children of the root fact type. There is no option to limit the option list to only show the children of the selected root fact type.

Using Date Facts, Date Functions, and Specifying Effective Dates

Oracle Business Rules provides functions that make it easier for you to work with times and dates, and provides effective date features to let you determine when rules are effective, based on times and dates.

- The **CurrentDate** fact allows you to reason on a fact representing the current date.
- The **Effective Date** value lets you specify a start date and end date that defines a date or date and time range when all the rules and Decision Tables in a ruleset, an individual rule, or an individual Decision Table are effective.

Table 4-9 describes the available **Effective Date** options.

Table 4-9 Effective Date Possible Values

Effective Date	Description
Always Valid	Specifies to set neither "From" nor "To" dates.
From (without To date set)	Valid from a certain date indefinitely into the future.
To (without a From date set)	Valid from now until a certain date.
From Set and To set	Valid only between two dates.

An effective date specification other than **Always** can be one of the following:

- Date only, with no time specification: In this case, an effective date assumes a time of midnight of that date in each time zone.

- Date, time zone, with no time specification: In this case, an effective date assumes a time of midnight as of the specified date in the specified time zone.
- Date, time zone, time specification: In this case, the date and time is fully specified.
- Time specification only, with no date and no time zone: applies for all days at the specified time.
- Time and time zone specified, with no date: applies for all days at the specified time.

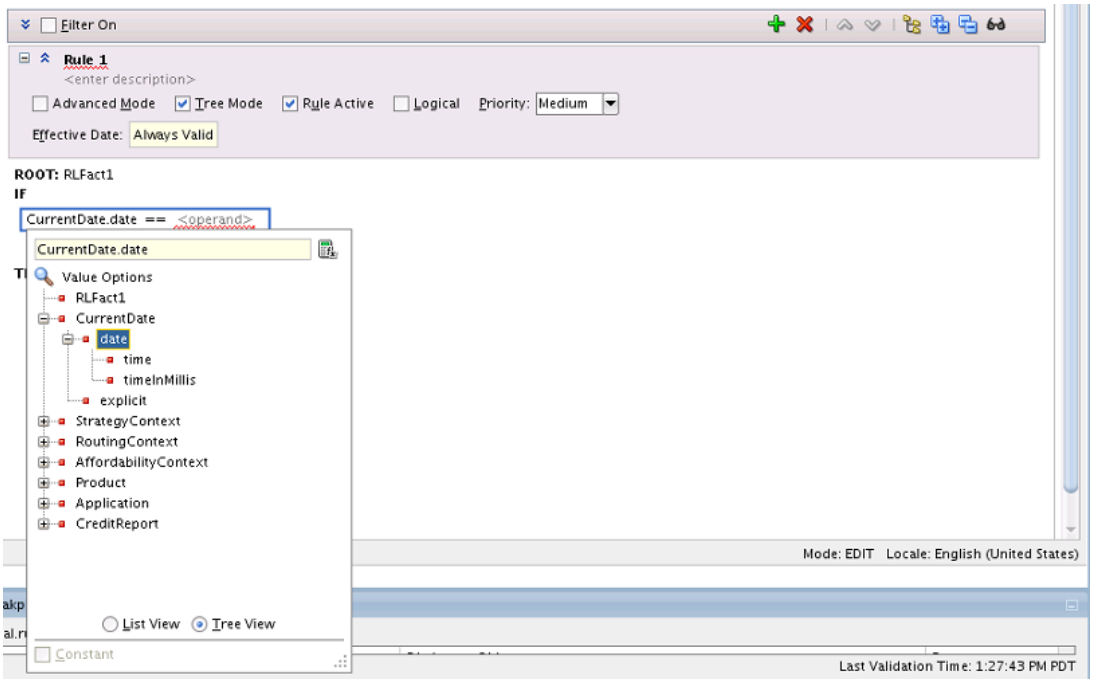
How to Use the Current Date Fact

You can use the current date fact in a rule or a Decision Table.

To use the CurrentDate fact:

1. Select a ruleset from the **Rulesets** navigation tab.
2. Select a rule within the ruleset.
3. In the **IF** area, add a condition that uses the CurrentDate fact and the date method of `Calendar` type, as shown in Figure 4-71.

Figure 4-71 Rule with Condition Using CurrentDate Fact



What You Need to Know About Effective Dates

By default, the Oracle Business Rules Engine implicitly manages the clock associated with the CurrentDate fact and the effective date, setting each to the value of the system date. Using the RL Language functions `setCurrentDate()` and `setEffectiveDate()` you can explicitly set the current date and the effective date. For more information, see Built-in Functions in the *Rules Language Reference for Oracle Business Process Management*guide.

An effective start date is defined as the first point in time at which a rule, Decision Table, or ruleset may actively participate in rule evaluations and fire. Thus, if a rule is effective it may fire if its condition is satisfied and if the rule is not effective, it does not fire whether the condition is satisfied or not.

An effective end date is the first moment in time at which the rule, Decision Table, or ruleset no longer actively participates in rule evaluations (not effective means the rule does not fire).

The effective start and end date can be set on a Decision Table, but these dates cannot be set individually for the rules within a Decision Table.

Rules and Decision Tables also include the **Rule Active** option. This option is set independent of the effective dates and makes dates effective without changing or removing the specified effective date. For more information on using the **Rule Active** option, see How to Select the Active Option.

The precedence of the effective date, when it is defined for both a ruleset and for the rules or Decision Tables within a ruleset, is as follows (with the top precedence being 1):

1. If the ruleset **Rule Active** option is cleared, then RL Language is not generated for that entity.
2. If one or both of the effective date properties are selected for a ruleset, then those effective start dates and effective end dates define the range of effective dates allowable for rules or Decision Tables that are defined within the ruleset (that is, if in the ruleset the **From** check box, the **To** check box, or both check boxes are selected in the Set Effective Date dialog).

   Thus, the effective dates specified for rules or Decision Tables within a ruleset must not violate the boundaries established by the ruleset that contains the rules or Decision Tables. For example, a rule may not have an effective end date that is later than the effective end date specified for a ruleset.
3. If any individual rule or Decision Table has **Rule Active** cleared, then RL Language is not generated for that rule or Decision Table.
4. If the Set Effective Date dialog for a ruleset includes **Time** selected or this option is selected on a rule or a Decision Table in the ruleset, then all instances of rules or Decision Tables in the ruleset must have **Time** selected when effective dates are specified. In this case, if **Both** or **Date** is selected then Rules Designer shows a validation warning:

   ```
   RUL-05742: Calendar form incompatibility detected with forms Time and DateTime.
   If the calendar form is set to Time on a rule set or any of the rules or
   decision tables within that ruleset then the calendar form for that entire
   rule set is restricted to Time.
   ```

How to Use Duration, JavaDate, OracleDate, and XMLDate Methods

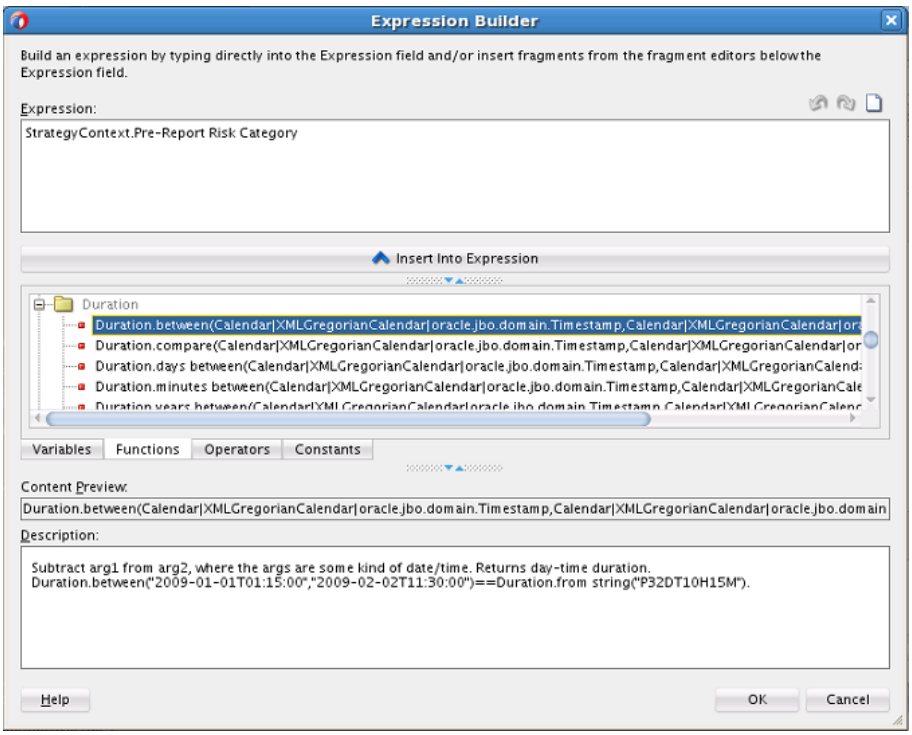
You can use the Duration, JavaDate, and XMLDate, OracleDate, and OracleDuration extension methods in a rule or a Decision Table. For more information, see Oracle Business Rules Built-in Classes and Functions.

To use a Duration method:

1. Select ruleset from the **Rulesets** navigation tab.

2. Select a rule within the ruleset (you can also use Duration methods in a Decision Table).
3. In the **IF** area, add a condition.
4. Select an operand in the rule condition.
5. From the list, select **Expression Builder...**. For more information, see Introduction to Expression Builder.
6. In the Expression Builder, select the **Functions** tab.
7. In the Expression Builder, in the Navigator, expand the **Duration** folder.
8. Double-click to select and insert the appropriate method as needed for your duration test.
9. Provide the appropriate arguments for the method. For example, see Figure 4-72.
10. Click OK to review your rule.

Figure 4-72 Using Duration Methods in a Rule



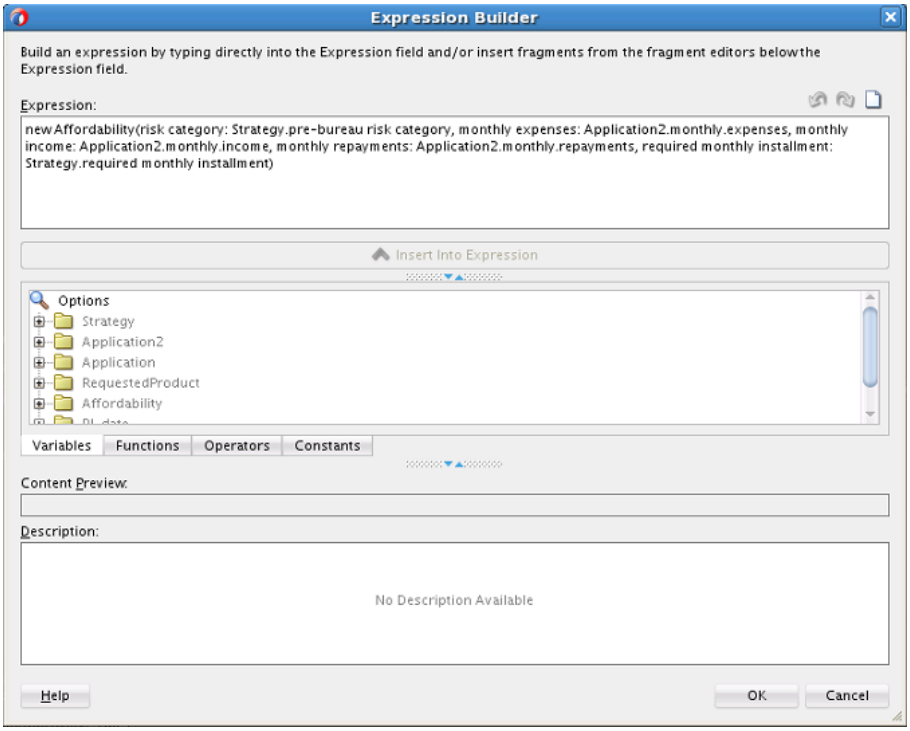
Introduction to Expression Builder

You can access the expression builder from different parts of Rules Designer, including in the Edit Globals dialog, and in the conditions area when you work with conditions in Decision Tables, and when you enter rules and Decision Tables in advanced mode with free form expressions select

Use the expression builder to create and edit expressions for Oracle Business Rules.

Figure 4-73 shows the Rules Designer expression builder.

Figure 4-73 Rules Designer Expression Builder



How to Use the Expression Builder

In the expression builder when you double-click items in the **Variables** or **Functions** navigation trees, or in the **Operators** tab, or in the **Constants** tab, this inserts the item into the expression in the **Expression** area. You can also create or edit expressions directly by entering text in the **Expression** area.

When you enter an expression, note that Variables are valid assignment targets and Constants are not valid assignment targets. Thus, you should use both tabs if you are unsure what type of item you want to add to the expression you are building.

Specify an argument for a selected function by placing the cursor inside the function in the **Expression** field and double-clicking the expression or function to insert. For example, place the cursor inside the parentheses of a function and select a variable. This inserts the variable in the expression at the cursor position.

What You Need to Know About Working with Expressions

XML fact types allow XML Schema types, elements, and attributes to be used when writing rules. Elements and types defined in XML Schema can be imported into the data model and can then be used to create rules and Decision Tables, just as with Java fact types and RL Fact types. The mapping between the XML Schema definition and the XML Fact types uses the Java Architecture for XML Binding (JAXB). By default, Oracle Business Rules uses the JAXB 2.0 shipped with the Oracle Application Server. JAXB as defined in JSR-222 provides a mapping between the types, names, and conventions in an XML Schema definition and the available types, allowed names and conventions in Java. For example, an element named `order-id` and of type `xsd:integer` is mapped to a Java Bean property named `orderID` of type `BigInteger` (and `xsd:int` type maps to Java `int`).

You can use expressions in Oracle Business Rules. Expressions allow arithmetic using the operators `*`, `+`, `/`, `%`, and other supported operators on primitive numerics, for example `double`, `int`, and the numeric types `Integer`, `Long`, `Short`, `Float`, `Double` `BigDecimal`, and `BigInteger` that are available in the built-in dictionary.

Expressions allow casting between any two numeric types, for example, `(short) ((BigInteger)1 + (Long)2)`. The following code example shows a few additional sample expressions in actions with types and casting.

```
assign new double db = 3.3
assign new BigDecimal bd = 3.3  // no cast required
assign db = bd // no cast required
assign bd = (BigDecimal)db // cast is required
```

The expression processor uses the XPath/Xquery rules for type promotion (XML Path Language (XPath) 2.0). For example, `BigDecimal` is promoted to `float/double`; type promotion going the other direction requires a cast, except for literals such as 3.3.

Using Value Sets as Constraints for Options Values in Rules

You can use List of Values value set and List of Ranges value sets to specify constraints for business terms in rules. This enables you to use Rules Designer to produce validation warnings for possible errors where a value supplied is out of range, or not within a set of possible values as specified in a value set.

Oracle Business Rules also lets you use value sets to specify constraints for global initial values, function return values, or function argument values. For more information, see Working with Oracle Business Rules Globals and Associating a Value Set with Business Terms.

How to Use a List of Ranges Value Set as a Constraint for a Business Term

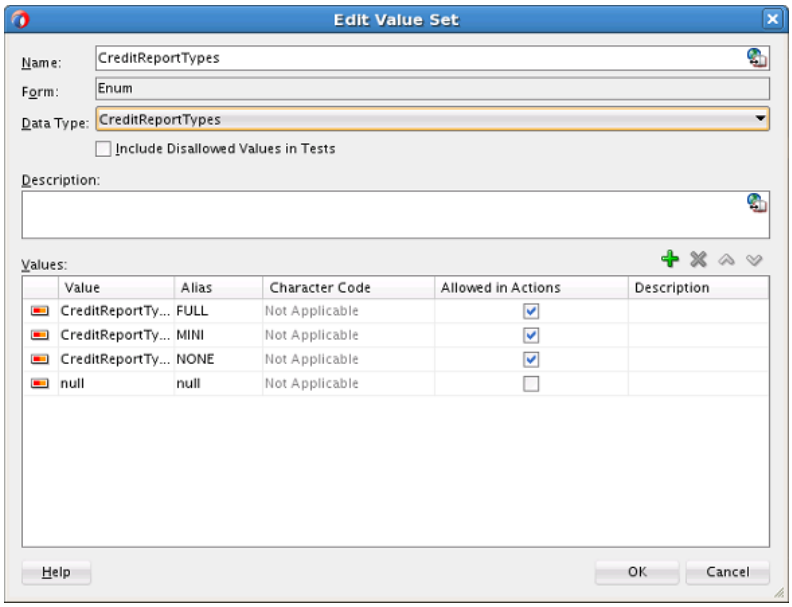
You can use a list of ranges value set as a constraint for any business term other than a function result.

For more information on using a list of values value set as a constraint, see How to Use a List of Values Value Set as a Constraint for a Fact Property.

To use a List of Ranges value set as a constraint for a fact property:

1. In the **Value Sets** tab, double-click a value set to open the **Edit Value Set** dialog.
2. Specify a value set that includes the ranges you want to include and select **Allowed in Actions** for all valid ranges. To include a range, clear **Allowed in Actions** for the top and bottom endpoints.
3. Select **Included Endpoint** as needed for the application.
4. Clear **Include Disallowed Values in Tests**. For example, for a value set that defines valid grades and that does not allow values greater than 100, or less than 0, define the value set endpoints.

Figure 4-74 Valid Value Sets for a Fact Property



5. Associate this value set with a business term. For this example, associate the value set with test_math1.

Now, if you define a rule with a test that uses the fact property you will receive a validation warning when a value is out of range. For example if you define a rule with an expression with the value -10, Rules Designer will show a validation warning.

How to Use a List of Values Value Set as a Constraint for a Fact Property

You can use a list of values value set as a constraint for a fact property.

For more information on using a list of ranges value set as a constraint, see How to Use a List of Ranges Value Set as a Constraint for a Business Term.

To use a List of Values value set as a constraint for a fact property:

1. Specify an LOV value set that includes the values you want to include, and select **Allowed in Actions** for all valid values. For more information, see How to Define a List of Values Global Value Set.
2. Clear **Allowed in Actions** for the otherwise value set.
3. Clear **Include Disallowed Values in Tests**.
4. Associate this value set with a fact property.

How to Use Value Sets to Provide Options for Test Expressions

You can use LOV value sets to provide options for expressions and actions.

To use value sets to provide options for rule expressions and actions:

1. In Rules Designer, define an LOV value set of a type corresponding to a fact property. For more information, see How to Define a List of Values Global Value Set.

2. Associate the value set with a fact property. For more information, see How to Associate a Value Set with a Fact Property.

3. When you enter expressions, Rules Designer shows the values in the values options. For example, when you associate a fact property `Driver.eye_test` with an LOV value set named `eyes`, with values: `pass`, `fail`, and `glasses_required`, and then you use `Driver.eye_test` in a test expression, the values are limited.

Importing Runtime Rules Changes From Repository Into JDeveloper

This section discusses how to import changes to a rule implemented in SOA Composer into the JDeveloper.

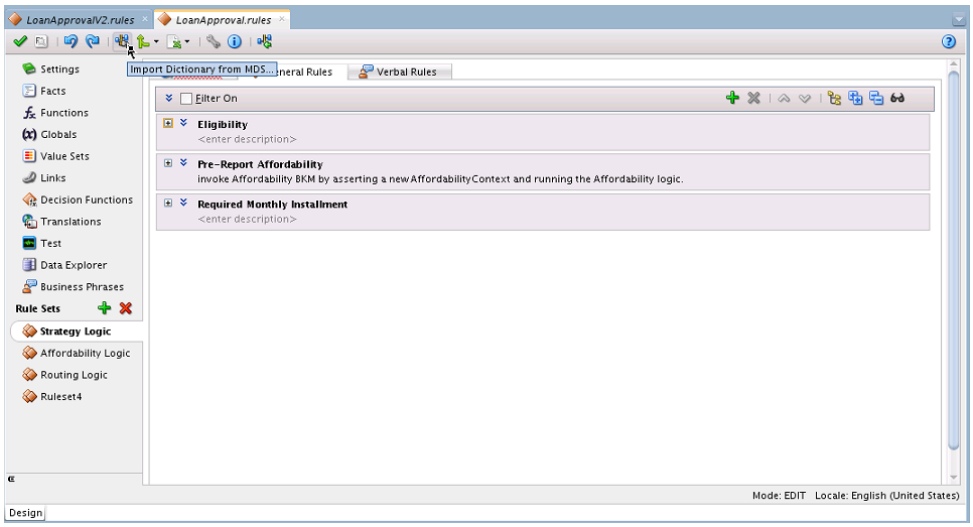
When you make changes to a dictionary in SOA Composer, you must upload them to MDS repository as described in Publishing Changes for an Oracle Business Rules Dictionary. However, these changes do not get updated in JDeveloper. You need to import the changes from MDS repository into JDeveloper manually.

To import the changes into the JDeveloper,

1. Select the rule in the application navigator for which changes were made.

2. Click the **Import From MDS** button in Rule Editor as shown in Figure 4-75.

Figure 4-75 Importing Changes from the MDS Repository



3. Select the MDS Repository from the Import Dictionary dialog.

4. Click **OK**.

   Changes are updated in JDeveloper and you can view the changes in the Rule Editor.

How to Model Rules When the Data Model is Deep

Use the following tips to avoid overly complex rules:

- Use rule test variables (inline aliases) to create a simple test.
- Any 1:1 prefix can be removed from the fact path.

Rule test variables:

Use rule test variables (inline aliases) to create a simple test that can help you model rules when there is a deep data model.

For example, a rule like this:

```
IF
task/payload/purchaseOrder/line.amount > 100
THEN
modify ...
```

Can be rewritten like this:

```
Root: task
IF
amount = task/payload/purchaseOrder/line.amount and
amount > 100.0
THEN
modify ...
```

(OR)

```
Root: task
IF
line = task/payload/purchaseOrder/line and line.amount > 100.0 and line.amount <
1000.0
THEN
modify ...
```

Remove 1:1 prefixes:

Note that any 1:1 prefix can be removed from the fact path (if not referenced in tests). For example, if you know that a task has at most 1 payload and a payload has at most one purchase order, and tests do not reference the task or the payload attributes, then you can use the shorter example as follows:

```
Root: PurchaseOrder
IF
line = PurchaseOrder/line and
line.amount > 100.0 and line.amount < 1000.0
THEN
...
```

You can also use the shorter path if the relationships are 1:many and you do not care about what task or payload contains which purchase order. You just want to process all the purchase orders.

5

Working with Decision Tables

This chapter describes how to use Decision Tables to create and use business rules in an easy to understand format that provides an alternative to the IF/THEN rule format. It also covers the various components of a Decision Table such as conditions, conflicts, actions, and discusses the various operations that you can perform on a Decision Table.

The chapter includes the following sections:

Introduction to Working with Decision Tables

Businesses invest in software to automate their business processes. Historically, this automation focused on the collection, presentation, and manipulation of data to facilitate human decision-making about that data. Increasingly, however, software designers and developers are called upon to automate the decision making process by putting detailed rules about business processes into software architectures. In addition, many enterprises are experiencing increasing pressure to make software systems more responsive to business changes.

In some cases, the role of writing and testing business rules is no longer assigned to software engineers, but is passed to trained business users. Alternatively, some organizations attempt to separate changes in the business behavior of software from the traditional software development cycles, and tie changes to business driven imperatives like product or sales cycles.

A Decision Table provides a mechanism for describing data processing tasks, especially when that description is done by business analysts rather than computer programmers.

The Decision Table format is intuitive for business analysts who are familiar with spreadsheets. The formal structure that a Decision Table provides makes it easier to author, understand, and change multiple similar rules and lets software check for rule completeness and consistency.

Oracle Business Rules Decision Tables provide the following features:

- Powerful Visualization: Compact and structured presentation. This visualization matches the way real world business policies are expressed: with many tables, declarative, and organized into simple steps.

- Error Prevention: Avoids incompleteness and inconsistency. Because a Decision Table is well structured, automated tools can check for conflicts, redundancy, and incompleteness to speed development of valid, consistent business rules.
- Modular Knowledge Organization: Group rules into a single table. A spreadsheet metaphor puts groups of rules that work together onto a single viewable pane. For example, if there are six rules that check an applicant's eligibility, it is more convenient to see all the rules than to view the rules as individual but related rules.
- Optimization of Rules and Performance Benefits: Oracle Business Rules Decision Tables provide automated features that can reduce the number of required rules, as compared to the IF/THEN rules (this is called rule coalescing).
- Rule Validation and Verification: Provides capabilities for ensuring the logical consistency of rules before deployment. Automated tools for checking conflicts or incompleteness help speed development of valid, consistent business rules.

Ease of verification and visualization are the major reasons for using Decision Tables.

For information, see Working with Rulesets and Rules.

What is a Decision Table?

A Decision Table displays multiple related rules in a single spreadsheet-style view. In Rules Designer a Decision Table presents a collection of related business rules with condition rows, rules, and actions presented in a tabular form that is easy to understand. Business users can compare cells and their values at a glance and can use Decision Table rule analysis features by clicking buttons and selecting values in Rules Designer to help identify and correct conflicting or missing rules.

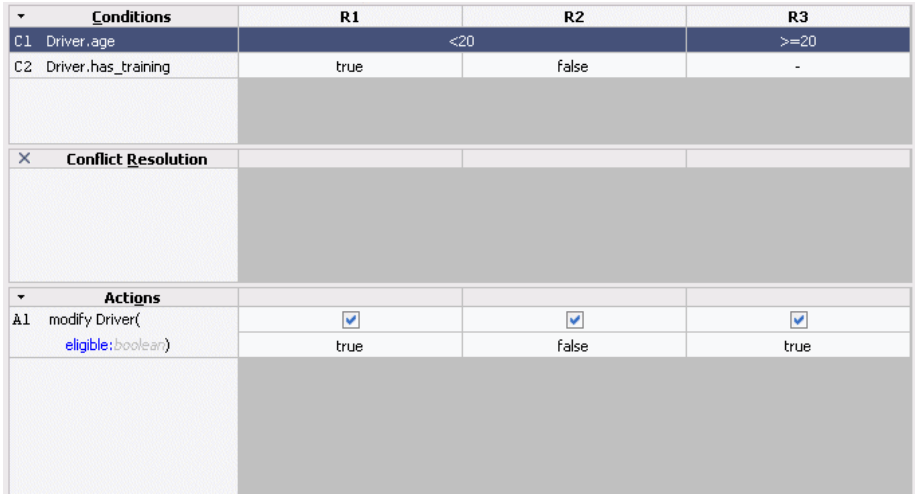
To help understand Decision Table concepts, consider a set of IF/THEN rules that determines if a driver is eligible for a license, and an equivalent Decision Table. Note if a driver has taken a driver training class then the driver has training certification.

The IF/THEN rules follow:

```
if driver.age < 20 and driver.has_training then training = true
if driver.age < 20 and driver.has_training = false then driver.eligible = false
if driver.age >= 20 then driver.eligible = true (do not care about training for this case)
```

Figure 5-1 shows a Decision Table representation of these rules that includes areas for Decision Table **Conditions** and **Actions**.

Figure 5-1 Sample Decision Table with Conditions and Actions



What You Need to Know About Decision Table Conditions

The **Conditions** area in a Decision Table includes one or more condition rows. Each condition row has a condition expression and, for each rule, a condition cell. A **condition expression** is an expression that you build in Rules Designer. The condition expression is often a fact property or a function result, but it can be any expression that has a type that can be associated with a value set. Test expressions are often used, such as `Driver.age<16`. These expressions are associated with the built-in boolean value set, with values `true` and `false`. The value or the range for a given **condition cell** takes its value or its range from one or more values or ranges in the associated LOV or Ranges value set. For more information on value sets, see Working with Value Sets.

For example, Figure 5-1 shows the condition expression for a `Driver` fact with the `Driver.age` property. The corresponding row in the Decision Table shows condition cells including values for the ranges `<20`, and `>=20`. The values in the cells come from the global value set named `driver_ages`.

Figure 5-1 also shows a condition row for the `Driver` fact with the `Driver.has_training` property. This condition row shows condition cells with the values, true, false, and -. The hyphen (`-`) means "do not care" (that is, `Driver.has_training` could be `true` or `false` in this case). The values for these condition cells come from the default value set associated with boolean types (this consists of default values for the values `true` and `false`).

The '-' (don't care) value is useful to ensure that a decision table will not have gaps when new values are added to a value set. For example, if a valueset initially contains 1, 2, and otherwise, a rule matching otherwise will fire if the input is 3. But after 3 is explicitly added to the valueset, then otherwise no longer matches an input value of 3. If no rule contains a '-' for this input, then no rule will fire when the input value is 3 and the decision table is said to have a gap.

Use 'otherwise' when you explicitly want to match the 'otherwise' value in the valueset, and not any other value. 'Otherwise' is useful to avoid conflicts in a decision table. '-' is used to match any value, and will often cause conflicts. These conflicts can be automatically resolved using the 'auto override' conflict policy.

Decision Tables show rules in bucket order, and to change the order of rules you need to change the order of buckets in the value set. Thus, the order of the buckets in the value set associated with a condition row determines the order of the condition cells,

and thus the order of the rules. You can control rule ordering in a Decision Table by changing the relative position of the buckets in an LOV value set associated with a condition row; however, you cannot reorder range buckets (values). For information on ordering buckets in a value set, see How to Define a List of Values Global Value Set.

What You Need to Know About Decision Table Actions

Actions are associated with rules in a Decision Table. At runtime, when facts match for condition cells, the Rules Engine prepares to run the actions associated with the rule.

Table 5-1 shows the types of actions you can choose in the **Actions** area. Thus, in an action you can call a function, assert a new fact, retract a fact, or modify a fact, and so on. In the **Actions** area the cells corresponding to an individual action for a rule are called **action cells**. For more information on advanced mode, see How to Select the Advanced Mode Option.

Table 5-1 Decision Table Actions for Action Cells

Action	Description
`assert new`	Assert a new fact.
`assign`	Assign a value to a variable.
`call`	Call a function.
`modify`	Modify a data value associated with a matched fact.
`retract`	Retract a fact.
`assert`	Assert a fact.
`assert tree`	Asserts a tree of facts given the root.
`assign new`	Assign a value to a new variable.
`expression`	Perform expression.
`return`	The return action returns from the action block of a function or a rule. A return action in a rule pops the ruleset stack, so that execution continues with the activations on the agenda that are from the ruleset that is currently at the top of the ruleset stack.
`throw`	Throw an exception, which must be a Java object that implements java.lang.Throwable. A thrown exception may be caught by a catch in a try action block.

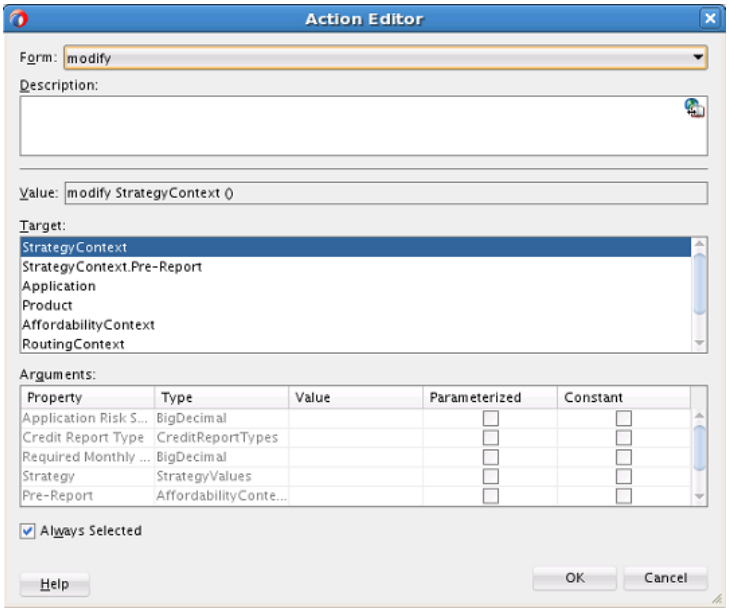
When you add multiple actions the actions that you add in the **Actions** area are ordered; actions appearing in the higher rows run before actions in the following rows.

The Decision Table actions such as `modify` can refer to facts matched in the condition cells. For example, given a Decision Table with condition rows on the `Driver` fact that includes condition rows for `Driver.age` and `Driver.has_training`, actions can modify the property `Driver.eligible` and you can specify a value for `Driver.eligible` for each action cell.

Certain types of actions in the **Actions** area include a **Parameterized** check box. This check box specifies that a property from the action can have its value set in the action

cell associated with a rule in the Decision Table. When the parameterized check box is selected, the value you supply for the expression value in the action, in the **Actions** area, becomes the default value for the property if a value is not supplied in the action cell. For example, see Figure 5-2 where the value `false` is assigned as the default value for the action property `eligible`.

Figure 5-2 Action Editor Showing Parameterized Action with Default Value



What You Need to Know About Decision Table Rules

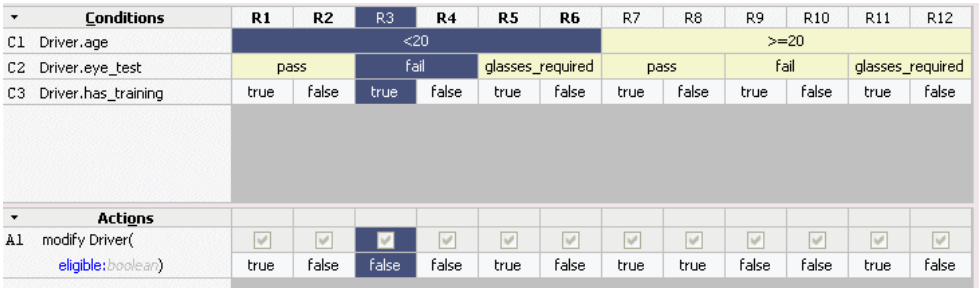
A ruleset contains a Decision Table; this provides a way to group the Decision Table along with IF/THEN rules. When rules and Decision Tables are grouped in a ruleset, the IF/THEN rules and the Decision Table rules all execute as a set of interrelated rules.

A rule in a Decision Table is not named. Although Rules Designer shows rules in a Decision Table with labels, for example, R1, R2, and R3, these rule labels are not names for individual rules but are labels derived from the current ordering of the rules in the Decision Table. Thus, a rule with the label R1 could be moved to position 3 and then Rules Designer relabels this rule R3.

Rules in a Decision Table are organized as a table that contains a tree of condition cells. The condition cells in the first row span the cells of later condition rows. A parent cell in row *i* spans its children in row *i*+1.

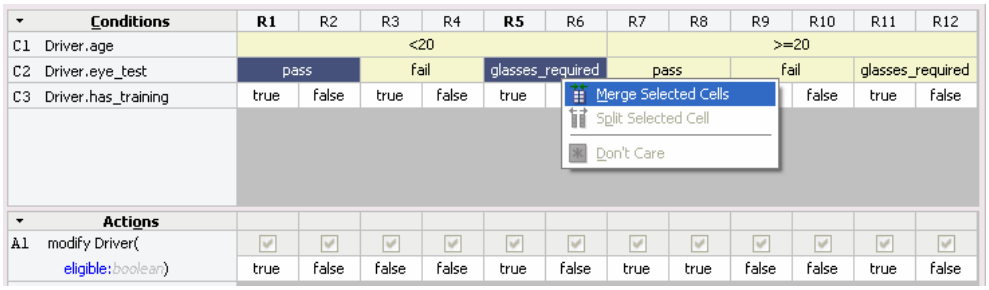
Figure 5-3 shows rules in a Decision Table where each rule consists of one cell from each row in the **Conditions** area, and an associated action cell in the same column in the **Actions** area. Figure 5-3 shows the rule with the label R3 defined by the first cell from condition 1 (the `Driver.age < 20` value), the second cell from condition 2 (the `Driver.eye_test` equals `fail` value), and the third cell from condition 3 (the `Driver.has_training` equals `true` value). Likewise for each of the other rules, R1 to R12, there is a unique path through the Decision Table.

Figure 5-3 Rules in a Decision Table



As shown in Figure 5-3, it is significant for a cell to be a parent of another cell and a parent cell spans lower cells. In the **Conditions** area, when condition cells have the same parent condition cell the cells are called **siblings**. Certain operations only apply for condition cells that are siblings. For example, Figure 5-4 shows two sibling cells that are selected; with these cells selected the **Merge Selected Cells** operation is valid. For these cells, the corresponding value set with the value `fail` for `Driver.eye_test` is also a sibling (as shown in the R3 and R4 columns in Figure 5-4). For more information, see How to Merge or Split Conditions in a Decision Table.

Figure 5-4 Sibling Condition Cells in a Decision Table

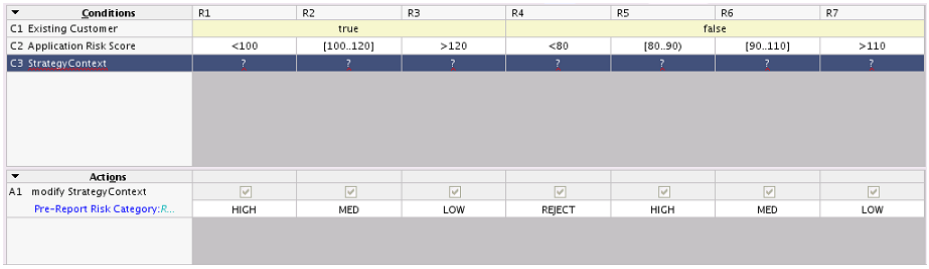


Rules Designer lets you easily reorder rows by selecting the row and clicking a **Move** button. By reordering rows in the **Conditions** area you can perform operations on condition cells at the desired granularity. Thus, the move operations can assist you when you want to split, merge, or assign certain values that might only be appropriate at a particular level in the tree, depending on the location of a condition cell or depending on the location of the parent, children, or siblings of a condition cell.

Understanding Condition Cell Values

By default, when you create a condition row, Rules Designer creates a single condition cell and assigns the "?" value to the cell. A condition cell with the value "?" indicates that the value of the cell is undefined in the value set. For example, Figure 5-5 shows a "?" value for `StrategyContext`. Note that contiguous value ranges in a condition cell are combined. For example, if you select <20 and [20..40] it will display as <=40.

Figure 5-5 Sample Decision Table Showing Undefined in Condition Cell



Understanding Action Cell Values

In the Decision Table **Actions** area you can specify that an action cell "do nothing." In this case, clear the action cell. When the action cell check box is cleared, this means do not perform this action when the pattern matches for the specified condition values in the Decision Table. Thus, for each action cell you can specify whether the rule associated with the action cell should activate the action, or does not perform the action.

In a Decision Table, when a condition cell represents a value that has been removed from the value set, Rules Designer provides a validation warning such as the following:

```
RUL-05831: Decision table value reference not found
```

To fix this type of validation warning you can do one of the following:

- Define a value by double-clicking the condition cell and selecting one or more values from the list.
- Add the missing value to the value set or associate the condition with another value set that contains the missing value.

What You Need to Know About Decision Table Loops

A Decision Table loop occurs when the value for a condition row is changed by an action. Loops can occur across the rules in a single Decision Table or spread over several Decision Tables, or spread over rules and Decision Tables in the same ruleset. Try not to create Decision Table actions that modify fact properties that are used in Decision Table conditions. This could cause an infinite loop.

Note:

You can prevent infinite loops by using the rule firing limit on the containing decision function.

Creating Decision Tables

You add a Decision Table by performing several steps.

These steps include creating a Decision Table, creating value sets, and then adding conditions and actions to Decision Table, and using the Decision Table to operate to validate, correct, and modify the Decision Table.

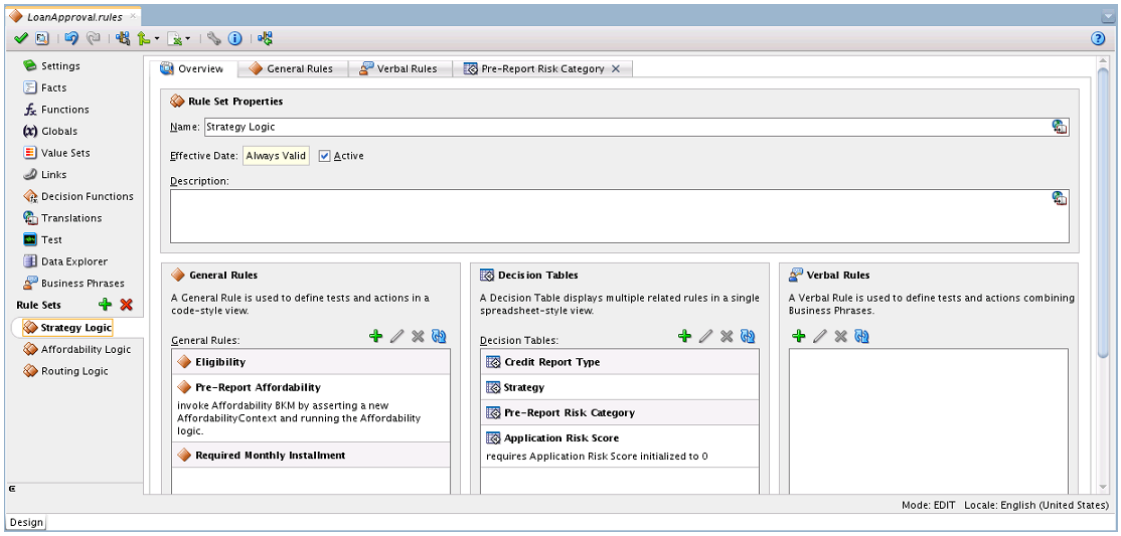
How to Create a Decision Table

To work with a Decision Table, start by creating a Decision Table in a ruleset.

To create a decision table:

1. From Rules Designer select an existing ruleset from the rulesets tab or create a ruleset by clicking **Create Rule Set...**.
2. Click **Create** from the **Decision Tables** area on the **Overview** tab, as shown in Figure 5-6. This creates a Decision Table.

Figure 5-6 Adding a Decision Table



Note:

When you add a Decision Table the rules validation log displays validation warnings. The Decision Table is not complete and does not validate without warnings until you add conditions and actions to the Decision Table.

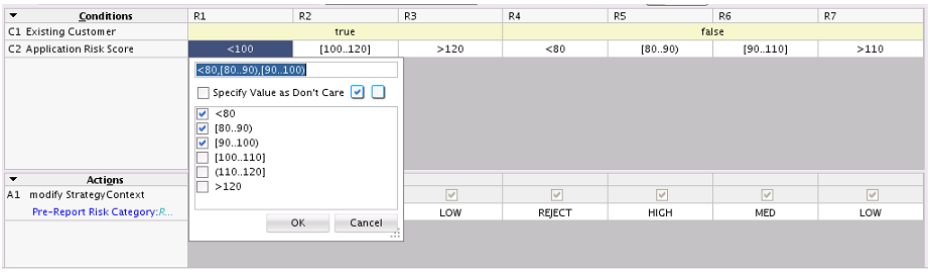
How to Add Condition Rows to a Decision Table

A Decision Table includes a **Conditions** area where you specify Decision Table condition rows. The condition rows determine the facts that the Oracle Rules Engine matches at runtime. To create a Decision Table you need to add one or more condition rows to the Decision Table.

To add condition rows to a decision table:

1. From Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to add conditions.
2. In the Decision Table area, from the list next to the **Add** button select **Condition**.
3. In the **Conditions** area, double-click **<edit-condition>** to display the navigator to select or enter an expression as shown in Figure 5-7.

Figure 5-7 Adding a Condition to a Decision Table



4. Enter an expression by clicking in the navigator to select a variable or click the **Expression Builder** button to display the **Expression Builder** window. The **Expression Builder** lets you build expressions.

5. Each condition row requires a value set from which to draw the values for each cell. When the value you select has an associated global value set, then by default the value set is associated with the condition row.

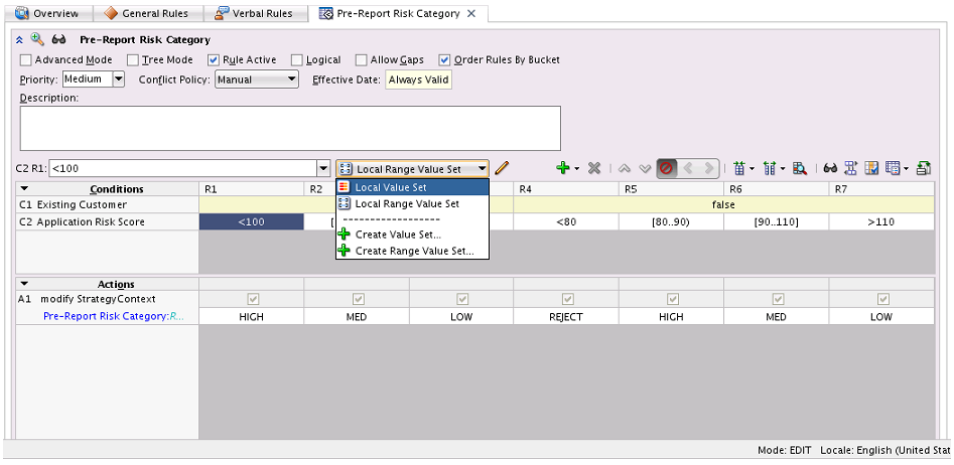
6. Repeat **Step 2** through **Step 5**, as required to add additional condition rows in the Decision Table.

How to Use or Specify the Value Set for a Decision Table Condition

1. Each condition row requires a value set from which to draw the values for each cell. When the value you select has an associated global value set, then by default the value set is associated with the condition row.

2. If there is no global value set associated with the value, then after you add a condition row to a Decision Table you need to specify either a Local List of Values or a Local List of Ranges value set to associate with the condition row, or specify an existing global value set. To add a value set for the condition, in the **Conditions** area select the condition and then select from the value set list to associate a value set, as shown in Figure 5-8. The value set list includes available global value sets of the appropriate type.

Figure 5-8 Specifying a Value Set For a Condition Row in a Decision Table



3. If you do not specify a global value set, then you can create and use a local value set by selecting either **Local Value Set** or **Local Range Value Set** to create and use the specified type of value set.

4. Repeat **Step 2** through **Step 3**, as required to define additional condition rows in the Decision Table.

For more information on creating value sets, see Working with Value Sets.

How to Add Actions to a Decision Table

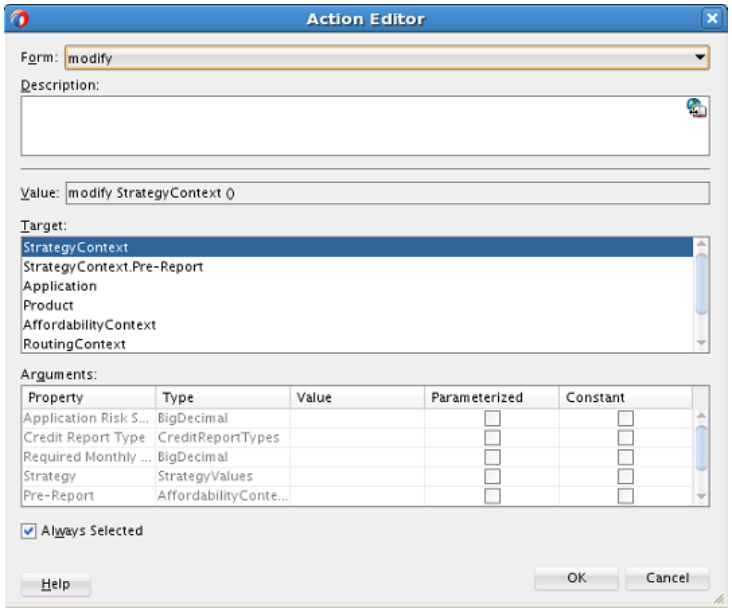
A Decision Table includes an **Actions** area where you specify Decision Table actions. The actions determine actions for rules in a Decision Table. To create a valid Decision Table, add actions to a Decision Table. For each action cell, where specific values apply, set the values for the action cells. For each action cell, when the action does not apply to the rule, deselect the action cell. By default when you add an action to a Decision Table, actions for all the rules are unselected

To add actions to a decision table:

1. From Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to add actions.

2. From the list next to the **Add** button, select **Action** and select an available action from the list. Table 5-1 lists the available actions. For example, select **Modify**. Rules Designer displays the **Action Editor** dialog as shown in Figure 5-9.

Figure 5-9 Adding an Action to a Decision Table



3. In the **Action Editor** dialog select the action target in the **Target** area. This specifies the data model object the action applies to.

4. For the specified target, as needed to make the action do what is required, modify the fields in the **Arguments** table. In the Action Editor dialog the **Arguments** table includes the fields shown in Table 5-2.

Table 5-2 Action Editor Dialog Arguments Fields

Field	Description
Property	Displays the property names for the specified target.
Type	Displays the type for the property.
Value	Select the default value for the action from the list of available actions. The specified value applies to either the entire action, as the default value, or when a particular action cell is selected, the value specified applies for that particular action cell.
Parameterized	This specifies a parameterized value. A parameterized value displays in a Decision Table action cell. When you select parameterized value for a property, this generally means that each rule supplies a different parameter value.
Constant	Select to specify a constant value.

5. In the Action Editor dialog, to select action cells for all the rules, select the **Always Selected** check box.
6. Repeat **Step 2** through **Step 5**, as required to define additional actions for the Decision Table.

How to Set Values for Action Cells in a Decision Table

To set values for action cells:

1. From Rules Designer, select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to specify action cell values.
2. In the **Actions** area, check that the appropriate action cells are selected. If the **Always Selected** check box is specified in the Action Editor dialog, then all action cells should be selected. If **Always Selected** is not selected, then select the appropriate action cells using the action cell check box.
3. In the **Actions** area, enter the appropriate value for parameterized properties for each selected action cell. To do this select the action cell property cell, and either enter a value, select a value from the list, or click the **Expression Builder** button to use the Expression Builder dialog.

   For more information on referring to a value set from a Decision Table, see How to Define a List of Ranges Global Value Set.

How to Deselect an Action Cell in a Decision Table

To deselect an action cell:

1. From Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want deselect an action cell.

2. In the **Actions** area, select the action cell and deselect the check box in the action cell. You are not allowed to deselect action cell values when **Always Selected** is selected for the action.

When you add actions, you may need to change the order of the actions. In Rules Designer you can use the **Move Down** button or **Move Up** button to change the order of actions.

How to Add a Rule to a Decision Table

You can add a rule to a Decision Table. Rules Designer adds a column for the rule to the left of the existing rules and each condition cell is initialized to "?", which actually means a validation error prompting you to populate the cell with relevant values.

To add a rule to a decision table:

1. From Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to add the rule.
2. From the list next to the **Add** button, select **Rule**.
3. Enter values for the condition cells. Notice that the new rule is added as the first rule of the Decision Table on the left and the other rules have moved as required to keep the values in their defined order.
4. Enter values for the action cells.

The **Order Rules By Bucket** check box under the Advanced Settings of a Decision Table is selected by default. In this case, the Decision Table layout changes automatically on adding new rules.

When you add a new rule to a Decision Table, the new rule is added as the first rule of the Decision Table and the other rules move as required to keep the values in their defined order. This is because **Order Rules By Bucket** is enabled, which means rule ordering in a Decision Table is set according to the relative position of values associated with a condition expression. If **Order Rules By Bucket** is not enabled when you add a rule, the new rule is added as the last rule of the Decision Table. In either case, the cells in the new rule column have "?" symbols, indicating the cells do not have values yet.

Note:

When **Order Rules By Bucket** is selected, the rules are ordered and duplicate rules (rules with exactly the same values) are combined. So, you cannot add two rules without any values to a Decision Table, because in that case, the rules are duplicates and would immediately be combined. When **Order Rules By Bucket** is cleared, then duplicate rules are allowed.

In addition, the **Move** buttons pertaining to a rule column are also enabled. You can use them to reposition rules. Use the **Flip the Table Rows and Columns** button to change the view of the Decision Table. This also affects the Move buttons: the move direction may be **Up** or **Down**, **Left** or **Right**. The **Merge**, **Compact** and **Span** options are also enabled. You can also cut, copy, or paste rules.

For more information, see Introduction to Decision Table Operations.

How to Define Tests in a Decision Table

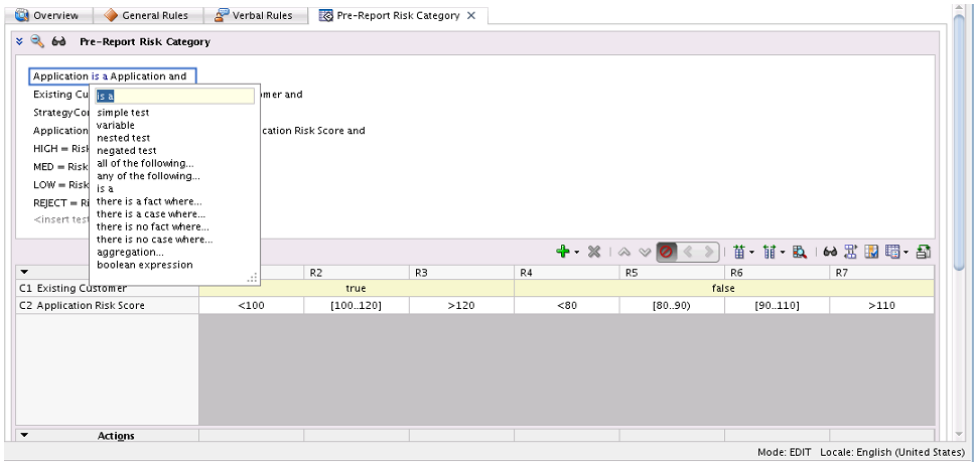
You can define tests in a Decision Table. The tests must evaluate to true for any rule in the decision table to fire. For more information about defining tests and working with rule conditions, see Working with Rules.

You can use the **Data Explorer** tab to find fact types and value sets in the data model.

To add tests to a Decision Table:

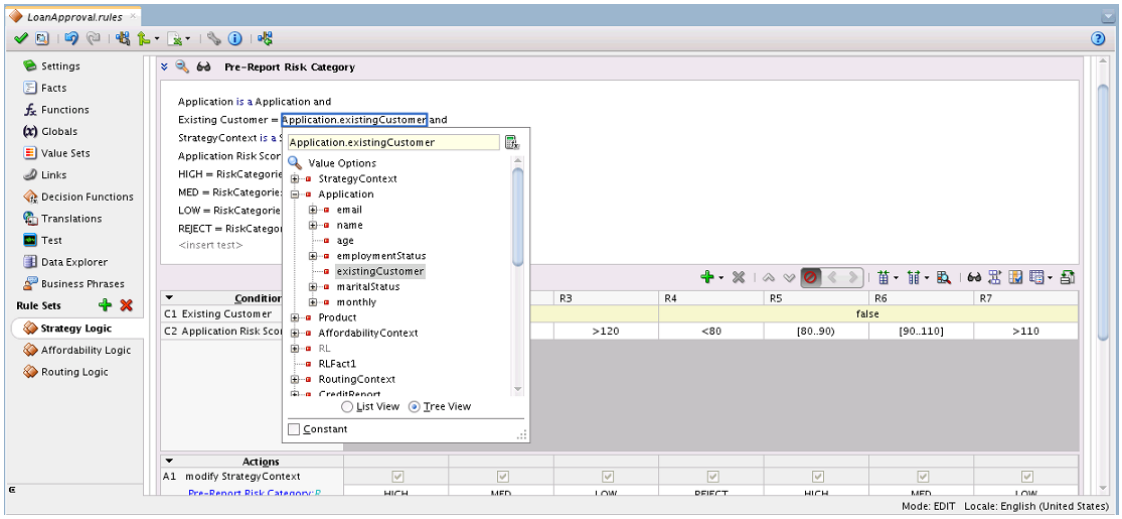
1. From Rules Designer, select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to add the rule.
2. Click the **Show Patterns/Tests** button (magnifying glass) left of the Decision Table name. If **Advanced Mode** is selected, clear the check box.
3. Select any of the options according to your requirements, as shown in Figure 5-10. Note that variables without any tests are often used so that the variables can be used in the decision table conditions and actions.
   - simple test
   - variable
   - nested test
   - negated test
   - all of the following...
   - any of the following...
   - is a
   - there is a fact where...
   - there is a case where...
   - there is no fact where...
   - there is no case where...
   - aggregation...
   - boolean expression

Figure 5-10 Options List



4. Click the left and the right **<operand>** to enter the operand values, and the operator list to select an operator, as in Figure 5-11:

Figure 5-11 Value Options List



For more information about writing tests, see Testing and Validating Business Rules.

Introduction to Decision Table Operations

After you create a Decision Table you may want to modify the contents of the Decision Table to produce a Decision Table that includes a complete set of rules for all cases, or to produce a Decision Table that provides the least number of rules for the cases.

After you create a Decision Table there are operations that you may want to perform on the Decision Table, including the following:

- Compact or split cells in a Decision Table.
- Merge a condition or split a condition in a Decision Table.
- Finding and resolving conflicts between rules in a Decision Table.
- Find and fix gaps (missing rules) in a Decision Table.

Understanding Decision Table Split and Compact Operations

The split and compact operations enable you to manipulate the contents of the condition cells in a Decision Table.

The split table operation creates a rule for every combination of values across the conditions. For example, in a Decision Table with 3 boolean conditions, 2 x 2 x 2 = 8 rules are created. In a Decision Table with 32 boolean conditions, 2**32 ~ 2 billion rules are created. Thus, you only use split table when the number of rules created is small enough that filling in the action cells is feasible.

When you want to apply match conditions for the "do not care" values in a Decision Table and create a match case for each cell, you use the split table operation.

Split can be applied to an entire Decision Table or to a single condition row. Additionally, split may be performed on an individual condition cell.

Depending on what is selected in the Decision Table, the split operation can create condition cells. Thus, using the split operation you can create rules in a Decision Table. Table 5-3 summarizes the split operation for a selected condition cell, condition row, or for a complete Decision Table.

Table 5-3 Summary of Split Operation

Operator	Description
Condition Cell	Creates one sibling condition cell for each value represented by the cell. If the condition cell value is "do not care", then the cell is split into one sibling cell for each value in the valueset that is not represented by a sibling condition cell, and "do not care" is no longer represented.
Condition Row	For each condition cell in the proceeding condition expression, create a sibling group which contains a cell for each value in the value set. The effect of this operation is the same as adding a "do not care" to each sibling group and calling split on each condition cell in each sibling group.
Decision Table	Same as calling split on each condition row in the Decision Table.

Depending on what is selected in the Decision Table, the compact table or merge cells operations remove condition cells. The compact table operation can be applied to an entire Decision Table. Additionally, the merge operation may be performed on sibling cells or on an entire condition row. Thus, using compact table or merge you can remove rules from a Decision Table. Table 5-4 summarizes the compact table and merge operations.

Table 5-4 Summary of Merge Operation

Operator	Description
Condition Cell	Merging two or more condition cells adds all values in the cells to a single cell, and removes all but one of the cells. If one of the cells represents "do not care", then the merged cell represents "do not care". This operation may merge action cells and this can create warnings for duplicate action cells, such as, `RUL-05847: Duplicate decision table action parameter`.

Table 5-4 (Cont.) Summary of Merge Operation

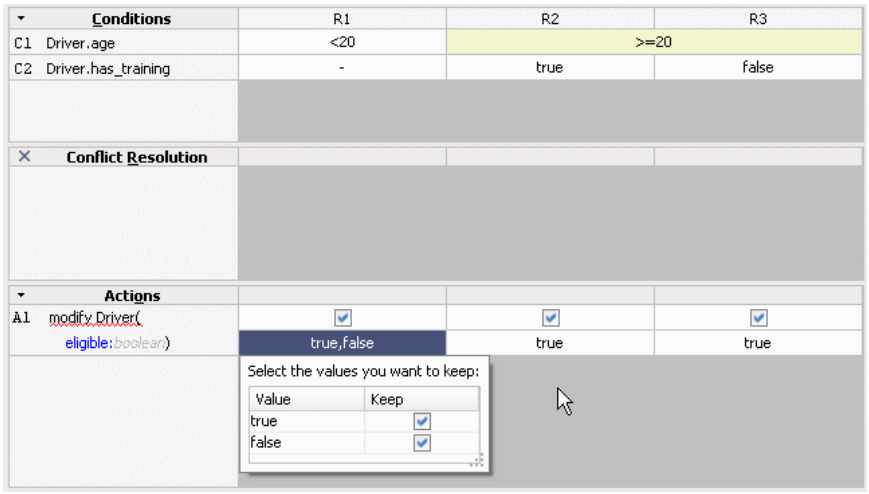
Operator	Description
Condition Row	Combine all values in each sibling group into a single "do not care" cell for each condition cell in the proceeding condition expression. The effect of this is the same as calling merge on all cells in each sibling group. This operation may merge action cells and this can create warnings for duplicate action cells, such as, `RUL-05847: Duplicate decision table action parameter`.
Decision Table	Compacts the Decision Table by merging conditions of rules with identical actions.

Split and merge are inverse operations when conflicting action cells are not associated with the operation. In this case, without conflicting action cells, a merge operation combines all the values from the siblings into one sibling, and discards the other sibling condition cells, and as a result of merging the condition cells, when a Decision Table contains action cells, the action cells are also merged. Thus, the merge operation combines multiple condition cells into a single condition cell and adds all values to the single cell.

When there are conflicting values for the action cells, a merge operation merges the action cells in a form that requires additional manual steps. Thus, if two action cells have conflicting parameters, after the merge the action cell contains multiple conflicting parameter values. These conflicting values are appended to the action cell and must be manually resolved by selecting and deleting the unwanted duplicate parameters. For example, see Figure 5-12 that shows conflicting values in an action cell.

An action cell that contains multiple values for a property is invalid. When you select the action cell Rules Designer shows a popup window with check boxes to allow you to select a single value for the action cell. As shown in the validation log in Figure 5-12, Rules Designer shows a validation warning until you select a single value.

Figure 5-12 Conflicting Properties to be Resolved for a Merged Action Cell



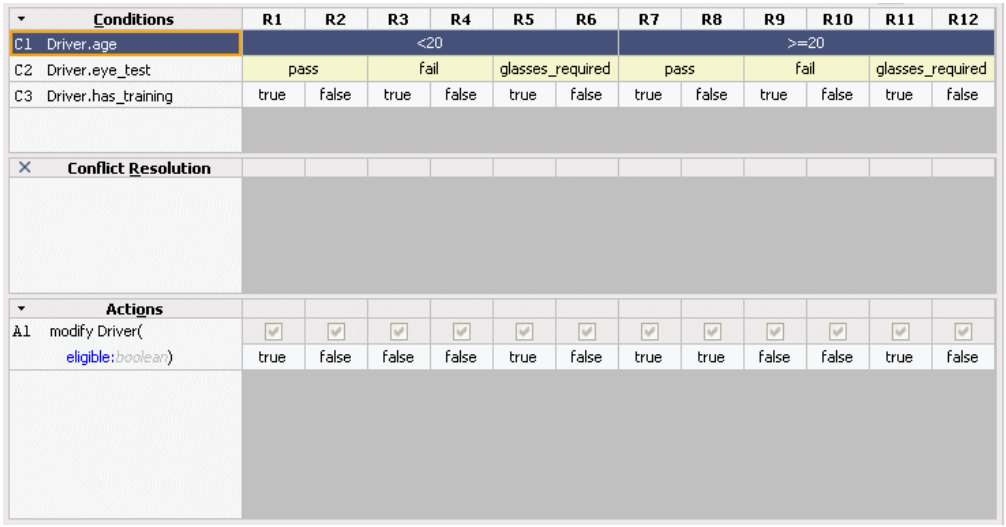
Understanding Decision Table Move Operations

You can move the conditions or actions in a Decision Table. The **Move** buttons let you reorder condition rows in the **Conditions** area and actions in the **Actions** area. Moving conditions up or down may reorder visual display of the rules, but these operations

does not change the logic in any way. For example, if (`x.a == 1 and x.b == 1`) is logically the same as if (`x.b == 1 and x.a == 1`).

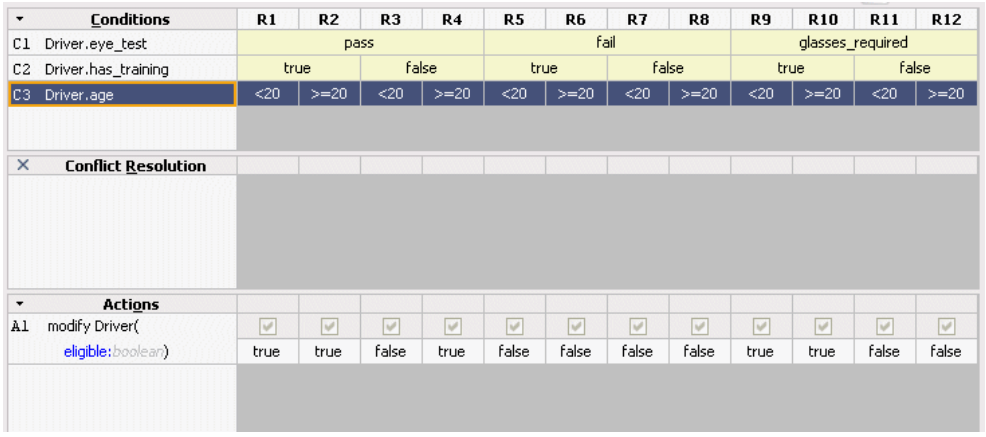
When you work with Decision Tables some operations only apply for condition cells that are siblings. Using the **Move** button you can reorder rows so that Decision Table operations apply to the tree at the desired granularity. For example, when you want to change the action of a condition cell for a single rule, then you need to move that condition cell to the last row in the Decision Table **Conditions** area. For example, consider the Decision Table shown in Figure 5-13.

Figure 5-13 Rules in a Decision Table



To view this table with granularity for the `Driver.age`, move the `Driver.age` condition from the first row to the third row, as shown in Figure 5-14.

Figure 5-14 Decision Table After Move Down with Age Condition Last



Now to make the `Driver.age` conditions "do not care" for the first two rules, where the driver passes the eyesight test and has had driver training is true, you can easily apply changes to these particular conditions when the `Driver.age` condition is in the last row. Thus, in this table, it is easier to view and modify age related rules when `Driver.age` is in the last row, with the finest granularity.In general, the move operations can assist you when you want to split, merge, or assign certain values that might only be appropriate at a particular level in the tree, depending on the location of a condition cell, or depending on the location of the parent, children, or siblings of a condition cell.

For actions in the **Actions** area, clicking **Move Up** or **Move Down** lets you reorder the actions. Actions are ordered so that when multiple actions apply, the first action runs before subsequent actions. Thus, using the **Move Up** or **Move Down** operation on an action may be appropriate, depending on your application.

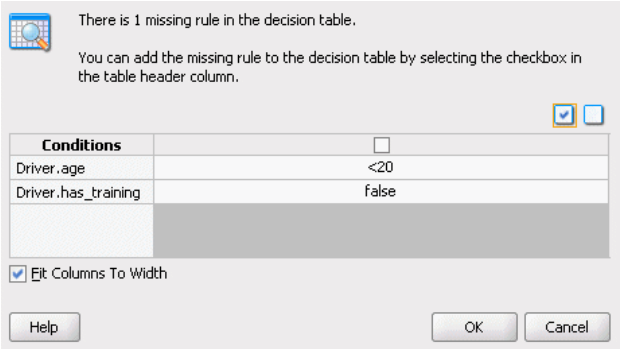
Understanding Decision Table Gap Checking

A gap is a "missing" rule in a Decision Table. A Decision Table has a gap if there is a combination of values, one from each condition, that is not covered by an existing rule. Rules Designer provides Gap Checking to check for gaps. When you click the **Gap Analysis** button, Rules Designer finds gaps and presents a dialog to fix any gaps that are found.

You can choose to make existence of gaps a validation warning. When you clear **Allow Gaps** in the **Advanced Settings** area, the Decision Table reports a validation warning when a gap is found. For more information, see Using Advanced Settings with Rules and Decision Tables.

For example, using the Driver example if you create a gap by deleting the rule that covers the case for `Driver.age` < 20 and `Driver.has_training` `false`, and then you click **Gap Analysis**, Rules Designer shows the Gap Analysis dialog as shown in Figure 5-15. Clicking **OK** with the check boxes selected adds either all rules or the selected rules to the Decision Table (this example only shows a single rule to add).

Figure 5-15 Checking Gaps



Gap checking generates different new rules for the following cases:

- Sibling rules: multiple missing sibling rules are added as a single new rule. For example, consider a rule with two conditions, `Driver.age` and `Driver.hair`. When there are two missing rules for different hair colors and the rules are siblings, that is, they have a common parent, then gap checking shows a single rule as shown in Figure 5-16.
- Non-sibling rules: multiple missing non-sibling rules are added as individual new rules. For example, when there are two different rules missing that do not have the same parent, then gap checking provides two rules, as shown in Figure 5-17.

Figure 5-16 Gap Checking with Missing Sibling Rules

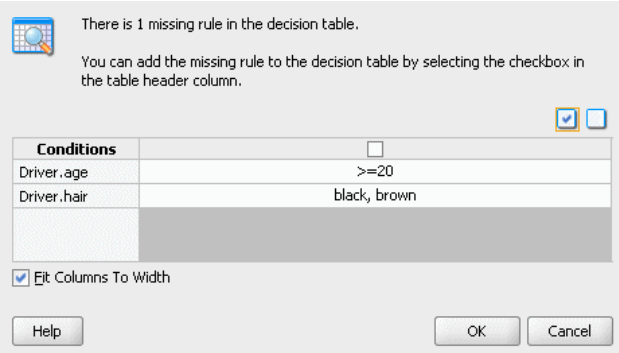
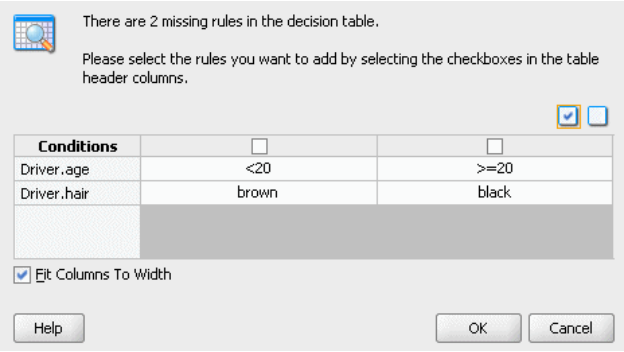


Figure 5-17 Gap Checking with Missing Non-Sibling Rules



In both of these cases shown in Figure 5-16 and Figure 5-17 there are two missing values, but for sibling rules the multiple values are combined in a single new rule. Thus, in general gap checking suggests fewer more general rules in preference to many more specific rules.

For sibling rules you can add multiple rules then edit each cell to pick the values you want. Alternatively, you can use **Find Gaps** to add a rule and then split the cell with multiple values, and delete the rules you do not want to keep.

Understanding Decision Table Conflict Analysis

The rules in a Decision Table can conflict. Two rules conflict when they overlap and they have different actions. Two rules overlap when at least one of their condition cells has a value in common. Overlap is common when a Decision Table contains "do not care" condition cells. Overlap without conflict is common and harmless.

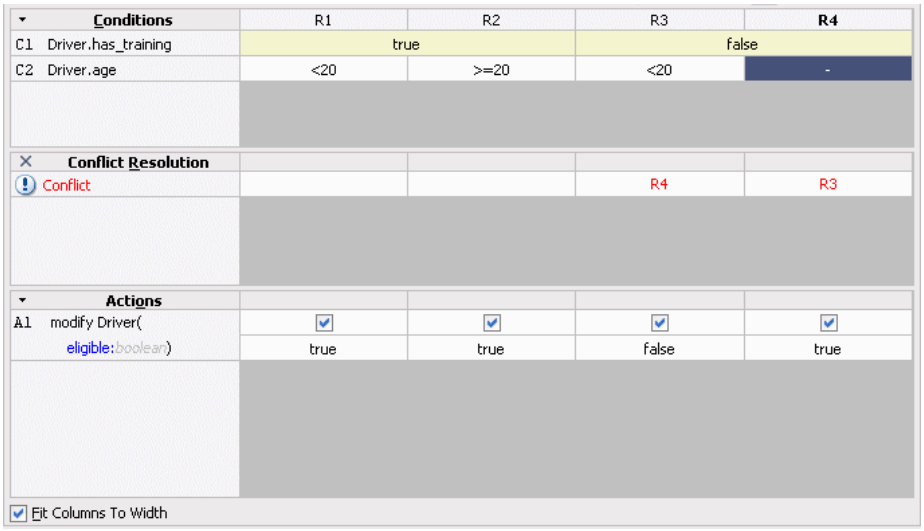
Rules Designer finds conflicts and you can see the conflicts in the **Conflict Resolution** row in the Decision Table when you click **Show Conflicts**. How you handle and resolve conflicts depends on the specified conflict policy. You can choose a conflict policy or use the default manual conflict policy. When you set a conflict policy using the **Conflict Policy** option in the **Advanced Settings** area, Rules Designer sets the conflict policy for the Decision Table. The **Conflict Policy** specifies the Decision Table conflict policy and is one of the following:

- **manual**: Conflicts are resolved by manually specifying a conflict resolution for each conflicting rule.
- **auto override**: Conflicts are resolved automatically using an override conflict resolution when this is possible, using the Oracle Business Rules automatic conflict resolution policies.

- **ignore**: Conflicts are ignored.

For more information, see Using Advanced Settings with Rules and Decision Tables. For example, Figure 5-18 shows a Decision Table with conflicting rules that you resolve with the default manual conflict policy.

Figure 5-18 Decision Table Showing Conflicting Rules in the Conflicts Area



By clicking on the cells in the Decision Table **Conflict Resolution** area Rules Designer lets you resolve conflicts between rules as follows:

- Override (**Override** and **OverriddenBy**): You override one rule with the other. Override specifies that one rule fires. Override is a combination of prioritization and mutual exclusion. Prioritization is transitive and not symmetric. Mutual exclusion is both transitive and symmetric. If A overrides C and B overrides C, then A or B runs before C but only one of A, B, or C runs.

- Run Before (**RunBefore** and **RunAfter**): You prioritize the rules. Run before lets the two rules fire in a prescribed order. Prioritization is transitive but not symmetric. That is, if A runs before B runs before C, then A runs before C but B does not run before A. This uses a Decision Table runBefore list specifying that the rule that runs before has a higher priority than rules in the list.

- Ignore (**NoConflict**): You OK the conflict. Ignore lets the two rules fire in arbitrary order. For example, consider the following conflicting rules in a decision table:

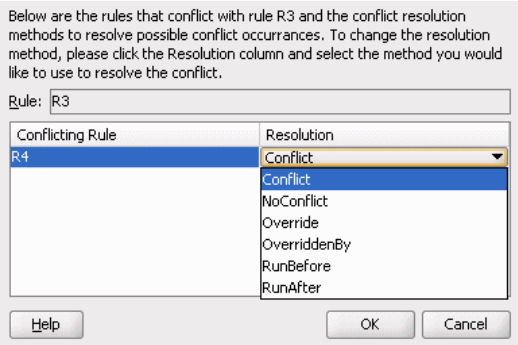
  ```
  rule1: everybody gets a 10% raise (as specified with a do not care value in a decision
  table condition cell)
  rule2: employee with Top Performer set to true gets a 5% raise
  ```

  In these rules, if rule2 overrides rule1, then a top performer gets a 5% raise, and everyone else gets a 10% raise. However, in this case, you would like to have both rules fire. Because it does not matter which rule fires first, and there is no conflict, then a top performer gets a 15.5% raise either way. In this case, use the NoConflict list to remove the conflict. Note that no conflict is what you get with IF/THEN rules with equal priorities, only you are not warned of a conflict and you have to think carefully if you want one rule to override the other.

Figure 5-19 shows the Rules Designer Conflict Resolution dialog shown when you select a conflicting rule in the **Conflict Resolution** area. This dialog lets you resolve

conflicts between rules by selecting overrides, prioritization with RunBefore or RunAfter options, and a NoConflict option.

Figure 5-19 Using the Decision Table Conflict Resolution Dialog



You can use the Decision Table Advanced Settings **Conflict Policy auto override** option to specify that, where possible, conflicts are automatically resolved. The automatic override conflict resolution policy specifies that a special case overrides a more general case. For more information, see Using Advanced Settings with Rules and Decision Tables.

Thus, when there are conflicts in a Decision Table, you can do one or more of the following to resolve the conflicts:

- Use auto override conflict resolution by selecting the **Conflict Policy** and then **auto override** option in the Decision Table.
- Ignore conflicts by selecting the **Conflict Policy** and then **ignore** option in the Decision Table.
- Use manual conflict resolution by selecting the **Conflict Policy** and then **manual** option in the Decision Table and set Conflict Resolution for each conflicting rule in the dialog by selecting cells in the **Conflict Resolution** area with the **Show Conflicts** check box selected.
- Change the Decision Table to remove an overlap.
- Combine actions to remove conflicts.

How to Compact or Split a Decision Table

Use the **Compact Table** and **Split Table** buttons to compact or split cells in a Decision Table. For more information, see Understanding Decision Table Split and Compact Operations.

To compact or split cells in a decision table:

1. In Rules Designer, select a rule set from the **Rule Sets** navigation tab. On the **Overview** tab, select the Decision Table and click **Edit.**
2. Click the **Compact Table** button to compact or the **Split Table** button to split the cells.

How to Merge or Split Conditions in a Decision Table

Use the merge condition and split condition operations to merge or split a condition in a Decision Table. For more information, see Understanding Decision Table Split and Compact Operations.

To merge or split a condition in a decision table:

1. In Rules Designer, select a rule set from the **Rule Sets** navigation tab. On the **Overview** tab, select the Decision Table where you want to merge or split a condition and click **Edit.**
2. In the **Conditions** area, select the condition you want to merge or split.
3. Right-click, and from the list select **Merge Condition** or **Split Condition**.

How to Use the Condition Cell Operations

Use the condition cell operations to split a condition cell, to merge sibling condition cells, or to specify a "do not care" value for a condition cell in a Decision Table. For more information, see Understanding Decision Table Split and Compact Operations.

How to Merge Sibling Cells in a Condition in a Decision Table

1. In Rules Designer, select a rule set from the **Rule Sets** navigation tab. On the **Overview** tab, and select the Decision Table where you want to merge condition cells and click **Edit.**
2. Select the sibling condition cells to merge.
3. Right-click, and from the list select **Merge selected cells**.

How to Split a Cell in a Condition in a Decision Table

1. In Rules Designer, select a rule set from the **Rule Sets** navigation tab. On the **Overview** tab, and select the Decision Table where you want to split a condition cell and click **Edit.**
2. Select the cell to split.
3. Right-click, and from the list select **Split selected cell**.

How to a "Do Not Care" Value for a Cell in a Condition in a Decision Table

1. From Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to set the "do not care" value.
2. Select the appropriate condition cell.
3. Right-click, and from the list select **Do Not Care**.

How to Select all Value Sets to Specify a "Do Not Care" Value for a Cell in a Condition:

1. In Rules Designer, select a rule set from the **Rule Sets** navigation tab. On the **Overview** tab, and select the Decision Table where you want to set the "do not care" value and click **Edit.**
2. Select the appropriate condition cell.
3. Double-click, and from the list select all the available check boxes for all possible values.

How to Perform Decision Table Gap Checking

A gap is a "missing" rule in a Decision Table. A Decision Table has a gap if there is a combination of values, one from each condition, that is not covered by an existing rule. Rules Designer provides Gap Checking to check for gaps. When you use this operation Rules Designer presents a window to fix gaps. For more information, see Understanding Decision Table Gap Checking.

You can choose to make existence of gaps a validation warning. When you clear **Allow Gaps** in the **Advanced Settings** area, the Decision Table reports a validation warning when a gap is found. For more information, see Using Advanced Settings with Rules and Decision Tables.

To perform decision table gap checking:

1. In Rules Designer, select a rule set from the **Rule Sets** navigation tab. On the **Overview** tab, and select the Decision Table where you want to perform gap checking and click **Edit.**
2. Click the **Gap Analysis** button.

How to Perform Decision Table Manual Conflict Resolution

The rules in a Decision Table can conflict. Two rules conflict when they overlap and they have different actions. Two rules overlap when at least one of their condition cells has a value in common. For more information, see Understanding Decision Table Conflict Analysis.

To perform manual decision table conflict resolution:

1. In Rules Designer, select a rule set from the **Rule Sets** navigation tab. On the **Overview** tab, and select the Decision Table where you want to check conflicts and click **Edit.**
2. Set the conflict policy to **manual** (this is the default conflict policy). For more information, see Understanding Decision Table Conflict Analysis.
3. In the **Conditions** area, in the conflicts area, when conflicts exist for each rule with a conflict double-click the appropriate condition cell to display the Conflict Resolution dialog.
4. In the Conflict Resolution dialog, for each conflicting rule, in the Resolution field select a resolution from the list.

How to Set the Decision Table Auto Override Conflict Resolution Policy

When you select the Advanced Settings option in a Decision Table, you can select that Decision Table conflicts are automatically resolved using the **auto override** conflict policy (this applies only when it is possible to resolve the conflict using the Oracle Business Rules automatic conflict resolution policies). The automatic override conflict resolution uses a policy where when there is a rule conflict a special case overrides a more general case. For more information, see Understanding Decision Table Conflict Analysis.

To select the auto override policy:

1. Select the rule or Decision Table where you want to use ignore conflict policy.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name.
3. From the **Conflict Policy** option select **auto override**.

How to Set the Decision Table Ignore Conflicts Policy

When you select the Advanced Settings option in a Decision Table, you can select that the Decision Table conflicts are ignored using the **ignore** conflict policy. The ignore policy tells Oracle Business Rules to ignore conflicts in the Decision Table. For more information, see Understanding Decision Table Conflict Analysis.

To select the ignore conflict policy:

1. Select the rule or Decision Table where you want to use the ignore conflicts policy.
2. Click the **Show Advanced Settings** button next to the rule or Decision Table name.
3. From the **Conflict Policy** option select **ignore**.

Creating and Running an Oracle Business Rules Decision Table Application

The Order Approval application demonstrates the integration of a SOA composite application with Oracle Business Rules and the use of a Decision Table.

In this application a process is modeled that uses the decision component to:

- Process rules from XML inputs including: a credit score and the annual spending of a customer, and the total cost of the incoming order.
- Provide output that determines if an order is approved, rejected, or requires manual processing.

To complete this procedure, you need to:

- Obtain the Source Files for the Order Approval Application
- Create an Application for Order Approval
- Create a Business Rule Service Component for Order Approval
- View Data Model Elements for Order Approval

- Add Value Sets to the Data Model for Order Approval
- Associate Value Sets with Order and CreditScore Properties
- Add a Decision Table for Order Approval
  - Split the Cells in the Decision Table and Add Actions
  - Compact the Decision Table
  - Replace Several Specific Rules with One General Rule
  - Add a General Rule
- Check Dictionary Business Rule Validation Log for Order Approval
- Deploy the Order Approval Application
- Test the Order Approval Application

How to Obtain the Source Files for the Order Approval Application

The source code for Oracle Business Rules-specific samples and SOA samples are available online in the Oracle SOA Suite samples page.

To work with the Order Approval application, you first need to obtain the `order.xsd` schema file either from the sample project that you obtain online or you can create the schema file and create all the application, project, and other files in Oracle JDeveloper. You can save the schema file provided in the following example locally to make it available to Oracle JDeveloper.

The following example shows the `order.xsd` schema file.

```
<?xml version="1.0" ?>
<schema attributeFormDefault="qualified" elementFormDefault="qualified"
        targetNamespace="http://example.com/ns/customerorder"
        xmlns:tns="http://example.com/ns/customerorder"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="CustomerOrder">
    <complexType>
      <sequence>
        <element name="name" type="string" />
        <element name="creditScore" type="int" />
        <element name="annualSpending" type="double" />
        <element name="value" type="string" />
        <element name="order" type="double" />
      </sequence>
    </complexType>
  </element>
  <element name="OrderApproval">
    <complexType>
      <sequence>
        <element name="status" type="tns:Status"/>
      </sequence>
    </complexType>
  </element>
  <simpleType name="Status">
    <restriction base="string">
        <enumeration value="manual"/>
        <enumeration value="approved"/>
        <enumeration value="rejected"/>
    </restriction>
  </simpleType>
 </schema>
```

How to Create an Application for Order Approval

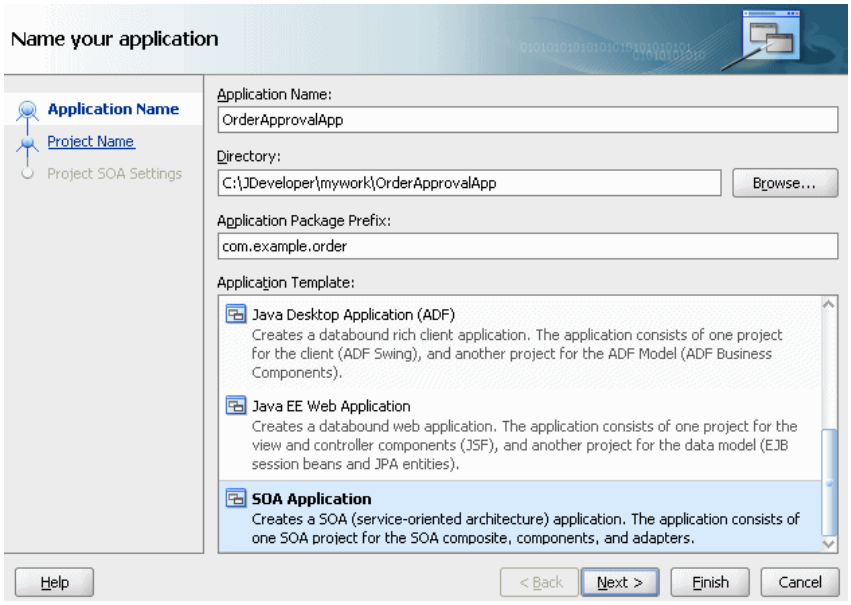
To work with Oracle Business Rules, you first create an application in Oracle JDeveloper.

To create an application for order approval:

1. In the Application Navigator, click **New Application**.
2. In the Name your application dialog, enter the name and location for the new application.
   a. In the **Application Name** field, enter an application name. For example, enter `OrderApprovalApp`.
   b. In the **Directory** field, specify a directory name or accept the default.
   c. In the **Application Package Prefix** field, enter an application package prefix, for example `com.example.order`.

      The prefix, followed by a period, applies to objects created in the initial project of an application.
   d. For a SOA composite with Oracle Business Rules, in the Application Template area select SOA Application for the application template. For example, see Figure 5-20.
   e. Click **Next**.

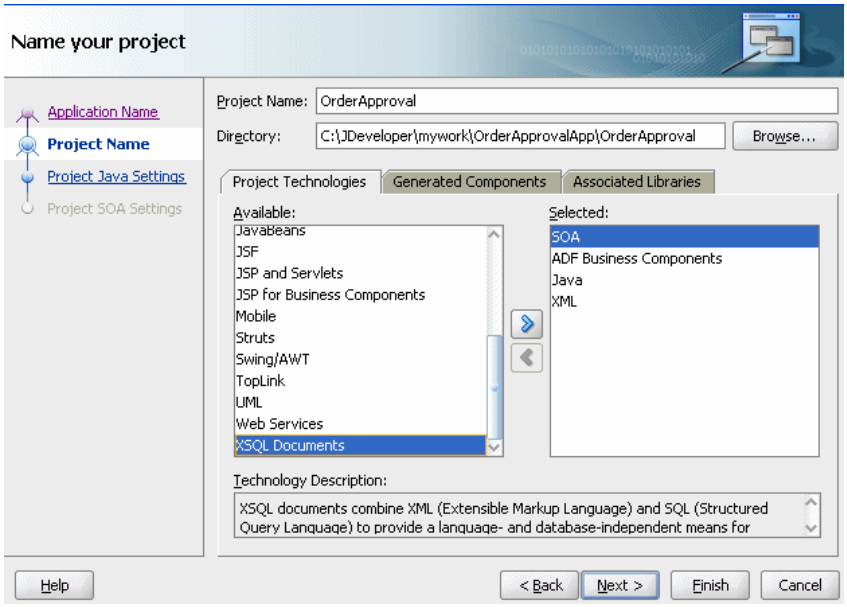
Figure 5-20 Adding the Order Approval Application



3. In the Name your project page enter the name and location for the project.
   a. In the **Project Name** field, enter a name. For example, enter `OrderApproval`.
   b. Enter or browse for a directory name, or accept the default.

c. For an Oracle Business Rules project, in the **Project Technologies** area ensure that SOA, ADF Business Components, Java, and XML are in the **Selected** area on the Project Technologies tab, as shown in Figure 5-21. If an item is missing, select it in the **Available** pane and add it to the **Selected** pane using the **Add** button.

Figure 5-21 Adding a Project to an Application



4. Click **Finish**.

How to Create a Business Rule Service Component for Order Approval

After creating a project in Oracle JDeveloper you need to create a Business Rule Service component within the project. When you add a business rule you can create input and output variables to provide input to the service component and to obtain results from the service component.

To use business rules with Oracle JDeveloper, you do the following:

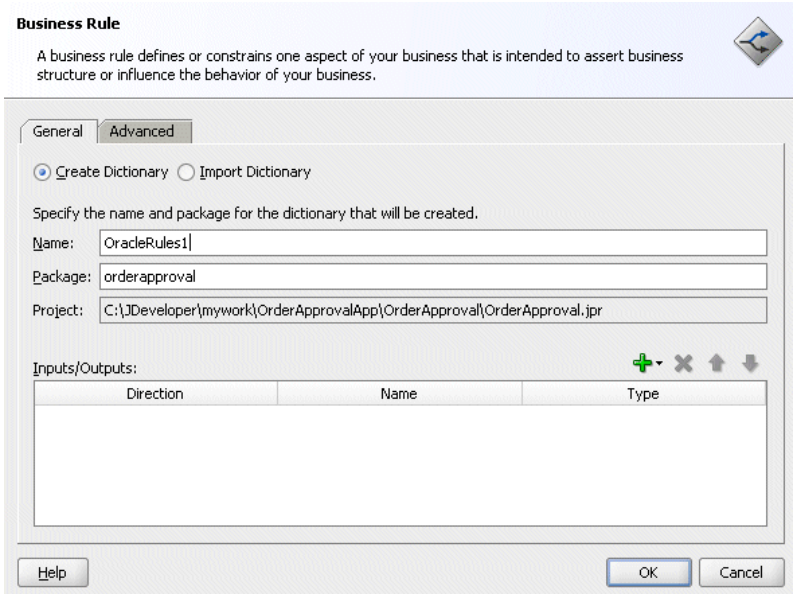
- Add a business rules service component
- Create input and output variables for the service component
- Create an Oracle Business Rules dictionary in the project

To create a business rule service component:

1. In the Application Navigator, in the **OrderApproval** project expand **SOA Content** and double-click `composite.xml` to launch the SOA composite editor (this may already be open after you create the project).
2. From the Component Palette, drag-and-drop a **Business Rule** from the **Service Components** area of the SOA menu to the **Components** lane of the `composite.xml` editor.

   Oracle JDeveloper displays a Create Business Rules page, as shown in Figure 5-22.

Figure 5-22 Adding a Business Rule Dictionary with the Create Business Rules Dialog



3. To add an input, from the list next to the **Add** button select **Input** to enter input for the business rule.

4. In the Type Chooser dialog, click the **Import Schema File**... button. This displays the Import Schema File dialog.

5. In the Import Schema dialog click **Browse Resources** to choose the XML schema elements for the input variable of the process. This displays the SOA Resource Lookup dialog.

6. In the SOA Resource Lookup dialog, navigate to find the `order.xsd` schema file and click **OK**.

7. In the Import Schema File dialog, make sure **Copy to Project** is selected, as shown in Figure 5-23. Click **OK**.

Figure 5-23 Importing the Order.xsd Schema File

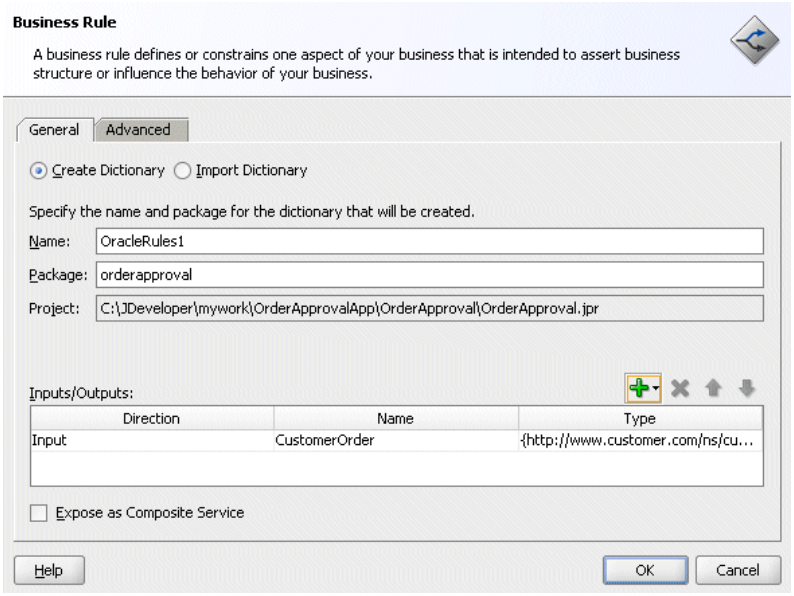
8. If the Localize Files dialog displays, click **OK** to copy the schema to the composite process directory.

9. In the Type Chooser, navigate to the Project Schemas Files folder to select the input variable.

   For this example, select `CustomerOrder` as the input variable.

   On the Type Chooser window, click **OK**. This displays the Create Business Rules dialog, as shown in Figure 5-24.

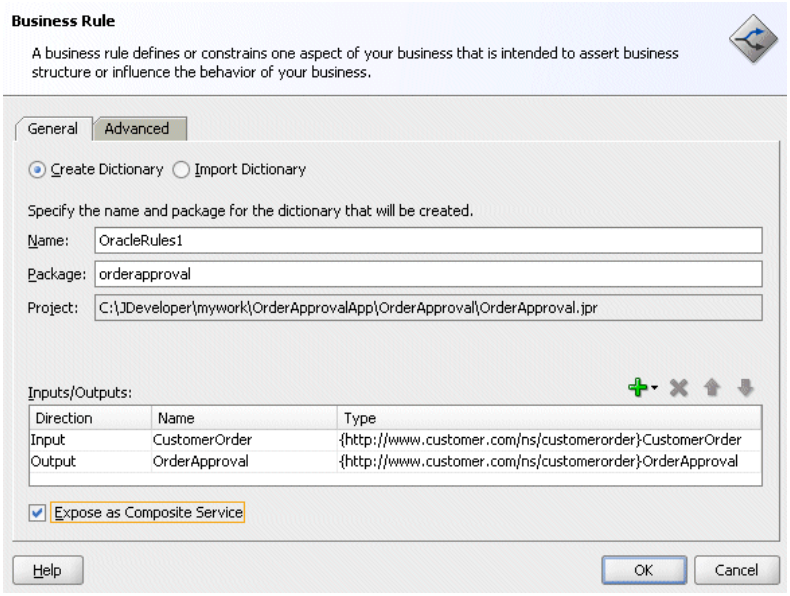
Figure 5-24 Create Business Rules Dialog with CustomerOrder Input



10. In a similar manner, add the output fact type `OrderApproval` from the imported `order.xsd`.

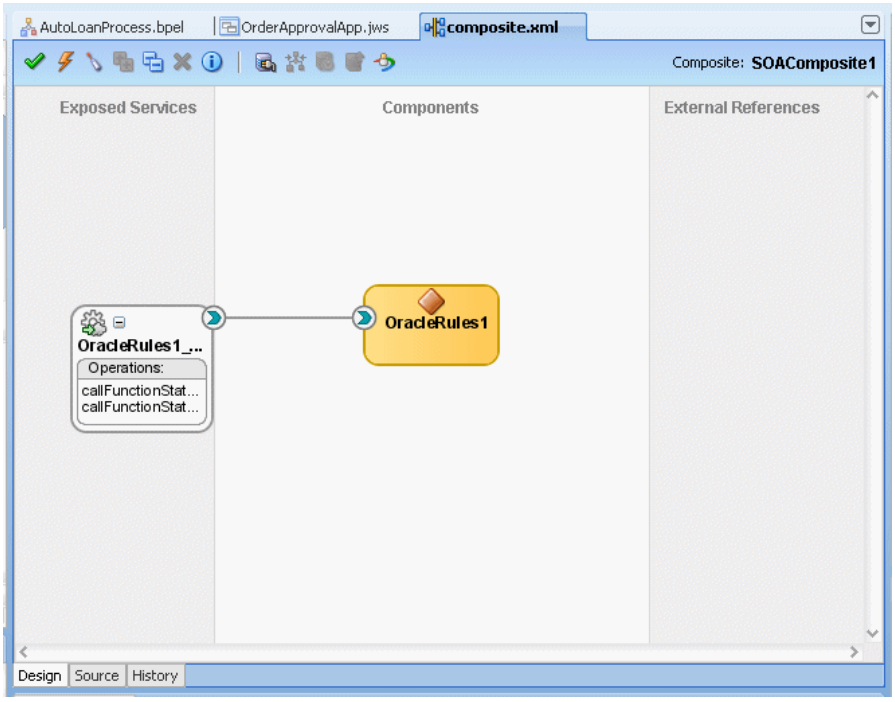
11. In the Create Business Rules dialog, select **Expose as Composite Service**, as shown in Figure 5-25.

Figure 5-25 Create Business Rules Dialog with Input and OrderApproval Output



Click **OK**. This creates the Business Rule component and Oracle JDeveloper shows the Business Rule in the canvas workspace, as shown in Figure 5-26.

Figure 5-26 Business Rules Component in OrderApproval Composite



The business rule service component enables you to integrate your SOA composite application with a business rule. This creates a business rule dictionary and enables you to execute business rules and make business decisions based on the rules.

How to View Data Model Elements for Order Approval

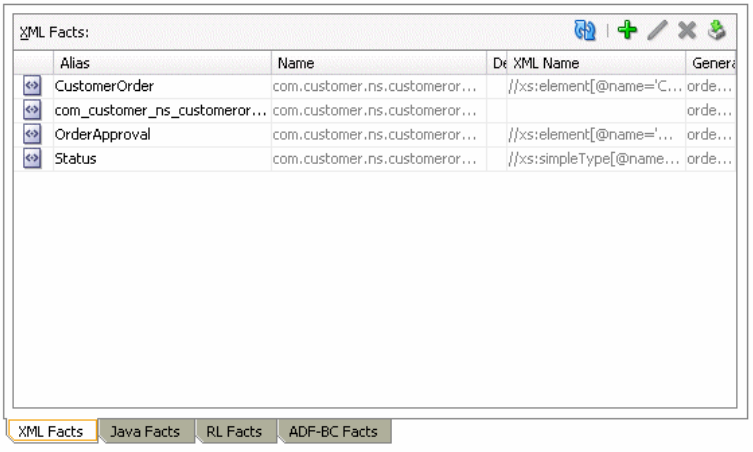
Before adding rules you need to create the Oracle Business Rules data model. The data model contains the business data definitions (types) and definitions for facts that you use to create rules. For example, for this sample the data model includes the XML schema elements from `order.xsd` that you specify when you define inputs and outputs for the business rule activity.

At times when you work with Rules Designer to create a rule or a Decision Table, you may need to create or modify elements in the data model.

To view data model elements for Oracle business rules:

1. Select the composite tab with the value **composite.xml**, and in the Components lane select the business rule (this surrounds the component, **OracleRules1** with a dashed selection box).
2. Double-click the selection box to launch Rules Designer.
3. In Rules Designer select the **Facts** navigation tab.
4. Select **XML Facts** tab in the **Facts** navigation tab as shown in Figure 5-27.

Figure 5-27 Opening a Business Rules Dictionary with Rules Designer



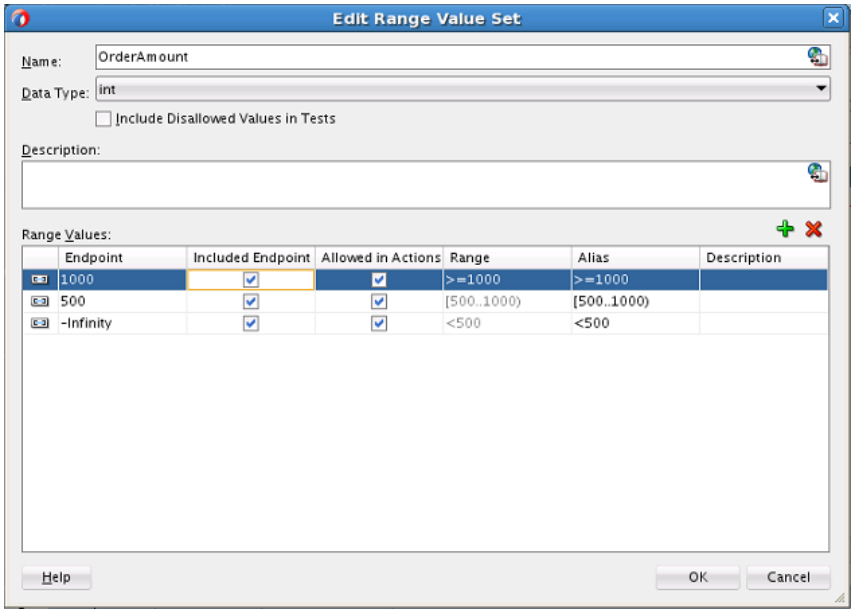
How to Add Value Sets to the Data Model for Order Approval

To use a Decision Table you need to define value sets that specify how to draw values for each cell for the conditions that constitute the Decision Table. For this example the value sets are defined with a list of ranges that you define in Rules Designer.

To add OrderAmount value set to the data model:

1. In Rules Designer, select the **Value Sets** navigation tab.
2. From the drop down next to the **Create Value Set...** button, select **Range Value Set**.
3. In the **Name** field, enter `OrderAmount`. Press **Enter** to accept the name.
4. Double-click the **OrderAmount** value set icon to display the **Edit Range Value Set** dialog.
5. Click **Add Value** to add a value.
6. Click **Add Value** again to add another value.
7. In the **Range Values** area, in the top **Endpoint** field enter 1000 for the endpoint value.
8. In the **Range Values** area, for the middle bucket in the **Endpoint** field enter 500 for the endpoint value.
9. In the **Included Endpoint** field for each value set ensure the check box is selected, as shown in Figure 5-28.

Figure 5-28 Adding the OrderAmount Value Set



10. Modify the **Alias** field for each value to **High, Medium, and Low**. Click **OK**.

How to Add CreditScore Value Set to the Data Model

To add CreditScore value set:

1. In Rules Designer select the **Value Sets** navigation tab.
2. From the drop down next to the **Create Valueset...** button, select **List of Ranges**.
3. In the **Name** field, enter `CreditScore`.
4. Double-click the **CreditScore** valueset icon to display the Edit Valueset dialog.
5. Click **Add Value** to add a value.
6. Click **Add Value** again to add another value.
7. In the top valueset, in the **Endpoint** field enter `750`.
8. For the middle valueset, in the **Endpoint** field enter `400`.
9. In the **Included Endpoint** field for each valueset, ensure the check box is selected.
10. Modify the **Alias** field for each endpoint value to **solid** for 750, **avg** for 400, and **risky** for -Infinity. Click **OK**.

How to Associate Value Sets with Order and CreditScore Properties

To prepare for creating Decision Tables you can associate a value set with fact properties in the data model. In this way condition cells in a Decision Table **Conditions** area can use the valuesets when you create a Decision Table.

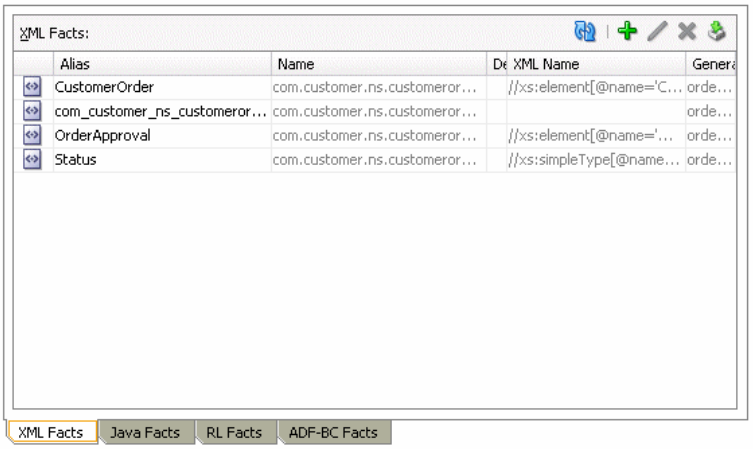
Note that the `OrderApproval.status` property is associated with the `Status` value set when the `OrderApproval` fact type is imported from the XML schema. In the schema, `Status` is a restricted `String` type and is therefore represented as an

enum valueset. Rules Designer creates the status value set. For more information, see What You Need to Know About XML Facts.

To associate value sets with Order and CreditScore properties:

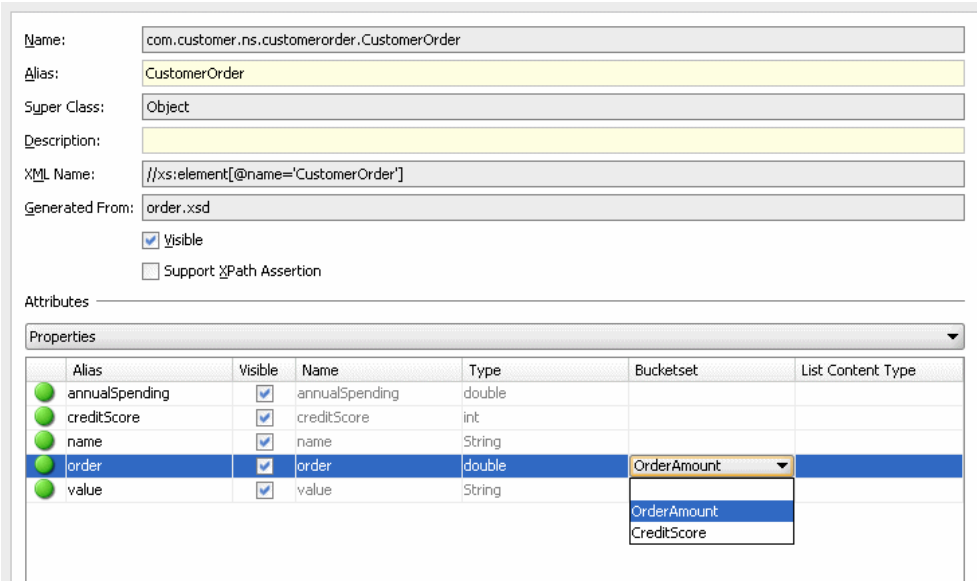
1. In Rules Designer select the **Facts** navigation tab.
2. Select the **XML Facts** tab in the **Facts** navigation tab as shown in Figure 5-29.

Figure 5-29 Opening a Business Rules Dictionary with Rules Designer



3. Select the type you want to modify. For example in the XML Facts table double-click the icon next to the **CustomerOrder** entry. This displays the Edit XML Fact dialog.
4. In the Edit XML Fact dialog, in the **Properties** table in the **Value Set** column select the cell for the appropriate property and from the list select the valueset you want to use. For example, for the property **order** select the **OrderAmount** valueset, as shown in Figure 5-30.

Figure 5-30 Associating the OrderAmount Valueset with CustomerOrder.order



5. In a similar manner, for the property **creditScore** select the **CreditScore** valueset.
6. Click **OK**.

How to Add a Decision Table for Order Approval

You create a Decision Table to process input facts and to produce output facts, or to produce intermediate conclusions that Oracle Business Rules can further process using additional rules or in another Decision Table.

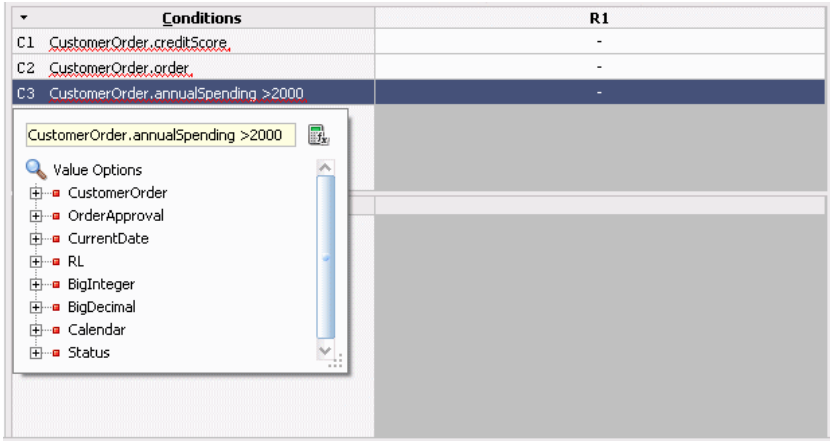
While you work with rules you can use the rule validation features in Rules Designer to assist you. Rules Designer performs dictionary validation when you make any change to the dictionary. To show the validation log window, click the **Validate** button or select **View**>**Log** and select the **Business Rule Validation** tab. If you view the rules validation log you should see warning messages. You remove these warning messages as you create the Decision Table. For more information on rule validation see Understanding Rule Validation.

To use a Decision Table for rules in this sample application you work with facts representing a customer spending level and a customer credit risk for a particular customer and a particular order. Then, you use a Decision Table to create rules based on customer spending, the order amount, and the credit risk of the customer.

To add a Decision Table for order approval:

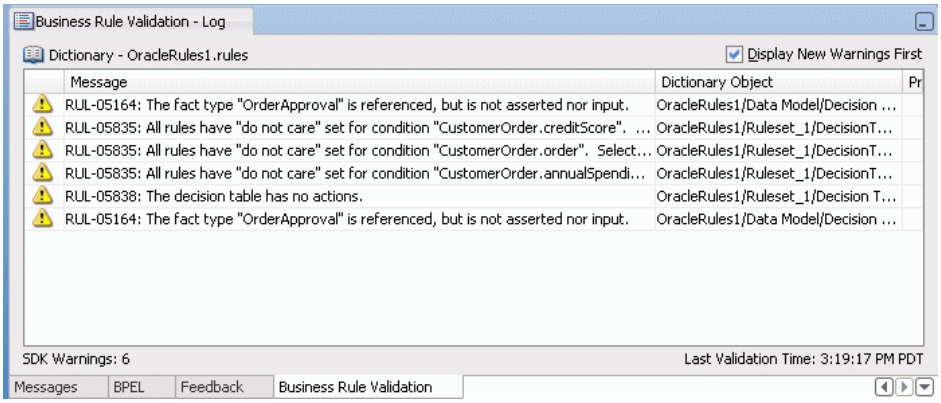
1. In Rules Designer, select **Ruleset_1** under the **Rulesets** navigation tab.
2. Click the **Add** button and from the list and select **Create Decision Table**.
3. In the Decision Table, click the **Add** button and from the list select **Condition**.
4. In the Decision Table, double-click **<edit condition>**. Then, in the navigator expand **CustomerOrder** and select **creditScore**. This enters the expression `CustomerOrder.creditScore` in the **Conditions** column.
5. Again, in the Decision Table, click the **Add** button and from the list select **Condition**.
6. In the Decision Table, in the **Conditions** area double-click the **<edit condition>**. Then, in the navigator expand **CustomerOrder** and select **order**. This enters the expression `CustomerOrder.order`.
7. Again, in the Decision Table, click the **Add** button and from the list select **Condition**.
8. In the Decision Table, double-click **<edit condition>**.
9. In the navigator expand **CustomerOrder** and select **annualSpending**. In the text entry area, add `>2000` as shown in Figure 5-31.

Figure 5-31 Adding the Annual Spending Entry to a Decision Table



10. Type **Enter** to accept the value, as shown in Figure 5-32. If you view the rules validation log you should see the warning messages as shown in Figure 5-32. You remove these warning messages as you modify the Decision Table in later steps.

Figure 5-32 Adding Conditions to the CustomerOrder Decision Table

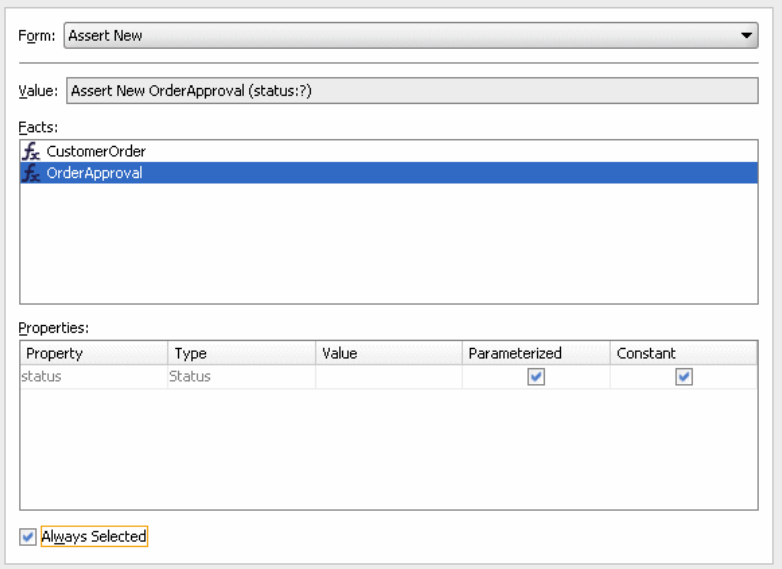


How to Create an action in a Decision Table

To create an action in a Decision Table:

1. In the Decision Table click the **Add** button and from the list select **Action** > **Assert New**.
2. In the **Actions** area, double-click **assert new(**. This displays the Action Editor dialog.
3. In the Action Editor dialog, in the **Facts** area select **OrderApproval**.
4. In the Action Editor dialog, in the Properties table for the property `status` select the **Parameterized** check box and the **Constant** check box. This specifies that each rule independently sets the status.
5. In the Action Editor dialog, select the **Always Selected** check box as shown in Figure 5-33.

Figure 5-33 Adding an Action to a Decision Table with the Action Editor Dialog



6. In the Action Editor dialog, click **OK**.

Next you need to add rules to the Decision Table and specify an action for each rule.

Split the Cells in the Decision Table and Add Actions

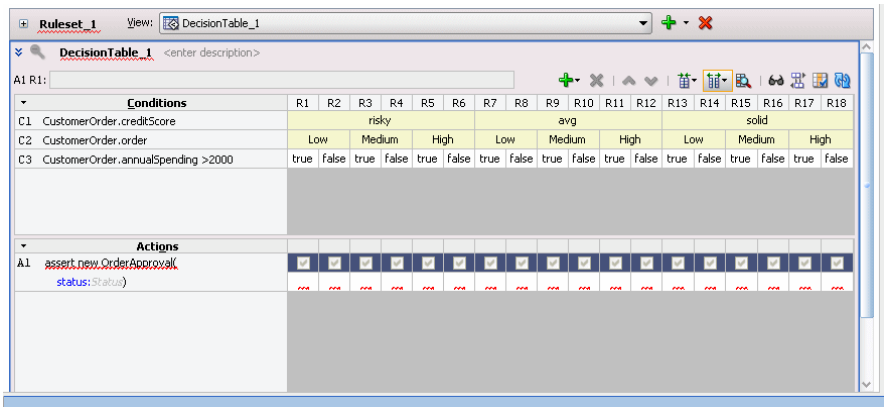
You can use the Decision Table split operation to create rules for the valuesets associated with the condition rows in the Decision Table. This creates one rule for every combination of condition valuesets. There are three order amount valuesets, three credit score valuesets, and two boolean valuesets for the annual spending amount for a total of eighteen rules (3 x 3 x 2 = 18).

To split cells in a decision table:

1. Select the Decision Table.
2. In the Decision Table, click the **Split Table** button and from the list select **Split Table**. The split table operation eliminates the "do not care" cells from the table. The table now shows eighteen rules that cover all ranges as shown in Figure 5-34.

These steps produce validation warnings for action cells with missing expressions. You fix these in later steps.

Figure 5-34 Splitting a Decision Table Using Split Table Operation



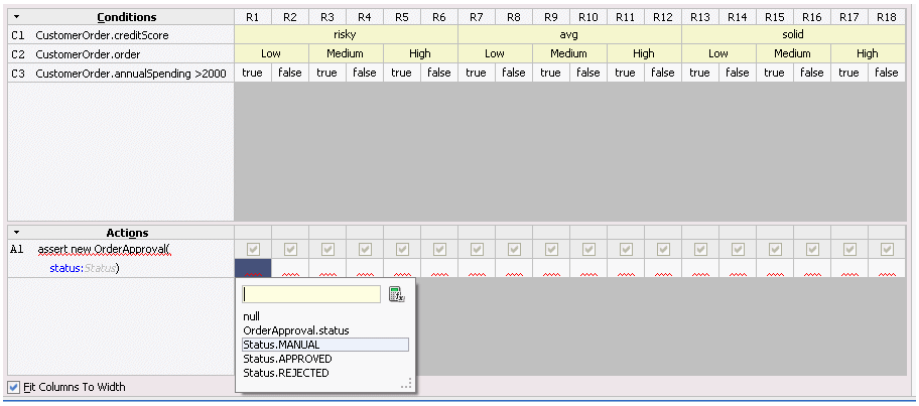
How to Add Actions for Each Rule in the Decision Table

In the Decision Table you specify a value for the status property associated with OrderApproval for each action cell in the **Actions** area. The possible choices are: `Status.MANUAL`, `Status.REJECTED`, or `Status.ACCEPTED`. In this step you fill in a value for status for each of the 18 rules. The values you enter correspond to the conditions that form each rule in the Decision Table.

To add actions for each rule in the decision table:

1. In the **Actions** area, double-click the action cell for the rule you want to work with, as shown in Figure 5-35.

Figure 5-35 Adding Action Cell Values to a Decision Table



2. In the list, select and enter a value for the action cell. For example, enter `Status.MANUAL`.

3. For each action cell, enter the appropriate value as determined by the logic of your application. For this sample application use the values for the Decision Table actions as shown in Table 5-5.

4. Select **Save All** from the **File** main menu to save your work.

Table 5-5 Values for Decision Table Actions

Rule	C1 creditScore	C2 order	C3 annualSpending > 2000	A1 OrderApproval status
R1	risky	Low	true	`Status.MANUAL`
R2	risky	Low	false	`Status.MANUAL`
R3	risky	Medium	true	`Status.MANUAL`
R4	risky	Medium	false	`Status.REJECTED`
R5	risky	High	true	`Status.MANUAL`
R6	risky	High	false	`Status.REJECTED`
R7	avg	Low	true	`Status.APPROVED`
R8	avg	Low	false	`Status.MANUAL`
R9	avg	Medium	true	`Status.APPROVED`
R10	avg	Medium	false	`Status.MANUAL`
R11	avg	High	true	`Status.MANUAL`
R12	avg	High	false	`Status.MANUAL`
R13	solid	Low	true	`Status.APPROVED`
R14	solid	Low	false	`Status.APPROVED`
R15	solid	Medium	true	`Status.APPROVED`
R16	solid	Medium	false	`Status.APPROVED`
R17	solid	High	true	`Status.APPROVED`
R18	solid	High	false	`Status.MANUAL`

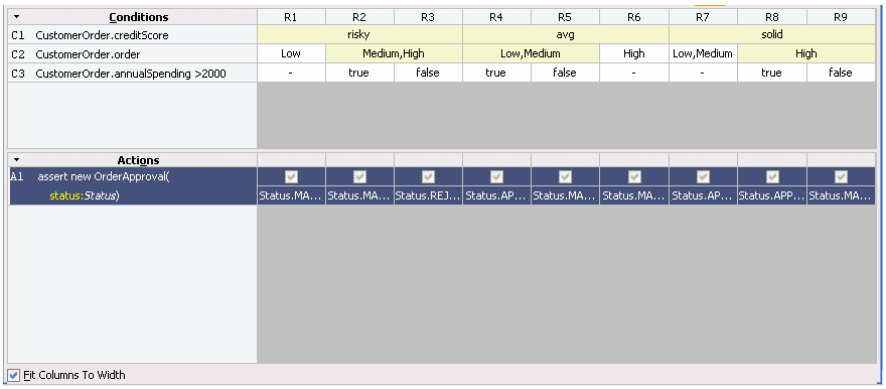
Compact the Decision Table

In this step you compact the rules to merge from eighteen rules to nine rules. This automatically eliminates the rules that are not needed and preserves the no gap, no conflict properties for the Decision Table.

To compact the decision table:

1. Select the Decision Table.
2. Click the **Resize All Columns to Same Width** button.
3. Click the **Compact Table** button and from the list select **Compact Table**. The compact table operation eliminates rules from the Decision Table. The Decision Table now shows nine rules, as shown in Figure 5-36.

Figure 5-36 Compacting a Decision Table Using Compact Table



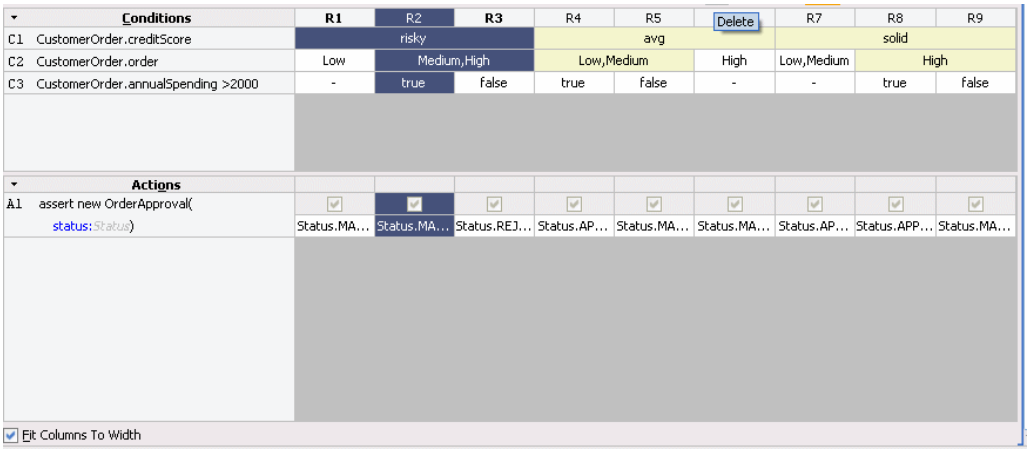
Replace Several Specific Rules with One General Rule

Notice that five of the nine remaining rules result in a manual order approval status. You can reduce the number of rules by deleting these five rules. Note it is often best practice to not do this (that is not replace several specific rules with one general rule). You need to compare the benefits of having fewer rules with the added complexity of managing the conflicts introduced when you reduce the number of rules.

To replace several specific rules with one general rule:

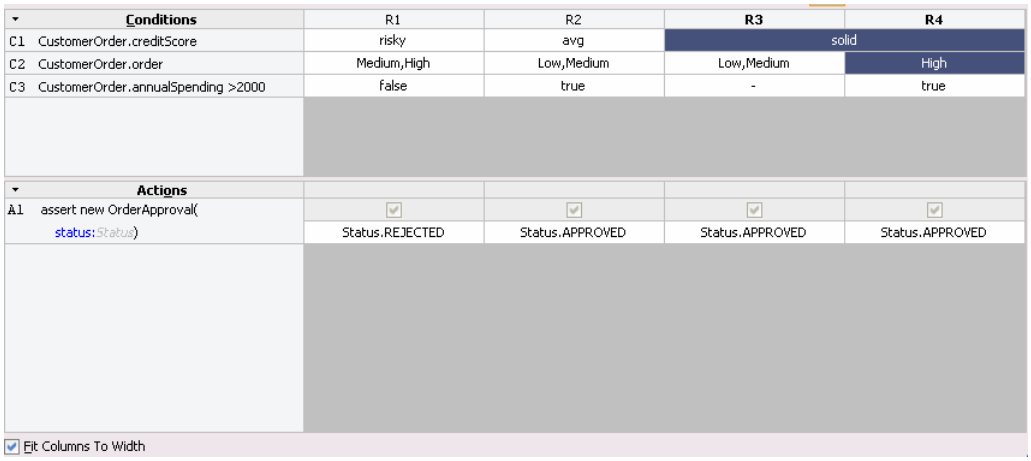
1. Select the Decision Table.
2. In the Decision Table, select a rule with OrderApproval status action set to `Status.MANUAL`. To select a rule, click the column heading. For example, click rule **R2** as shown in Figure 5-37.
3. Click **Delete** to remove a rule in the Decision Table. Be careful to click the delete button in the Decision Table area to delete a rule in the decision table (there is also a delete button shown in the **Ruleset** area that deletes the complete Decision Table).

Figure 5-37 Deleting Rules from a Decision Table



4. Repeat these steps to delete all the rules with action set to `Status.MANUAL`. This should leave the Decision Table with four rules as shown in Figure 5-38.

Figure 5-38 Decision Table After Manual Actions Removed



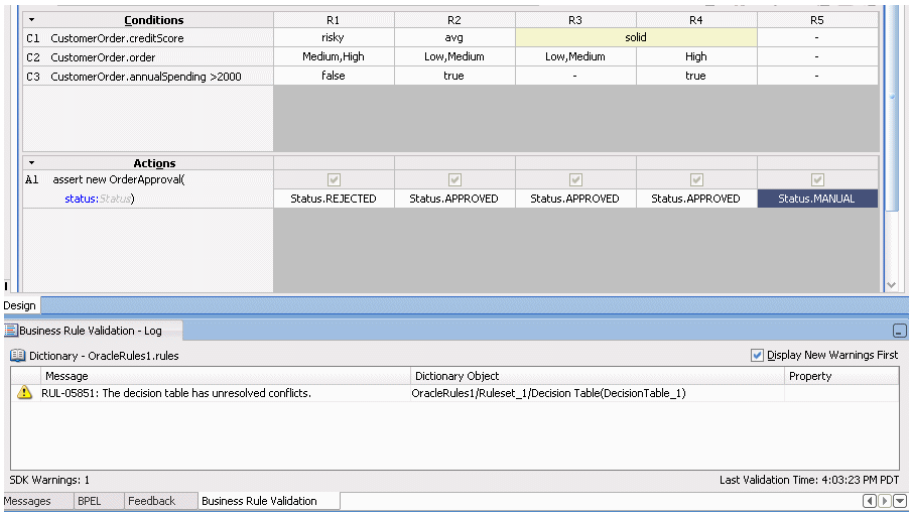
Add a General Rule

Now you can add a single rule to handle the manual case. After adding this rule you set the conflict policy with the option **Conflict Policy auto override** for conflict resolution.

To add a general rule:

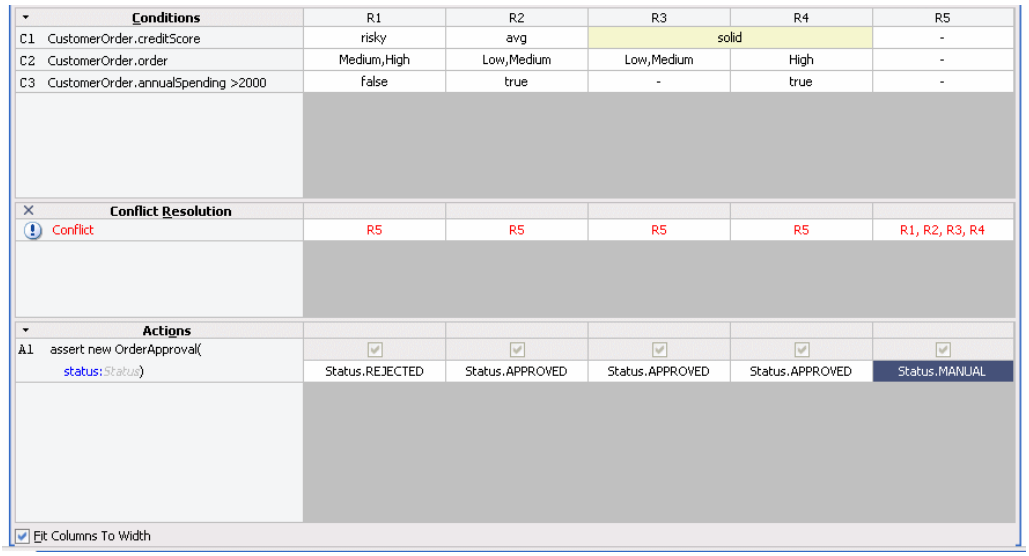
1. In the Decision Table, click the **Add** button and from the list select **Rule**.
2. In the **Conditions** area, for the three conditions leave the "-" do not care value for each cell in the rule.
3. In the **Actions** area, enter `Status.MANUAL`, as shown in Figure 5-39. Notice that the **Business Rule Validation** log includes the warning `RUL-05851` for unresolved conflicts.

Figure 5-39 Decision Table with Conflicting Rules



4. Show the conflicting rules by clicking the **Toggle Display of Conflict Resolution** button, as shown in Figure 5-40.

Figure 5-40 Adding a Rule to Handle Status Manual



Conditions	R1	R2	R3	R4	R5
C1 CustomerOrder.creditScore	risky	avg	solid		-
C2 CustomerOrder.order	Medium,High	Low,Medium	Low,Medium	High	-
C3 CustomerOrder.annualSpending >2000	false	true	-	true	-
Conflict Resolution					
Conflict	R5	R5	R5	R5	R1, R2, R3, R4
Actions					
A1 assert new OrderApproval(	☑	☑	☑	☑	☑
status:Status)	Status.REJECTED	Status.APPROVED	Status.APPROVED	Status.APPROVED	Status.MANUAL

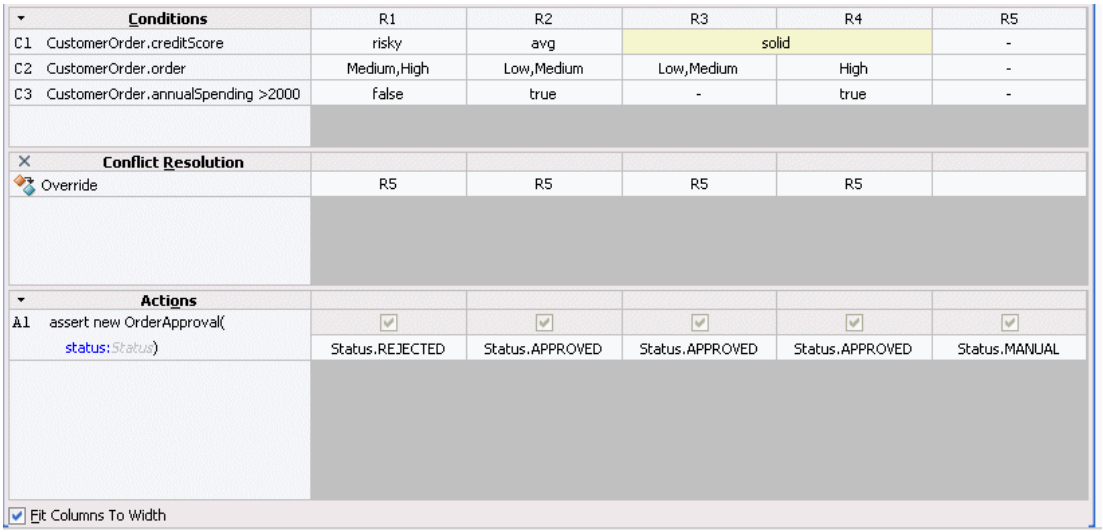
Fit Columns To Width

How to Enable the Auto Override Conflict Resolution Policy

To enable the auto override conflict resolution policy:

1. In the Decision Table click **Show Advanced Settings** (next to the Decision Table name).
2. In the Conflict Policy list, select **auto override**. After adding the manual case rule and selecting **auto override**, notice that the conflicts are resolved and special cases override the general case, as shown in Figure 5-41.

Figure 5-41 Adding a Rule to Handle Status Manual with Auto Override Conflict Policy



Conditions	R1	R2	R3	R4	R5
C1 CustomerOrder.creditScore	risky	avg	solid		-
C2 CustomerOrder.order	Medium,High	Low,Medium	Low,Medium	High	-
C3 CustomerOrder.annualSpending >2000	false	true	-	true	-
Conflict Resolution					
Override	R5	R5	R5	R5	
Actions					
A1 assert new OrderApproval(	☑	☑	☑	☑	☑
status:Status)	Status.REJECTED	Status.APPROVED	Status.APPROVED	Status.APPROVED	Status.MANUAL

Fit Columns To Width

How to Check the Business Rule Validation Log for Order Approval

Before you can deploy the application you need to make sure the dictionary validates without warnings. If there are any validation warnings, you need to fix any associated problems. To validate the dictionary, in the Business Rule Validation Log, check for

any validation warnings. If there are warnings, perform appropriate actions to correct the problems.

How to Deploy the Order Approval Application

Business rules created in a SOA application are deployed as part of the SOA composite when you create a deployment profile in Oracle JDeveloper. You deploy a SOA composite application to Oracle WebLogic Server.

To deploy and run the order approval application:

1. If you have not started your application server instance, then start the Oracle WebLogic Server.
2. In the Application Navigator, right-click the **OrderApproval** project and select **Deploy** > **OrderApproval** > to the appropriate server name.

   Then the SOA Deployment Configuration dialog displays. Select your Application connection which you either have created already or you can create it now. The connection contains the authorization and other connection information (server name, port, etc).
3. Click **Next**.
4. In Select Server select or create and then select your application connection.
5. Click **Next, Next** and **Finish.**

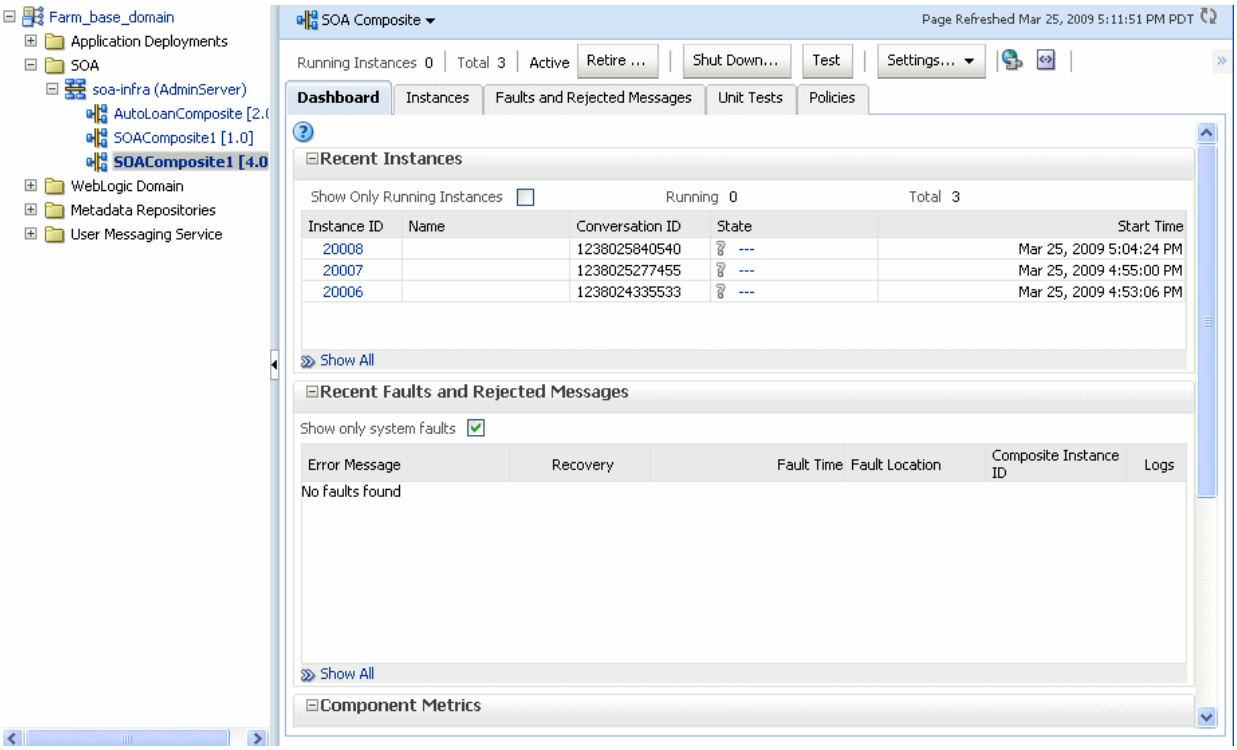
How to Test the Order Approval Application

After deploying the application you can test the Decision Table in the SOA composite application with the Oracle Enterprise Manager Fusion Middleware Control Console.

To test the application:

1. Open the composite application in Oracle Enterprise Manager Fusion Middleware Control Console, as shown in Figure 5-42.

Figure 5-42 Testing the Order Approval Application



2. Click **Test**.

3. In the **Input Arguments** area, select **XML View**. Replace the XML with the contents of the sample input for testing Order Approval application example as shown below.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body xmlns:ns1="http://xmlns.oracle.com/OracleRules1/
OracleRules1_DecisionService_1">
        <ns1:callFunctionStateless name="OracleRules1_DecisionService_1">
                  <ns1:parameterList xmlns:ns3="http://example.com/ns/
customerorder">
                           <ns3:CustomerOrder>
                                  <ns3:name>Gary</ns3:name>
                                      <ns3:creditScore>600</ns3:creditScore>
                                      <ns3:annualSpending>2001.0</
ns3:annualSpending>
                                      <ns3:value>High</ns3:value>
                                      <ns3:order>100.0</ns3:order>
                    </ns3:CustomerOrder>
                  </ns1:parameterList>
            </ns1:callFunctionStateless>
       </soap:Body>
</soap:Envelope>
```

4. Replace the values in the input shown in step 3 as desired for your test.

5. Click **Test Web Service**.

6. In the **Response** tab, view the results. For example, for this input:

```
/OracleRules1_DecisionService_1" xmlns:ns2="http://xmlns.oracle.com/bpel">
      <resultList>
```

```
          <OrderApproval:OrderApproval      xmlns:OrderApproval="http://
example.com/ns/customerorder"
xmlns="http://example.com/ns/customerorder">
               <status>approved</status>
          </OrderApproval:OrderApproval>
       </resultList>
  </callFunctionStatefulDecision>
```

Editing Decision Tables in Microsoft Excel

Business users may find that editing Decision Tables is easier to do in Microsoft Excel. New functionality enables both developers and business users to export and edit Decision Tables in Excel and then import the Decision Tables back into the dictionary.

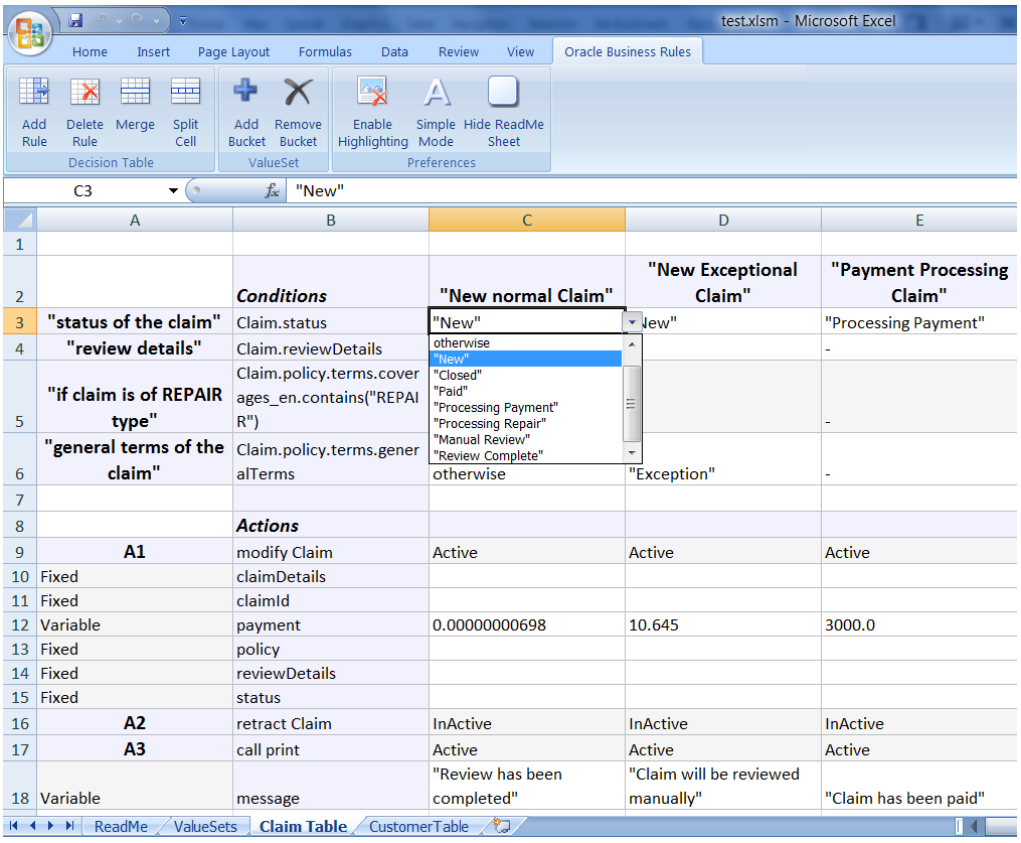
You can export and edit Decision Tables at design-time in Oracle JDeveloper or Business Process Composer. At runtime, you can export and edit in SOA Composer. You can export one or more Decision Tables from a Rule dictionary to the same Excel workbook.

When you import back into the dictionary, you can create a new dictionary, overwrite the existing dictionary, or perform a Diff-Merge. The Diff-Merge enables you to compare dictionaries and accept (merge) or reject any differences.

For more information about comparing dictionaries, see How to Compare or Merge Two or More Dictionaries.

The Excel workbook structure consists of several worksheets: a Readme sheet, a Value Set sheet, and one sheet for each exported Decision Table, as shown in Figure 5-43. Only Rules and Value Sets can be edited in Excel. You can export to .xlsm (default) or .xls.

Figure 5-43 Microsoft Excel Workbook



	A	B	C	D	E
1					
2		***Conditions***	**"New normal Claim"**	**"New Exceptional Claim"**	**"Payment Processing Claim"**
3	**"status of the claim"**	Claim.status	"New"	"New"	"Processing Payment"
4	**"review details"**	Claim.reviewDetails			-
5	**"if claim is of REPAIR type"**	Claim.policy.terms.coverages_en.contains("REPAIR")			-
6	**"general terms of the claim"**	Claim.policy.terms.generalTerms	otherwise	"Exception"	-
7					
8		***Actions***			
9	**A1**	modify Claim	Active	Active	Active
10	Fixed	claimDetails			
11	Fixed	claimId			
12	Variable	payment	0.00000000698	10.645	3000.0
13	Fixed	policy			
14	Fixed	reviewDetails			
15	Fixed	status			
16	**A2**	retract Claim	InActive	InActive	InActive
17	**A3**	call print	Active	Active	Active
18	Variable	message	"Review has been completed"	"Claim will be reviewed manually"	"Claim has been paid"

When you open the Excel workbook, the macros are disabled by default. If you enable the macros, a new tab called Oracle Business Rules, appears. This tab enables you to add or delete rules, merge or split cells, and add or remove values from value sets. You can also disable or enable highlighting, use a simple or advanced mode and hide or show the Readme worksheet.

You can edit with the macros disabled, though you will not be able to:

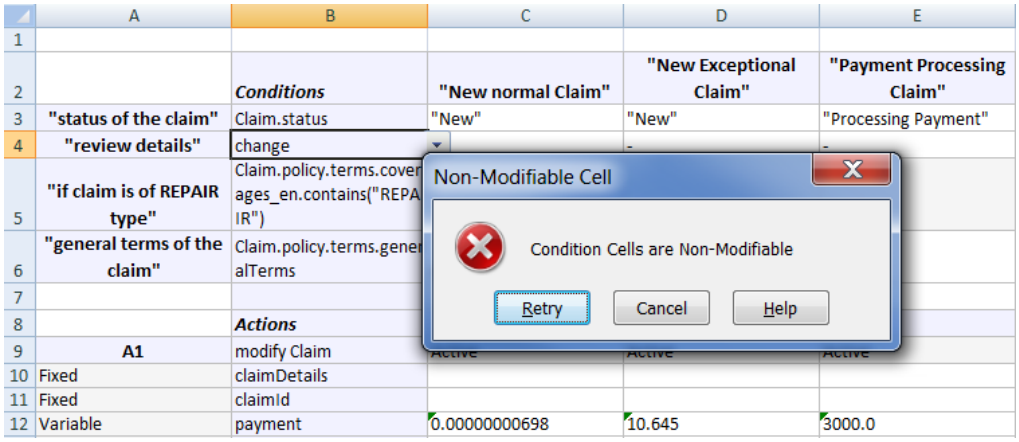
- Choose values from drop lists for restricted cells.
- Edit free form cells.
- Copy and paste a range of cells to add a rule or Value Set.
- Delete a range of cells to delete a rule or Value Set.
- Split or merge cells.
- Create Value Sets automatically.
- Validate the structure of Decision Tables or Value Sets.

Using the predefined macros, you can:

- Add and delete rules.
- Split or merge cells.
- Add or delete Value Sets.
- Editable cells include:
  - Description for Rules, Conditions, Actions.
  - Condition and Action nodes.
  - Action state.
  - Parameterized options for Action parameters.
- Non-editable cells include:
  - Condition expressions.
  - Action expressions.
  - Action parameters.

  If you try to edit these cells, you will get an error message, as shown in Figure 5-44.

Figure 5-44 Non-Modifiable Cell



	A	B	C	D	E
1					
2		***Conditions***	**"New normal Claim"**	**"New Exceptional Claim"**	**"Payment Processing Claim"**
3	**"status of the claim"**	Claim.status	"New"	"New"	"Processing Payment"
4	**"review details"**	change	-	-	-
5	**"if claim is of REPAIR type"**	Claim.policy.terms.coverages_en.contains("REPAIR")			
6	**"general terms of the claim"**	Claim.policy.terms.generalTerms			
7					
8		***Actions***			
9	**A1**	modify Claim			
10	Fixed	claimDetails			
11	Fixed	claimId			
12	Variable	payment	0.00000000698	10.645	3000.0

Understanding What is Exported

In the SDK, there are shared Value Sets that can be associated with multiple conditions across Decision Tables. However, in Excel there are no shared Value Sets--each condition has its own Value Set--so you can only export a Value Set if it is modifiable in Excel. The Value Sets that are non-modifiable include:

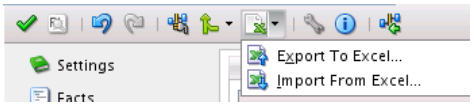
- Linked Dictionary Value Sets.
- Enums.
- Internal Value Sets, for example, boolean Value Sets.

In the worksheet, you can only select values from the drop down for the conditions associated with non-modifiable Value Sets. A highlighting mechanism informs you which conditions are associated with non-modifiable Value Sets.

How to Export Decision Tables

The export and import functionality is invoked using the toolbar button, as shown in Figure 5-45.

Figure 5-45 Export and Import Toolbar Button



To export Decision Table:

1. In Rules Designer, click **Export to Excel**.
2. In the **Export to Excel** dialog box, select the **Format** and browse to the folder where you want to save the workbook.
3. Click **Add** and select the Decision Table(s) to export and click **OK.**
4. Check the **Read Only Value Set** check box to make all of the value sets read-only in Excel. There will not be any Value Sets sheet in the Excel workbook. All conditions will have drop down menus from which values can be selected but no values can be added or removed.

5. Click **Export.** You can now open the workbook and edit the Decision Table.

How to Import Edited Decision Tables Back to the Dictionary

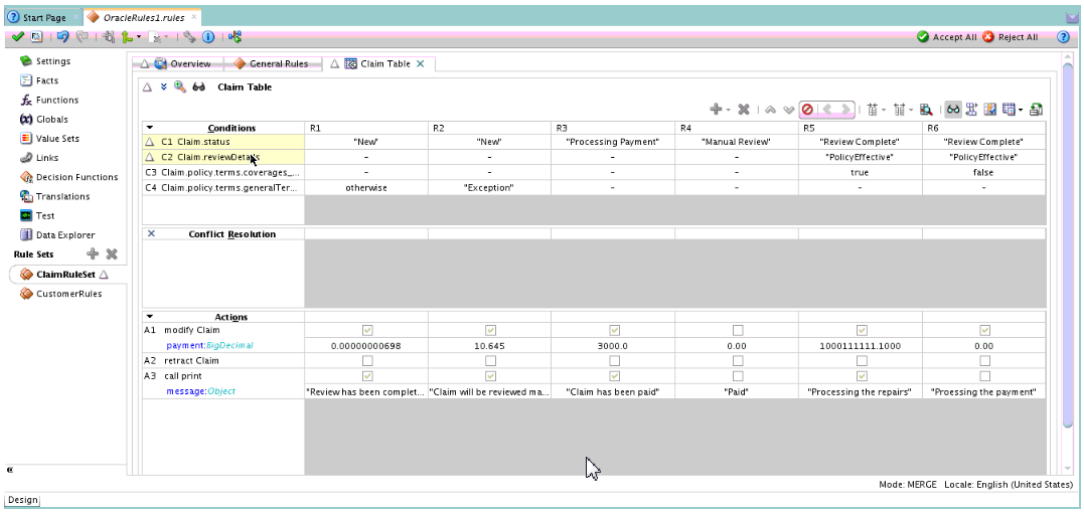
The export and import functionality is invoked using the toolbar button, as shown in Figure 5-45. You can only import Excel workbooks that have been previously exported.

To import edited Decision tables:

1. In Rules Designer, click **Import from Excel**.
2. In the **Import from Excel** dialog box, select the **File** to browse to the folder where you saved the workbook.
3. The **Perform Diff-Merge on Import** check box is selected by default. Browse to the **Base Dictionary** that you want to compare your file to. The base dictionary is required for a 3 way diff-merge.
4. Clear the **Perform Diff-Merge on Import** check box and select **Create New** or **Overwrite.**
5. Click **Import.** The decision table is imported into Rules Designer, where you can accept or reject changes, as shown in Figure 5-46. Each changed artifact is flagged with a change icon. Merging dictionaries should be done with caution.

   For more information about using the Diff-Merge, see How to Compare or Merge Two or More Dictionaries.

Figure 5-46 Perform Diff-Merge on Import



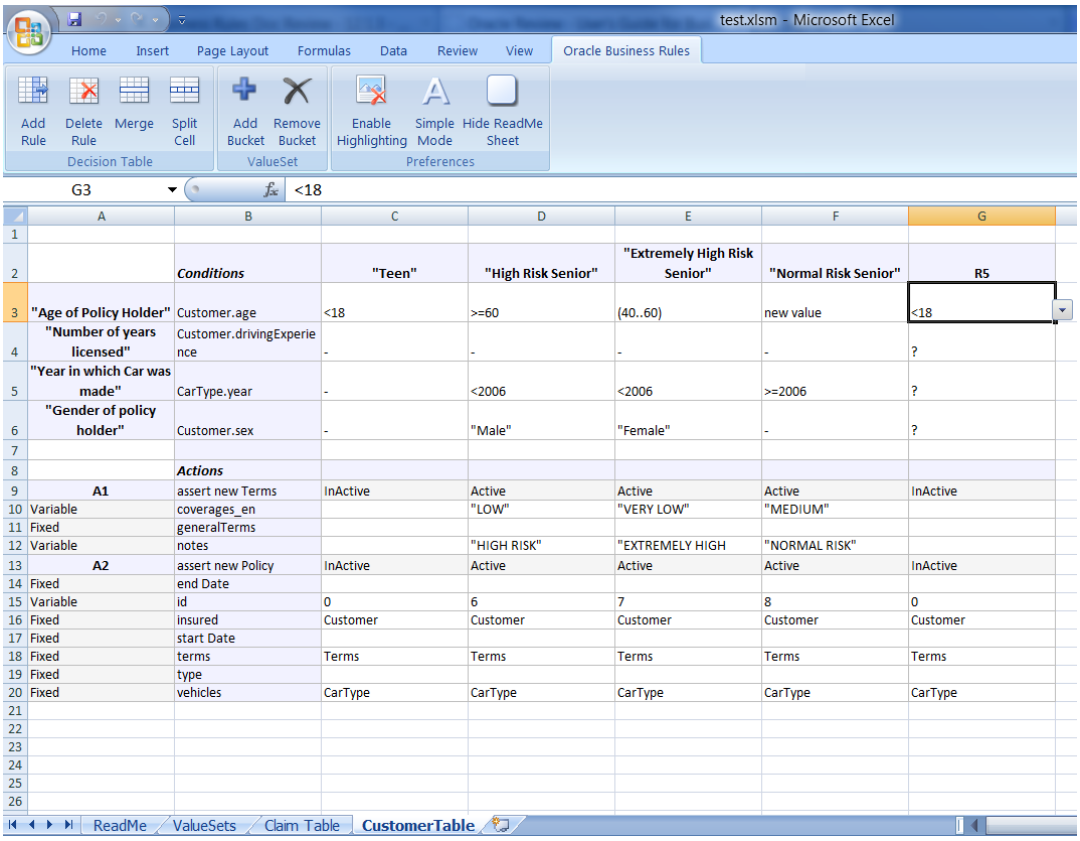
How to Edit Decision Tables in Excel

In Excel, enable the macros to view the Oracle Business Rules tab, which provides you with options to author rules, edit Value Sets, and set preferences.

Adding or Deleting Rules and Merging or Splitting Cells

For each Decision Table worksheet, you can add a rule, as shown in Figure 5-47, delete rules, and merge or split cells.

Figure 5-47 Oracle Business Rules tab in Excel

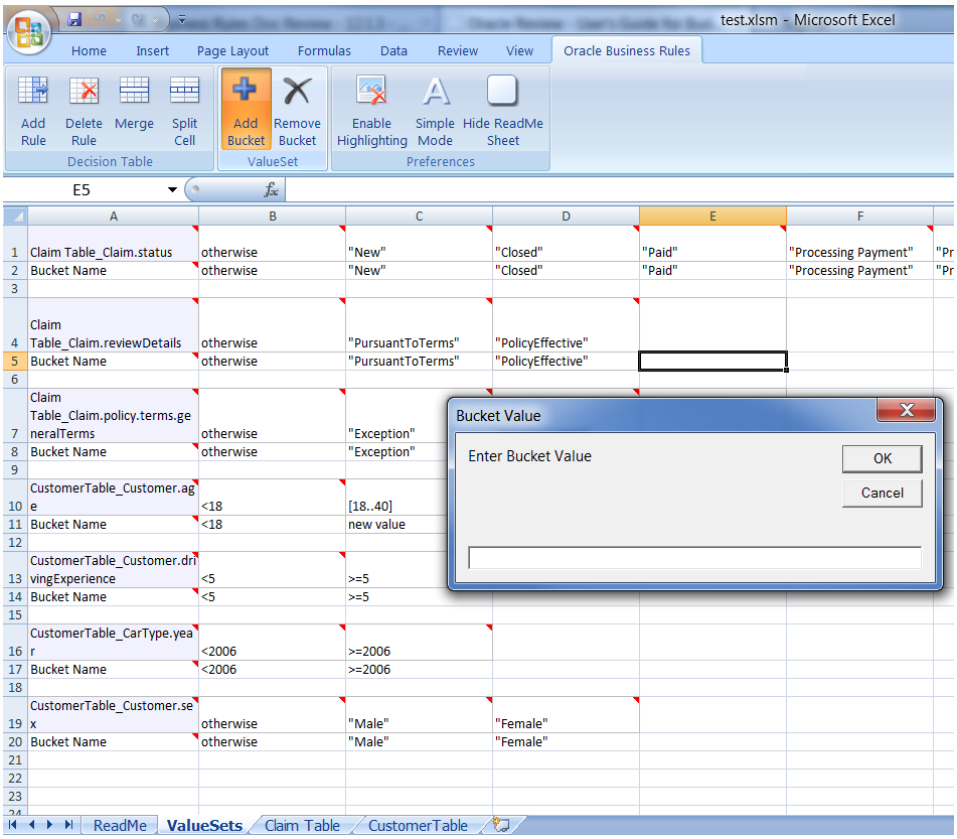


	A	B	C	D	E	F	G
1							
2		***Conditions***	**"Teen"**	**"High Risk Senior"**	**"Extremely High Risk Senior"**	**"Normal Risk Senior"**	**R5**
3	**"Age of Policy Holder"**	Customer.age	<18	>=60	(40..60)	new value	<18
4	**"Number of years licensed"**	Customer.drivingExperience	-	-	-	-	?
5	**"Year in which Car was made"**	CarType.year	-	<2006	<2006	>=2006	?
6	**"Gender of policy holder"**	Customer.sex	-	"Male"	"Female"	-	?
7							
8		***Actions***					
9	**A1**	assert new Terms	InActive	Active	Active	Active	InActive
10	Variable	coverages_en		"LOW"	"VERY LOW"	"MEDIUM"	
11	Fixed	generalTerms					
12	Variable	notes		"HIGH RISK"	"EXTREMELY HIGH	"NORMAL RISK"	
13	**A2**	assert new Policy	InActive	Active	Active	Active	InActive
14	Fixed	end Date					
15	Variable	id	0	6	7	8	0
16	Fixed	insured	Customer	Customer	Customer	Customer	Customer
17	Fixed	start Date					
18	Fixed	terms	Terms	Terms	Terms	Terms	Terms
19	Fixed	type					
20	Fixed	vehicles	CarType	CarType	CarType	CarType	CarType
21							
22							
23							
24							
25							
26							

Adding or Removing Value Sets

In the ValueSets tab, you can add or remove Value Sets, as shown in Figure 5-48. Depending on the cell you click in, your options will vary: endpoints or values.

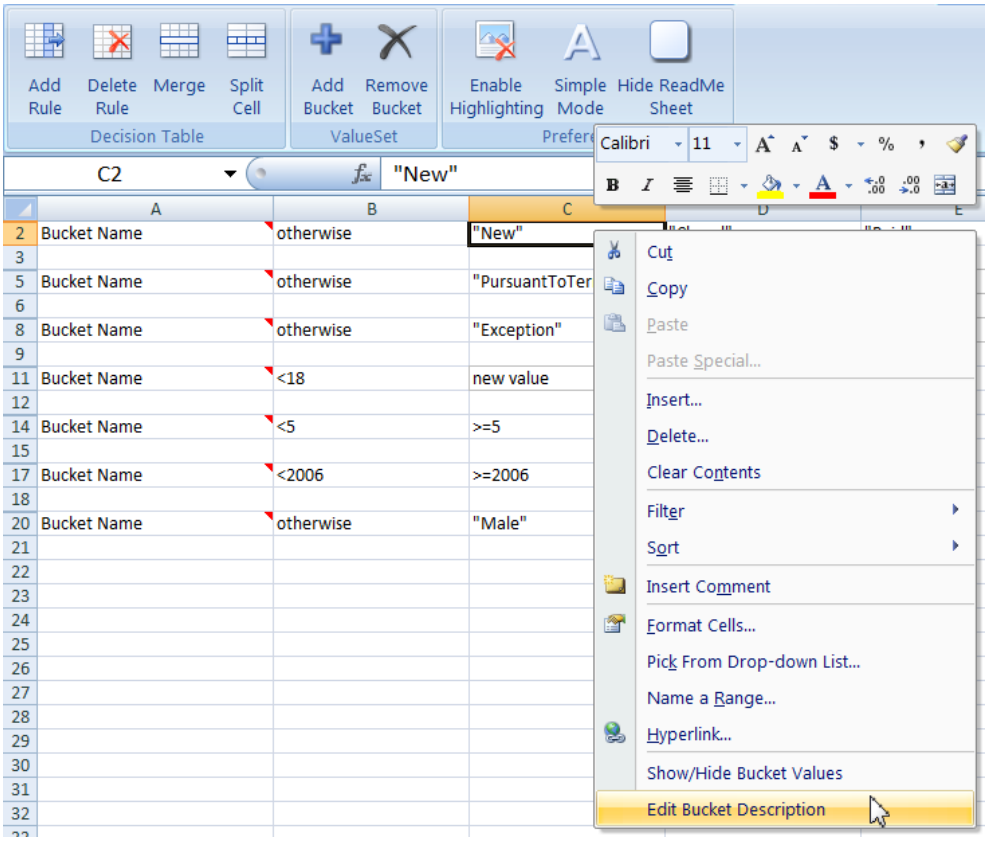
Figure 5-48 Value Sets Worksheet



Showing or Hiding Value Sets and Editing the Description

On the Value Sets worksheet, right click and select **Show/Hide Values** to toggle between viewing or hiding values as shown in Figure 5-49. You can also right click and select **Edit Bucket Description** to change the description.

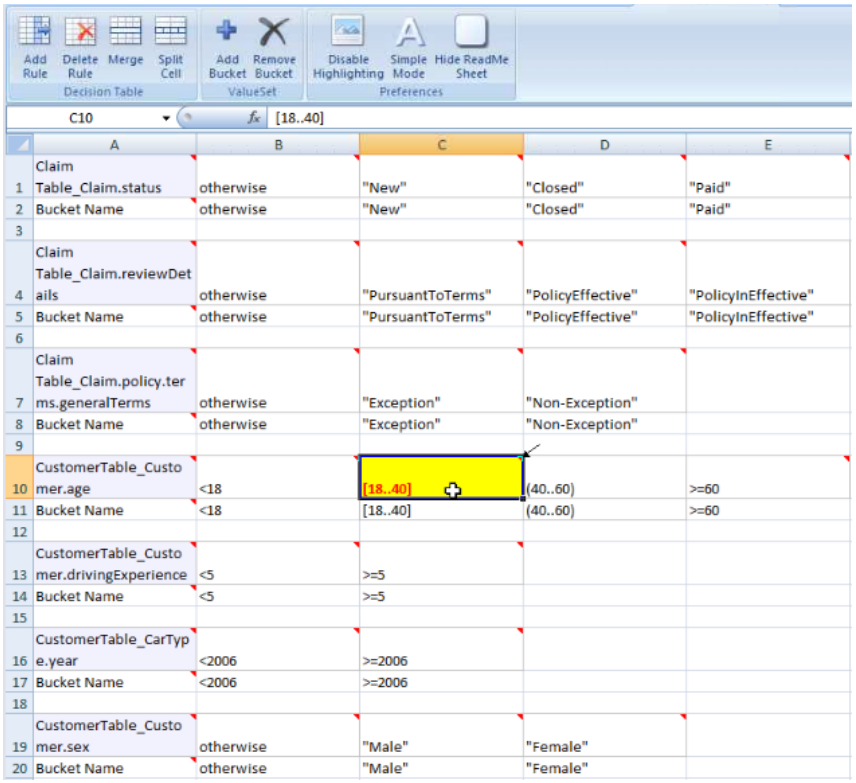
Figure 5-49 Show/Hide Value Sets



Setting Preferences

In the Value Sets tab, click Enable Highlighting or Disable Highlighting, as shown in Figure 5-50.

Figure 5-50 Enabling Highlighting



	A	B	C	D	E
1	Claim Table_Claim.status	otherwise	"New"	"Closed"	"Paid"
2	Bucket Name	otherwise	"New"	"Closed"	"Paid"
3					
4	Claim Table_Claim.reviewDetails	otherwise	"PursuantToTerms"	"PolicyEffective"	"PolicyInEffective"
5	Bucket Name	otherwise	"PursuantToTerms"	"PolicyEffective"	"PolicyInEffective"
6					
7	Claim Table_Claim.policy.terms.generalTerms	otherwise	"Exception"	"Non-Exception"	
8	Bucket Name	otherwise	"Exception"	"Non-Exception"	
9					
10	CustomerTable_Customer.age	<18	[18..40]	(40..60)	>=60
11	Bucket Name	<18	[18..40]	(40..60)	>=60
12					
13	CustomerTable_Customer.drivingExperience	<5	>=5		
14	Bucket Name	<5	>=5		
15					
16	CustomerTable_CarType.year	<2006	>=2006		
17	Bucket Name	<2006	>=2006		
18					
19	CustomerTable_Customer.sex	otherwise	"Male"	"Female"	
20	Bucket Name	otherwise	"Male"	"Female"	

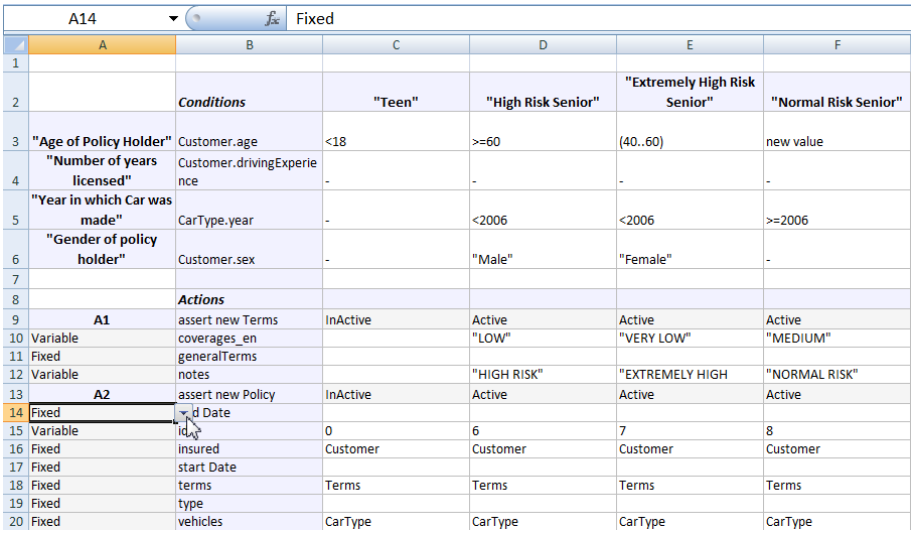
Using Simple or Advanced Mode

In your worksheet, click Simple Mode or Advanced Mode to toggle between the two modes.

Simple mode displays only the descriptions of conditions and actions and not the actual expressions. Also, action parameters are displayed, but you cannot specify them as fixed or variable.

Advanced mode displays both the descriptions and expressions for conditions and actions, as shown in Figure 5-51. Also, you can specify the action parameter type from fixed and variable, which is equivalent to specifying "Parameterized/Constant" in the SDK.

Figure 5-51 Advanced Mode

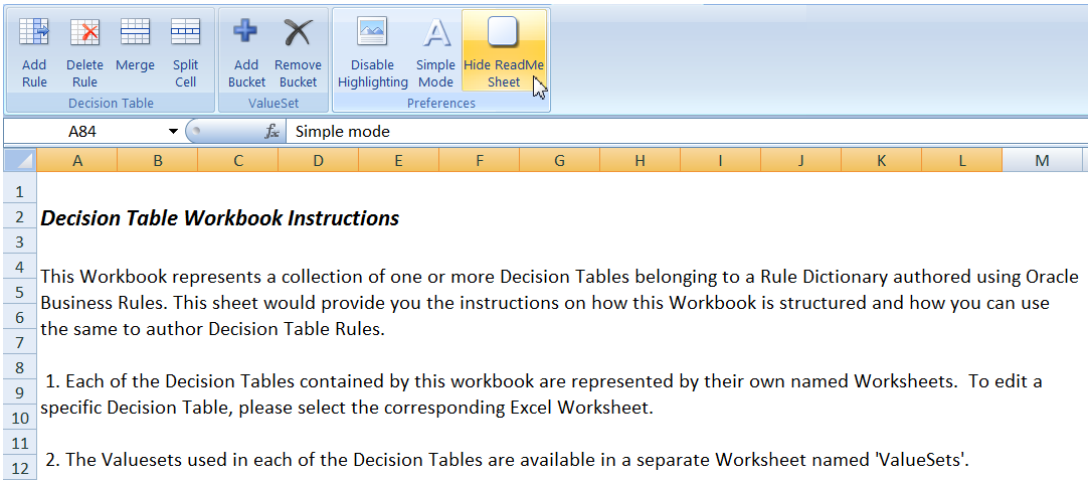


	A	B	C	D	E	F
1						
2		***Conditions***	**"Teen"**	**"High Risk Senior"**	**"Extremely High Risk Senior"**	**"Normal Risk Senior"**
3	**"Age of Policy Holder"**	Customer.age	<18	>=60	(40..60)	new value
4	**"Number of years licensed"**	Customer.drivingExperience	-	-	-	-
5	**"Year in which Car was made"**	CarType.year	-	<2006	<2006	>=2006
6	**"Gender of policy holder"**	Customer.sex	-	"Male"	"Female"	-
7						
8		***Actions***				
9	**A1**	assert new Terms	InActive	Active	Active	Active
10	Variable	coverages_en		"LOW"	"VERY LOW"	"MEDIUM"
11	Fixed	generalTerms				
12	Variable	notes		"HIGH RISK"	"EXTREMELY HIGH	"NORMAL RISK"
13	**A2**	assert new Policy	InActive	Active	Active	Active
14	Fixed	d Date				
15	Variable	id	0	6	7	8
16	Fixed	insured	Customer	Customer	Customer	Customer
17	Fixed	start Date				
18	Fixed	terms	Terms	Terms	Terms	Terms
19	Fixed	type				
20	Fixed	vehicles	CarType	CarType	CarType	CarType

Hiding or Showing the Readme Worksheet

Click Hide or Show ReadMe Sheet to toggle between the modes, as shown in Figure 5-52. The ReadMe worksheet provides helpful information about how to use the features on the Oracle Business Rules tab.

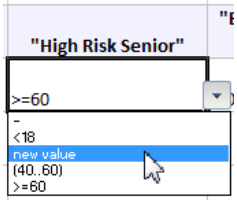
Figure 5-52 Show/Hide Readme



Editing Condition Cells

You can choose from the drop down or use auto-addition to add new values, shown in Figure 5-53. For some of the condition cells, you can only choose values from the drop down menu. These cells have been differentiated by using color code. Any conditions you change between a Value Set or Decision Table are automatically synced.

Figure 5-53 Editing Conditions



Editing Actions

You can select the action state (active/inactive) from the drop down, as shown in Figure 5-54.

Figure 5-54 Editing Action States

8		***Actions***			
9	**A1**	assert new Terms	InActive	Active	Active
10	Variable	coverages_en	Active InActive	DW"	"VERY LOW"
11	Fixed	generalTerms			
12	Variable	notes		"HIGH RISK"	"EXTREMELY HIGH
13	**A2**	assert new Policy	InActive	Active	Active
14	Fixed	end Date			
15	Variable	id	0	6	7
16	Fixed	insured	Customer	Customer	Customer
17	Fixed	start Date			
18	Fixed	terms	Terms	Terms	Terms
19	Fixed	type			
20	Fixed	vehicles	CarType	CarType	CarType

Editing Expressions

You can edit the values of action expression cells. Use care to maintain the validity of these cells when editing.

Editing Action Expression Parameters

You can make action parameters fixed or variable, as shown in Figure 5-55. If the action parameter is fixed, then all the rules will have the same value for that particular parameter. If the action parameter is variable, then different rules can have different values for that particular parameter.

Figure 5-55 Editing Action Expression Parameters

	Actions		
A1	assert new EURentRulesBase.DriverType	Active	InActive
Variable	e	18	25
Fixed Variable	stName	"FirstName"	"FirstName"
Fixed	lastName	null	null
Variable	licenseNumber	"ABCD1234"	"ABCD1234"

Editing Descriptions

You can edit descriptions for actions, conditions, and rules. If the description is not provided for any of the action or condition or rule then it will be defaulted to "A", "C" or "R" followed by a number which denotes its position in the decision table, respectively.

Figure 5-56 Editing Descriptions

	A	B	C	D
1				
2		***Conditions***	**Minor**	**R2**
3	**C1**	EURentRulesBase.DriverType.age	<18	3..60)
4	**C2**	EURentRulesBase.TCaseEvent.milestoneEvent.milestoneEvent	-	-
5	**C3**	EURentRulesBase.DriverType.firstName	"John"	"Carter"
6				
7		***Actions***		
8	**A1**	assert new EURentRulesBase.DriverType	Active	InActive
9	Variable	age	18	25
10	Fixed	firstName	"FirstName"	"FirstName"
11	Fixed	lastName	null	null
12	Variable	licenseNumber	"ABCD1234"	"ABCD1234"

Using the Auto-Addition Feature

You can add values in the value sets in two ways:

1. Go to the specific value set in the value sets worksheet. In the Oracle Business Rules tab, click Add Bucket.

2. Enter a value (in case of LOV valuesets) or end point (in case of Range valuesets) in the condition cell. This is called auto-addition as the value will be automatically added to the corresponding value set, as shown in Figure 5-57.

Figure 5-57 Entering a Value in the Condition Cell

1					
2		***Conditions***	**Minor**	**R2**	**R3**
3	**C1**	EURentRulesBase.DriverType.age	<18	[18..60)	60
4	**C2**	EURentRulesBase.TCaseEvent.milestoneEvent.milestoneEvent	-		
5	**C3**	EURentRulesBase.DriverType.firstName	"John"	"Carter"	-

-
<18
[18..60)
>=60

The value set above has three values: – 1) <18 , 2) [18..60) , and 3) >=60.

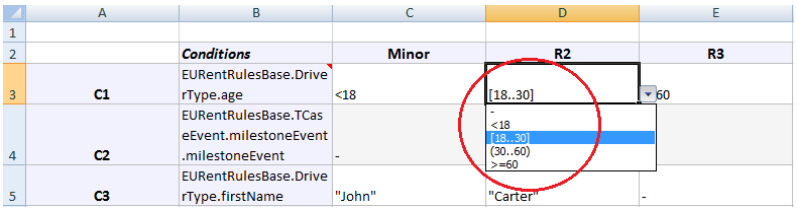
3. To add a new value, for example, [18..30] and (30..60), type 30 in the cell as shown in Figure 5-58 and press Enter.

Figure 5-58 Adding a New Value

	A	B	C	D	E
1					
2		***Conditions***	**Minor**	**R2**	**R3**
3	**C1**	EURentRulesBase.DriverType.age	<18	30	60
4	**C2**	EURentRulesBase.TCaseEvent.milestoneEvent.milestoneEvent	-	-	-

4. After you press enter, the value will be added to the value set and will be shown in the drop-down as shown in Figure 5-59.

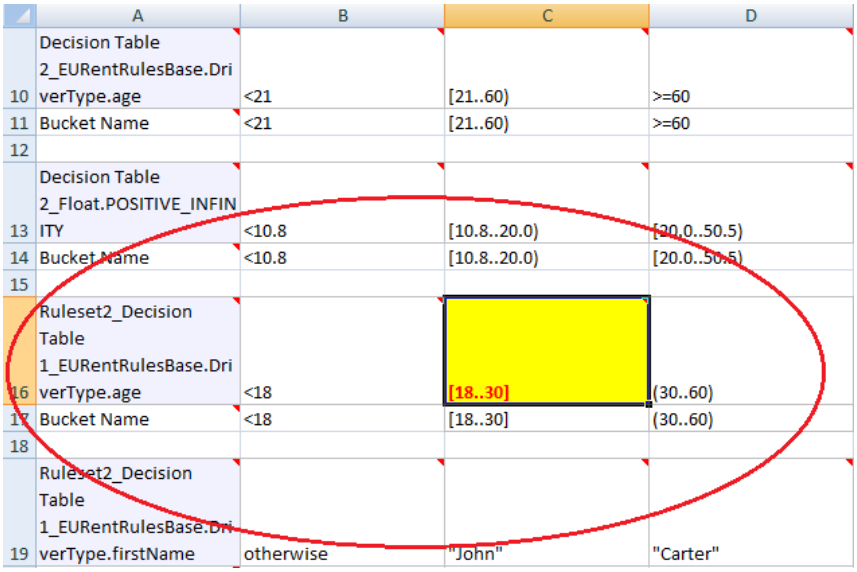
Figure 5-59 Value is Auto-Added



Various highlighting techniques are used to inform you about auto-added values in the value set, see the following examples. The comment and the highlighting of the value is removed after you select another value for any other rule for that condition or if a new value is added in the same value set.

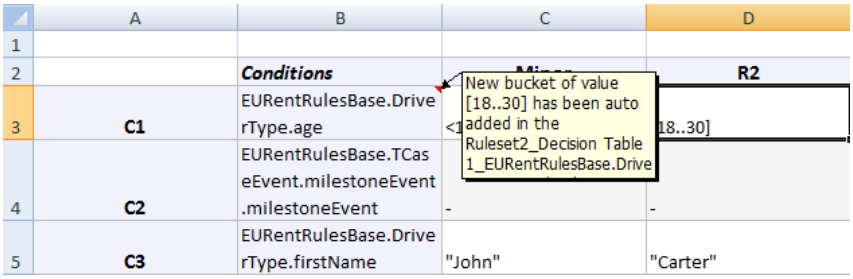
The first is to highlight the newly added value in the value set sheet as shown in Figure 5-60.

Figure 5-60 Highlighted Value Set



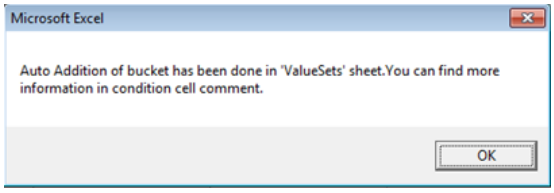
The second is the addition of a comment in the condition cell, as shown in Figure 5-61.

Figure 5-61 Comments in Condition Cells



The third is to print a message box, shown in Figure 5-62. Note that the box is only shown the first time when the value is auto-added.

Figure 5-62 Message Dialog



Aliases of Values in the Value Sets Worksheet

In the value sets sheet, there are two rows for every value set. The first row denotes the value and the second one denotes the alias of the value. It is the alias of the value that is shown in the drop-down of condition cells.The aliases can be edited. Any change made in aliases will be immediately available in corresponding condition cells.

Syncing Value Sets and Conditions

The value sets and condition cells are always in sync. Any change made in value set is promptly synced with the condition cells whether it is an addition/deletion of any value, or any change in the alias. The sync is always maintained between value set and the corresponding condition cells.

6

Working with Decision Functions

This chapter describes how to use a decision function to call rules from a Java application, from a composite, or from a BPEL process.

The chapter includes the following sections:

Introduction to Decision Functions

A decision function is a module of execution that can be invoked to reason on the inputs to arrive at outputs by applying a given ruleset or other decision functions.

A decision function contains the following declarations:

- Input facts.
- Rulesets and nested decision functions.
- Output facts.

A decision function performs the following operations:

- Asserts inputs as rule facts into the Oracle Business Rules Engine working memory.
- Runs rulesets configured in the current decision function and in nested decision functions in order.
- Returns output facts from the Oracle Business Rules Engine working memory.

You can create a decision function to simplify the use of Oracle Business Rules from a Java application or from a BPEL process. In a decision function the rules you want to use can be organized into several rulesets, and those rulesets can be executed in a prescribed order. Facts may flow to the first ruleset, and this ruleset may assert additional facts that flow to the second and subsequent rulesets until finally facts flow back to the decision function as decision function output.

Working with Decision Functions

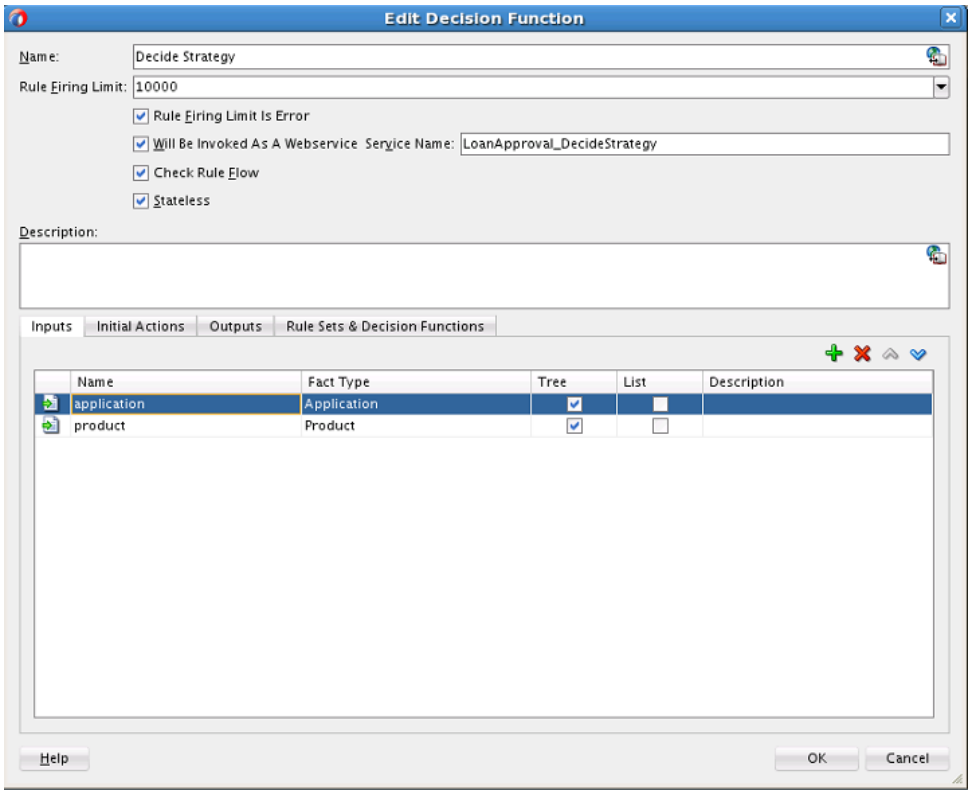
You use Rules Designer to add a decision function.

To add a decision function:

1. In Rules Designer, click the **Decision Functions** tab.
2. In the **Decision Functions** area, click the **Create** button.

   A new Decision Function is created and an **Edit Decision Function** dialog is displayed, as shown in Figure 6-1.

Figure 6-1 Edit Decision Function Dialog



3. Enter a name for the Decision Function in **Name** field and a description in the **Description** field.
4. In the **Rule Firing Limit** field, select **unlimited** or an integer value. In some cases when you are debugging a decision function, you may want to enter a value for the rule firing limit. The **Rule Firing Limit** can also be used in primary rules processing when you want the execution to stop after a specified number of rules fire.

   For more information, see What You Need to Know About Rule Firing Limit Option for Debugging Rules.
5. Select the appropriate decision function options:

   - **Rule Firing Limit is Error:** is used to indicate whether the rule firing limit is an error condition or not. Clear the check box if this is a scenario where you want rules processing to stop after *n* number of rules fire. When cleared, the rule firing limit is honored, but not reported as an error.

     Select this check box if this is a scenario where you are using the rule firing limit as a way to, for example, prevent an infinite loop. The system throws an error when the firing limit is reached.

- **Will be invoked as a Web Service:** select whether the decision function will be invoked as a Web Service and provide the Web Service name.
- **Check Rule Flow**: Rule flow checking verifies the following to generate validation warnings:
  - Types required by rules executed by the decision function are either inputs to the decision function or asserted by other rules.
  - Types generated by rules executed by the decision function are either inputs to other rules or outputs to the decision function.

    Rule flow checking might not identify rule flow issues spanning Java code that is used in rules. In such cases, the warnings can be ignored by turning off rule flow checking.
- **Stateless**: when selected, this option specifies that the decision function is stateless. For more information, see What You Need to Know About the Decision Function Stateless Option.

6. In the Inputs tab, click **Add** to add inputs. For each input in the Inputs Table, select the appropriate options:
   - **Name** - enter an input name and press **Enter** or accept the default name.
   - **Fact Type** - select the appropriate fact type from the list.
   - **Tree** - When cleared, the input is asserted using the `assert` function. When selected, the input is asserted using the `assertTree` function. When selected, all objects referenced by the root object(s) are asserted. For more information, see Working with Tree Mode Rules.
   - **List** - When unselected, the input must be a single object and the assertion applies only to that single input object. When selected, the input must be a `List` of objects and the assertion applies to each object in the input `List` (`java.util.List`).
   - **Description** - Description of the input.
7. In the Outputs tab, click **Add** to add outputs. For each output in the Outputs Table, select the appropriate options:
   - **Name** - enter an output name and press **Enter** or accept the default name.
   - **Fact Type** - select the appropriate fact type from the list.
   - **Tree** - When selected, this option sets a flag that enables certain design-time decision function argument checking. For an output argument, this option has no effect on runtime behavior. However, at design time in the case where several decision functions are called in a sequence, it is useful to notate explicitly that the output of one decision function is a tree. This implies that the input of another decision function in the sequence is expecting a tree as an input. For more information, see Working with Tree Mode Rules.
   - **List** - When unselected the output is a single object. When selected the output is a group of objects. For more information on the behavior of the List option on an output argument, see What You Need to Know to About Decision Function Arguments.

- **Description** - Description of the output.

Note:

The visible attribute of a fact type controls whether a fact can be matched by a rule or can be asserted. Non-visible fact types are visible when they are part of a visible fact type, or a variable.

When inputs or outputs of a decision function are non-visible fact types, then a visible wrapper fact type is generated for the non-visible inputs named DF.in and a visible wrapper fact type for the non-visible outputs named DF.out.

For example, if i is an input of type int and o is a output of type int, then the following rule copies input to output:

```
IF
    DF.in != null
THEN
    assert new DF.out(o: DF.in.i)
```

If you do not reference DF.in or assert DF.out, a rule flow warning occurs.

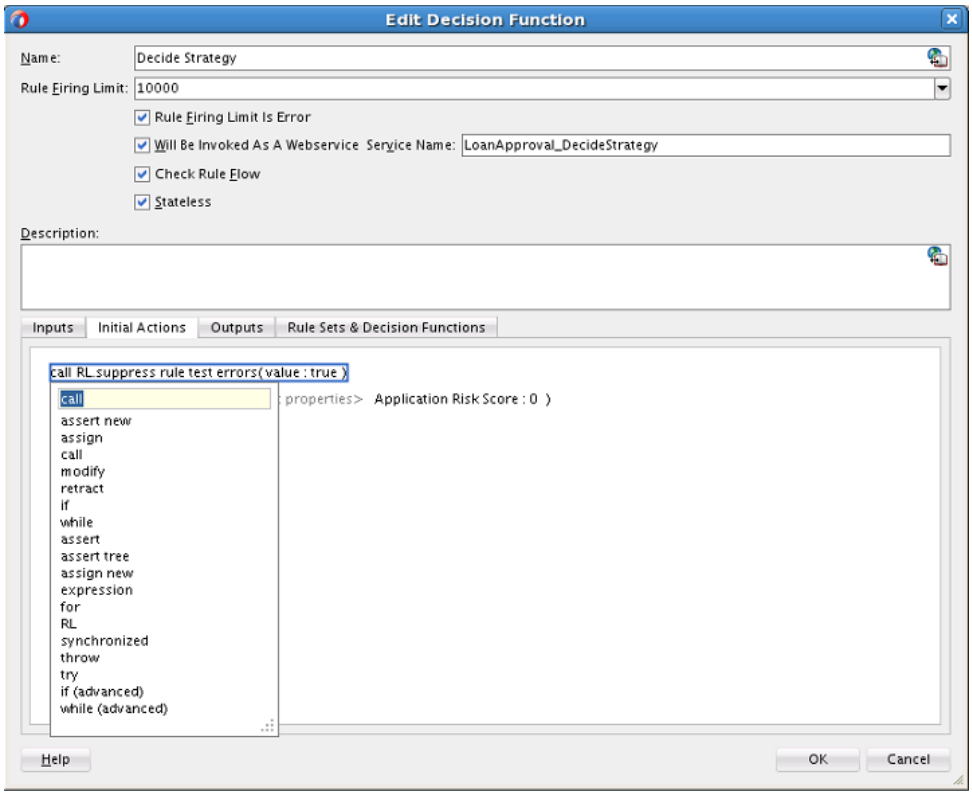
---

8. In the **Initial Actions** tab, you can add actions that could be used to change input facts before they are asserted, change the ruleset stack, set the effective date, or even assert output facts. These actions could be used instead of rules, or to "set up" the environment for running rules.

   Consider a situation where a decision function (DF1) calls another decision function (DF2) using the **Initial Actions** tab. DF1 is configured to push Ruleset1 to the ruleset stack. DF2 is configured to push Ruleset2. In DF1, before the initial actions are executed, Ruleset1 is pushed to the ruleset stack. Then, when DF2 is called, Ruleset2 is also pushed. So when rules start running, rules from both rulesets fire because of the ruleset stack. If you want to push Ruleset2 (because in the initial actions, you are calling DF2), you can use initial actions in DF1 to clear the ruleset stack before calling DF2, and push Ruleset1 on the stack after calling DF2.

   You can add any required action ranging from `assert`, `call`, `modify` to even conditional actions such as `if`, `else`, `elseif`, `while`, `for`, `if (advanced)`, and `while (advanced)` as shown in Figure 6-2.

Figure 6-2 Adding Initial Actions



Note:

If decision function DF1 contains DF2 in the **Rulesets & Decision Functions** tab, then DF2 may not have any initial actions.

The `if (advanced)` and `while (advanced)` conditional actions accept only `boolean` values. For each of the action conditions, you can add different test form types.

9. In the **Rulesets and Decision Functions** area, use the shuttle to move items from the **Available** box to the **Selected** box.
10. Select an item in the **Selected** box, and click **Move Up** or **Move Down** as appropriate to order the rulesets and the decision functions.

How to Edit an Existing Decision Function

To edit an existing decision function:

1. In Rules Designer, click the **Decision Functions** tab.
2. Select the decision function to edit and click the **Edit** button or double-click the decision function icon.
3. Edit the appropriate decision function fields in the same manner as you would when you add a decision function.

How to Change the Order of Inputs

To change the order of inputs:

1. In Rules Designer, click the **Decision Functions** tab.
2. Select the decision function to edit and click the **Edit** button or double-click the decision function icon.
3. Select the input argument you want to move. Click either **Move Up** or **Move Down** to reorder the input argument.

How to Change the Order of Outputs

To change the order of outputs:

1. In Rules Designer, click the **Decision Functions** tab.
2. Select the decision function to edit and click the **Edit** button or double-click the decision function icon.
3. Select the output argument you want to move. Click either **Move Up** or **Move Down** to reorder the output argument.

How to Edit a Decision Function

To edit a Decision Function

1. In Rules Designer, click the **Decision Functions** tab.
2. Select the Decision Function you want to edit and click the **Edit** icon.

   The **Edit Decision Function** dialog is displayed.
3. Make necessary changes using the process that you have used for adding a new Decision Function.

What You Need to Know About Rule Firing Limit Option for Debugging Rules

The **Rule Firing Limit** allows you to set the maximum number of steps (rule firings) that are allowed at runtime.

Using this option and specifying a value other than unlimited can help you debug certain rule design problems and in some cases might help prevent `java.lang.OutOfMemoryError` errors at runtime. This is can be useful when debugging infinitely recursive rule firings.

When you choose a value other than unlimited, and choose **Rule Firing Limit is Error**, the system throws an error once the limit is reached.

What You Need to Know to About Decision Function Arguments

Oracle Business Rules generates a corresponding RL Language function for each decision function.

The signature of a generated decision function is similar to:

```
function <name>(InputFactType1 input1, ... InputFactTypeN inputN) returns List
```

In a decision function, each parameter is generated depending on the **List** option, with the decision function input, as follows:

- Input argument, **List** option unselected: for FactType*i* the input must be a single object and the assertion applies only to that single input object.
- Input **List** option selected: for List<FactType*i*> the input must be a `List` of objects and the assertion applies to each object in the input `List` (`java.util.List`).

The generated RL Language function includes calls either to `assert` or `assertTree` for each argument, depending on the decision function Input option, **Tree**. When **Tree** is cleared, the input is asserted using the `assert` function. When **Tree** is selected, the input is asserted using the `assertTree` function. When selected, all objects referenced by the root object(s) are asserted.

For the decision function selected rulesets, as specified in the **Rulesets and Decision Functions** area **Selected** box, the generated RL Language function includes a call to `run()` with the selected rulesets in the selected ruleset stack order.

The generated RL Language function returns a list. The list has an element for each decision function output in order. If the output is declared to be a list, then the corresponding element is a list. However, if the output is not declared to be a list, then the corresponding element is the output fact or null (if there is no output fact of the declared type). If an output is not declared to be a list, and more than one output fact of the specified type is found in the working memory of Oracle Business Rules Engine, then an exception is thrown.

After you edit a decision function, for example, to change or add inputs and outputs, the changes are visible in BPEL for new Business Rule activities. However, the changes are not visible to existing Business Rule activities. For more information, see "Getting Started with Oracle Business Rules" in the *Developing SOA Applications with Oracle SOA Suite*.

What You Need to Know About the Decision Function Stateless Option

A decision function supports either stateful or stateless operation. The **Stateless** check box in the Edit Decision Function dialog provides support for these two modes of operation.

By default the **Stateless** check box is selected which indicates stateless operation. With stateless operation, at runtime, the rule session is released after each invocation of the decision function.

When **Stateless** is cleared, the underlying Oracle Business Rules object is kept in the memory of the Business Rules service engine, so that it is not given back to the Rule Session Pool when the operation is finished. A subsequent use of the decision function re-uses the cached RuleSession object, with all its state information from the previous invocation. Thus, when **Stateless** is cleared, the rule session is saved for a subsequent request and a sequence of decision function invocations from the same process should always end with a stateless invocation.

7

Testing and Validating Business Rules

This chapter describes how to test and validate the rules you have created or edited.

The chapter includes the following sections:

Overview

The test feature enables both developers and business users to quickly check that a rule satisfies the expected behavior or, if modified, to see if a rule regresses existing functionality.

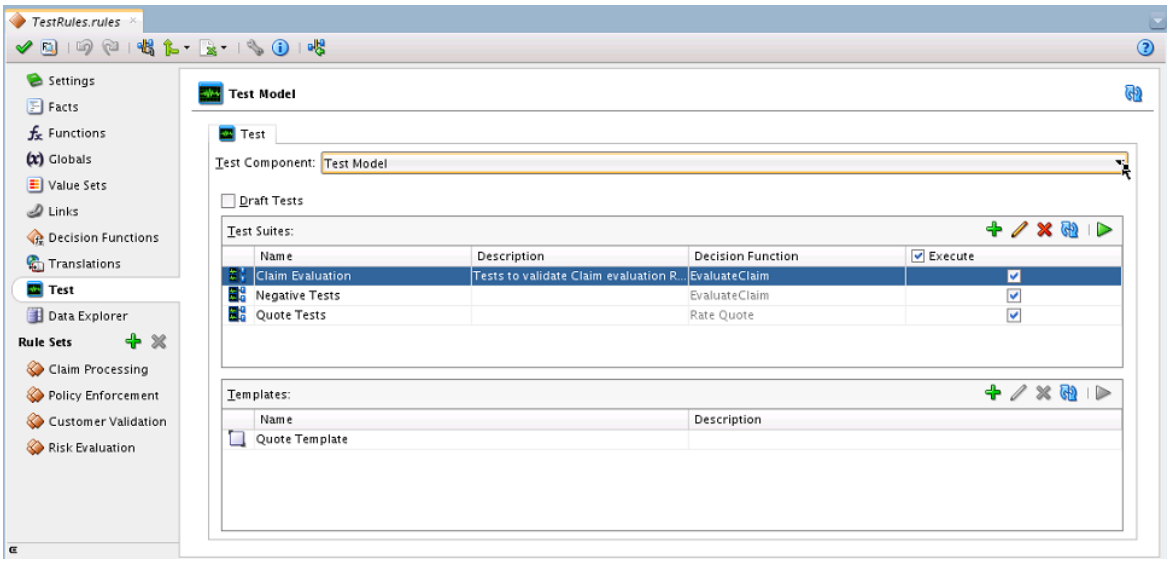
You can author and test rules at design-time in Oracle JDeveloper or Business Process Composer. At runtime, you can test rules in SOA Composer.

You can write tests declaratively, with no need for knowledge of XML or prior rules actions or programming languages such as Java. Additionally, tests support all types of facts (XML, Java, RL, and ADF-BC) and can be run on SOA or non-SOA use cases.

The test feature provides test reports with diagnostic comments and visual differences between the expected and actual values that can be used to correct the rules or fix the tests.

Figure 7-1 shows the UI in JDeveloper. For more information about using JDeveloper, see Introduction to Oracle JDeveloper in *Oracle Fusion Middleware Developing Applications with Oracle JDeveloper*.

Figure 7-1 Test Tab in JDeveloper



Components of the Test Feature

No matter which UI you use, the testing functionality behaves mostly the same way in JDeveloper, BP Composer, and SOA Composer.

Decision functions must have been already created before you begin--there is a one to one mapping between decision functions and tests. Once a decision function is associated with a test suite or test template, it cannot be changed later.

The components of the test feature are:

- **Test Suites and Test Cases**

  You can create a test suite with one or more test cases. You can also create templates that serve as templates for creating other test cases.

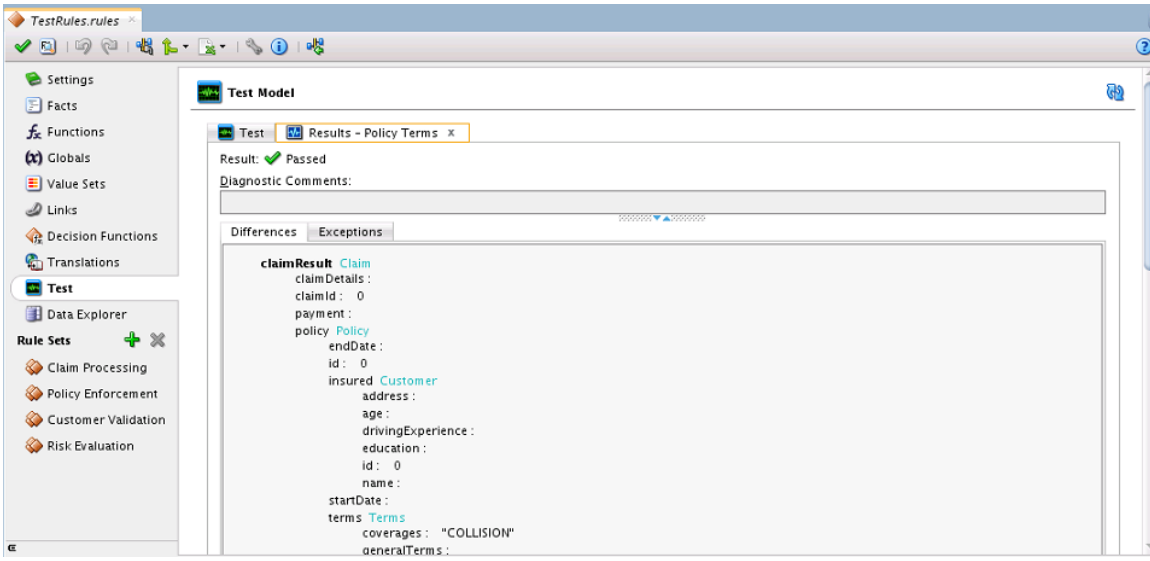
- **Test Templates**

  Test templates enable you to create similar test cases that differ only by the values of a few Fact properties. Templates also let you execute ad-hoc tests by specifying values for parameters. Ad-hoc tests enable you to perform sanity tests and try different value combinations for specific parameters before creating them as test cases in a test suite.

- **Test Execution**

  Executing test suites or test cases invokes the decision function and executes the rulesets defined in the decision function and presents the results in a new tab, as shown in Figure 7-2. Tests are executed via RL generation.

Figure 7-2 Test Results Tab



Keep the following in mind:

- Tests can be executed either from the Test tab of the Dictionary or from the Decision Functions tab.
- Tests can be defined in the current dictionary for decision functions in linked dictionaries.
- Tests defined in linked dictionaries can be executed in the current dictionary. Tests from linked dictionaries are available as read-only for execution.
- If you modify the inputs/outputs in a decision function, the changes are automatically synced to the tests you have defined. Tests are synced to fact-types referenced in the tests. If you remove facts from a decision function, the test feature enables you to delete those facts from the input/output tree of the test.
- A Fact is an instance of a FactType that defines the Test Data and has property values corresponding to each of the FactType properties. If a Property value is a complex data type, it is defined using Fact instances as well.

Testing Rules in JDeveloper

You can test your rules as you design them in JDeveloper.

In the Test Case editor, you define the inputs and expected output values for a Test Case. The values here can be simple values or expressions that use globals, functions, and so on.

The input and output Fact trees are auto-initialized based on the inputs/outputs specified for the Decision Function.

The test input and output Fact trees are also auto-synchronized with any changes to the Decision Function (if you add, delete, modify inputs or outputs) or fact types (if you add, delete or modify properties). The auto-synchronization flags and highlights invalid Facts or Property values that were changed in a Decision Function or Fact type. These flags in the test input/output help you to identify and fix issues in your test definitions.

Testing Permission Related Cases

Before testing permission related cases, change the refresh time 10 seconds. This is an important prerequisite for reliable test results.

To change the refresh time to 10 seconds:

1. Navigate to `$MW_HOME/user_projects/domains/soainfra/config/fmwconfig`.
2. Open `jps-config.xml`.
3. Set `oracle.security.jps.ldap.policystore.refresh.interval` to 10000.

```
<serviceInstance name="pdp.service" provider="pdp.service.provider">
    <property name="oracle.security.jps.ldap.policystore.refresh.interval"
value="10000"/>
</serviceInstance>
```

4. Restart the server.

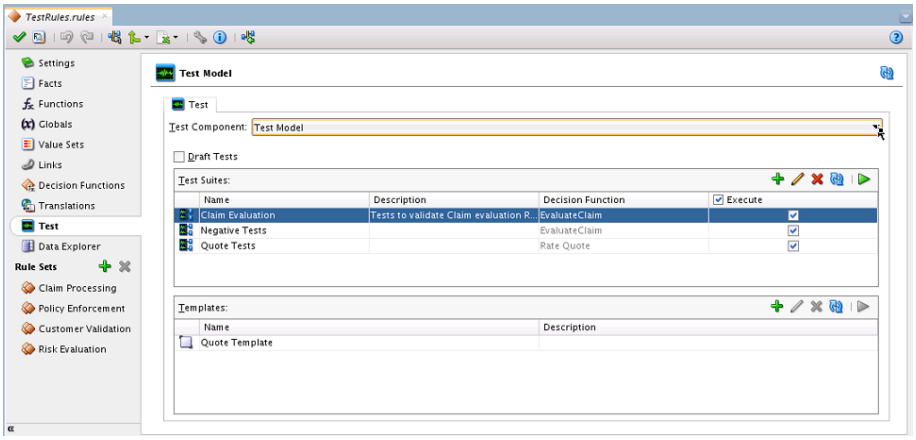
How to Create and Manage Test Suites and Cases

You can create a test suite with one or more test cases. Test suites can only be defined for specific decision functions.

For more information about decision functions, see Working with Decision Functions.

For detailed documentation of fields and other UI controls, click **Help** from within JDeveloper.

Figure 7-3 Test Tab in JDeveloper

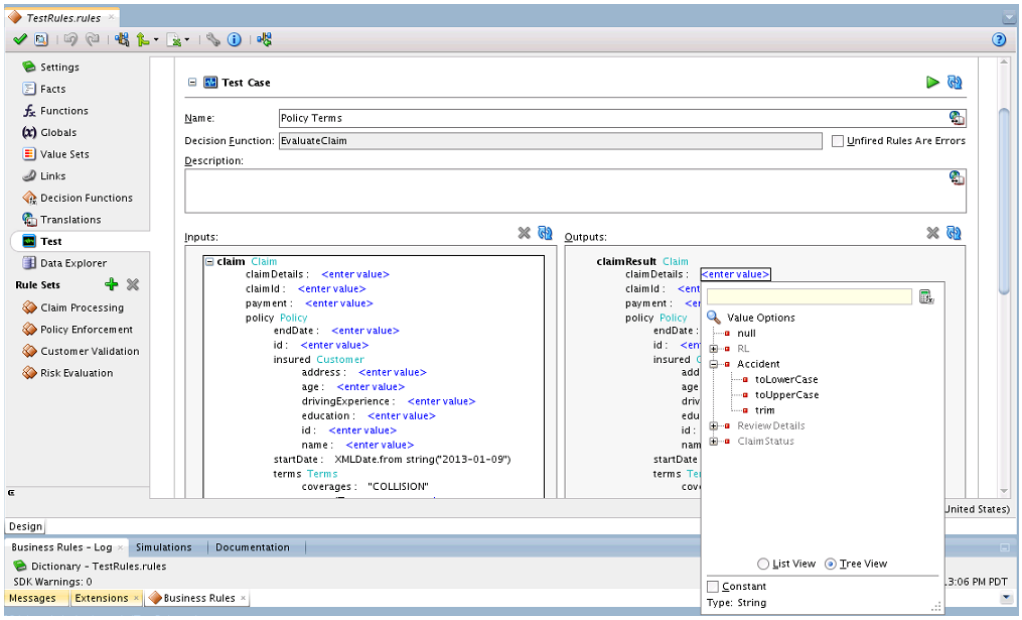


To create a test suite:

1. In Rules Designer, click the **Test** tab.
2. Click the **Test Component** drop down and select a **Test Model** from the list.
3. Click **+** to create a new test case for the test suite.
4. Enter a **name**, choose a **Decision Function**, enter a **Description**.
5. Click **OK**. The test suite is displayed.

6. Click **Edit** to review the **Input** and **Output** documents for the test case. This is where you can edit values to specify the input and the expected output, as shown in Figure 7-4:

Figure 7-4 Inputs and Outputs



7. Check the **Unfired Rules are Errors** check box if unfired rules are treated as errors from the execution.

8. Click the **Draft Test** check box to turn off the test validation.

When you have finished creating test suites and cases, you can run them. For more information, see How to Run Test Suites or Cases.

How to Create Test Templates

Test templates enable you to reuse input and output values to repeat tests on those fields and values.

For detailed documentation of fields and other UI controls, click **Help** from within JDeveloper.

To create a test template:

1. In Rules Designer, click the **Test** tab.

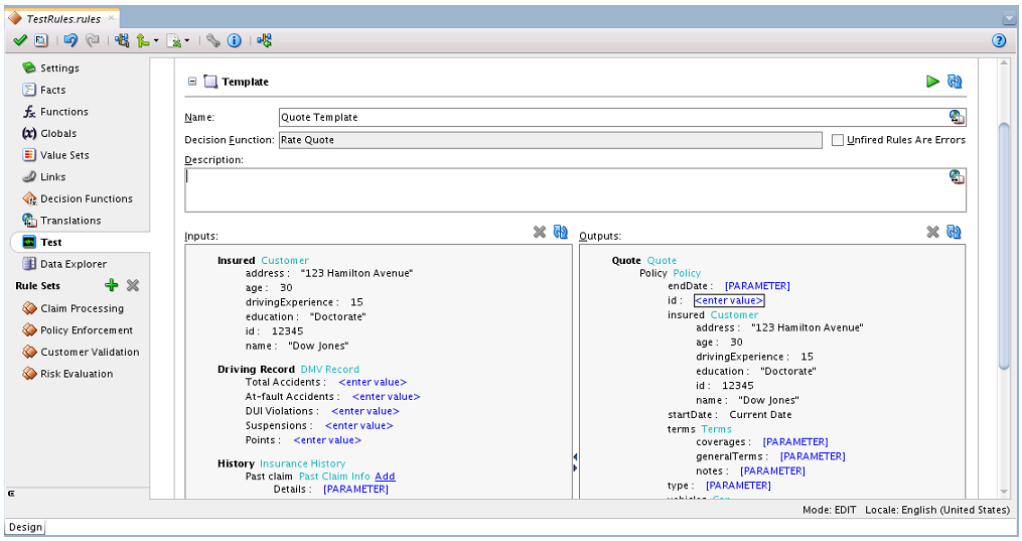
2. Click the **Test Component** drop down and select the **Test Model/Templates** from the list.

3. Go to the Templates table and click **+** to create a new test template.

4. Enter a **name**, choose a **Decision Function**, enter a **Description**.

5. Click **OK.** The test template is displayed, where you can see the Input and Output documents. This is where you can edit values to specify the input and the expected output, as shown in Figure 7-5.

Figure 7-5 Test Template



When you have finished setting up your test templates, you can run them. For more information, see How to Run Ad-hoc Tests from Test Templates.

How to Run Test Suites or Cases

When you run a test, a new tab is opened, and you can see the diagnostic comments, exceptions, and test results. Tests can be run either as a suite, multiple test cases, or as individual test cases.

For detailed documentation of fields and other UI controls, click **Help** from within JDeveloper.

To run a test suite or case:

1. Select a Test Suite or Test Case to run, and click **Execute.**
2. A new tab opens. A new tab Results opens. Click it to see the test results.

   For test suite execution, the tab shows a summary of the test results by default, but you can double-click each test case to see its test results. For test case execution, the tab shows the test results.

   If a test fails, the test results will show diagnostic comments and output differences or exceptions depending on the cause of the failure.

The execute button is enabled only when a test suite or test case (or test template) is selected from the table and as long as there are no validation warnings in the current dictionary.

How to Run Ad-hoc Tests from Test Templates

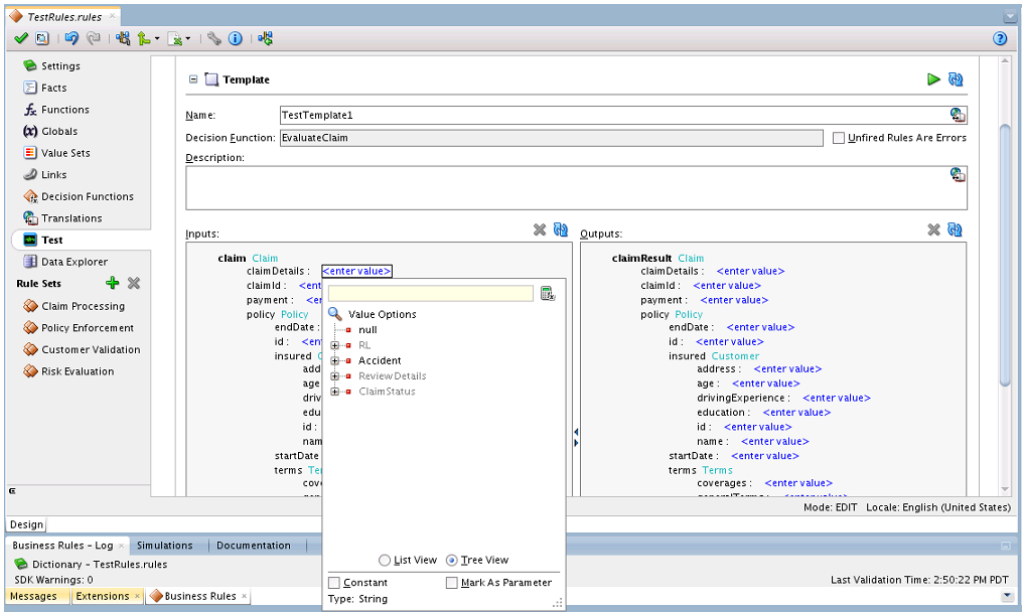
This is where you can run ad-hoc tests from templates by editing the nodes in input and output trees. The inputs and outputs are from the decision function.

For detailed documentation of fields and other UI controls, click **Help** from within JDeveloper.

To run ad-hoc tests from test templates:

1. Go to the **Templates** table and select a test template. The inputs and outputs fields are displayed, as shown in Figure 7-6.

Figure 7-6 Template Inputs and Outputs



2. Enter values for variable or parameter fields and click **Run**. The **Ad-Hoc Test** dialog is displayed.

   Select the appropriate options:

   - Check the **Unfired Rules Are Errors** check box if unfired rules are treated as errors from the execution.
   - Click **List View** or **Tree View** to toggle between the views.
   - Check the **Constant** check box and select a constant from the list.
   - Check the **Mark as Parameter** check box for variable fields of the test template. Values for variable fields are entered when the template is consumed like when the template is tested or used to create a test case.

3. Click **Execute Test** to run the template.

4. From the new **Results** tab, check the test results.

How to Run Tests for a Specific Decision Function

You can run tests from the Decision Function tab. This view only shows you the Test Suites and Test Cases for the specific Decision Function.

For detailed documentation of fields and other UI controls, click Help from within JDeveloper.

To run tests for a specific decision function:

1. In Rules Designer, click the **Decision Functions** tab.

2. Click to select a test case and click the Test button.
3. The Decision Function Test Editor dialog appears. This dialog is just another view of the testing feature.

Testing Rules in Business Process Composer

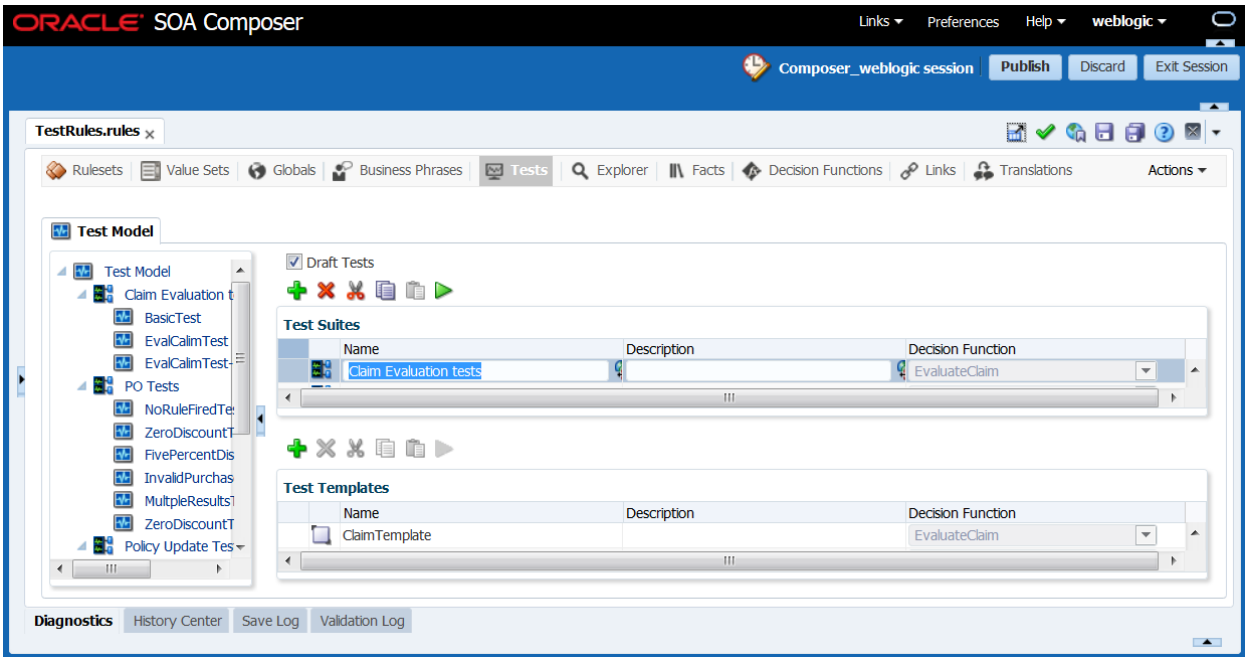
You can test your rules as you design them in Business Process Composer.

For more information about using Business Process Composer, see Introduction to Oracle Business Process Composer in *Oracle Fusion Middleware Developing Business Processes with Oracle Business Process Composer*.

Testing Rules in SOA Composer

At runtime, you can use SOA Composer to regression test rules. This enables business users to quickly check if a modified rule changes the existing functionality. Figure 7-7 shows the Tests tab in SOA Composer. The Tests tab only appears if you have a deployed composite and are in a SOA Composer session. Click **Create Session** to open a session.

Figure 7-7 Tests Tab in SOA Composer

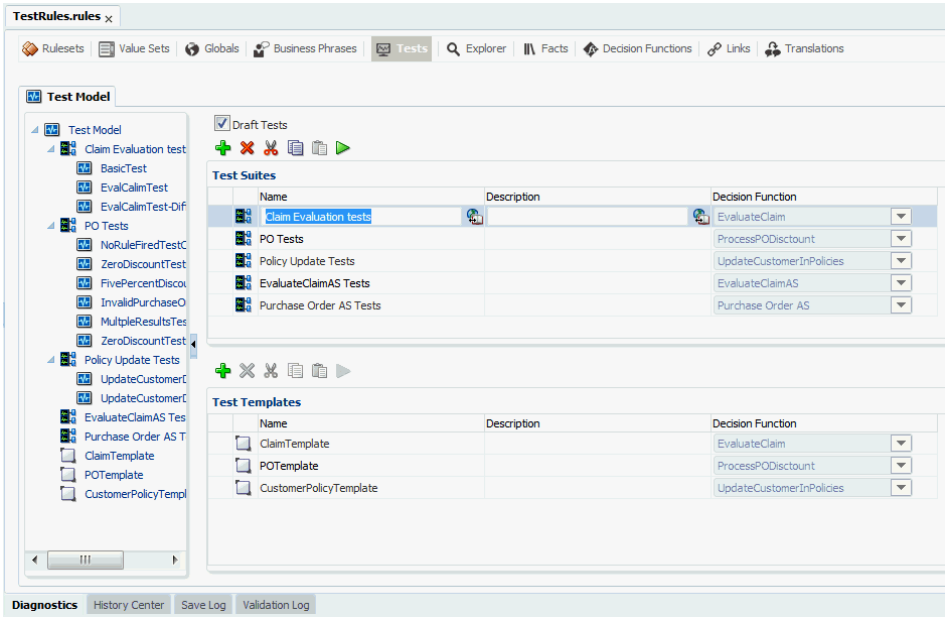


How to Create and Manage Test Suites and Cases

You can create a test suite with one or more test cases. Test suites can only be defined for specific decision functions.

For more information about decision functions, see Working with Decision Functions.

Figure 7-8 Test Suite Page



For detailed documentation of fields and other UI controls, click **Help, Help for This Page** from within SOA Composer.

To manage test suites and cases:

1. In Rules Designer, click the **Tests** tab and click in a Test Suite row to enable the action buttons.

   Click the **Draft Tests** check box if you want to turn off test validation.

2. Click **+** to create a new Test Suite.

3. Enter a **Name** and **Description**, then choose a **Decision Function.**

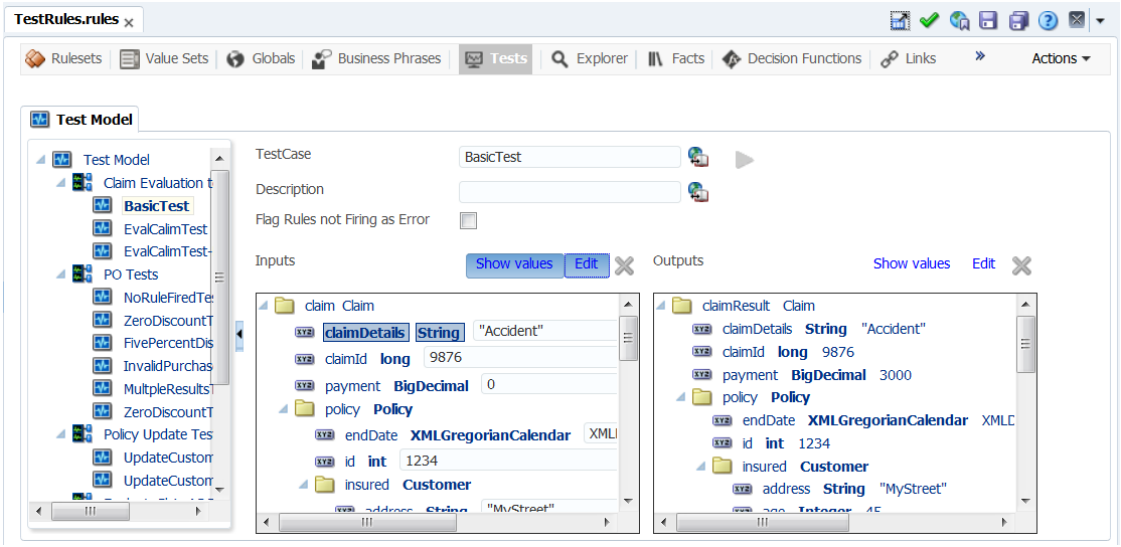
4. The test suite is displayed.

5. After creating a test suite, if you want to create test cases, click the test suite in the Test Model tree and click **+** to create a **Test Case** or a **Test Case from Templates**.

6. You can **Save Changes in Current Tab** to save data at any time or click **Publish** if you are done with changes.

7. You can also click a test case in the Test Model tree to see the **Input** and **Output** documents for the test case. This is where you can edit values to specify the input and the expected output, as shown in Figure 7-9.

Figure 7-9 Inputs and Outputs from Decision Functions



In the Test Case editor, you define the inputs and expected output values for a Test Case. The values here can be simple values or expressions that use globals, functions, and so on.

The test input and output Fact trees are auto-initialized based on the inputs and outputs specified for the Decision Function.

The input and output Fact trees are also auto-synchronized with any changes to the Decision Function (if you add, delete, modify inputs or outputs) or fact types (if you add, delete or modify properties). The auto-synchronization flags and highlights invalid Facts or Property values that were changed in a Decision Function or Fact type. These flags in the test input/output help you to identify and fix issues in your test definitions.

8. Click **Edit** to make all of the nodes in the tree editable.
9. If you edit a field in the tree, click **Show Values** to show only those values.
10. Check the **Flag Rules not Firing as Error** check box if unfired rules are treated as errors from the execution.

When you have finished setting up your test suites and cases, you can run them. For more information, see How to Run Test Suites or Cases.

How to Create Test Templates

Test templates enable you to reuse input and output values to repeat tests on those fields and values.

For detailed documentation of fields and other UI controls, click **Help, Help for This Page** from within SOA Composer.

To create test templates:

1. In Rules Designer, click the **Tests** tab.
2. Click **Test Model** in the navigation tree. In the **Test Templates** region, click **+** to create a new test template.

3. Enter a **Name**, **Description**, and choose a **Decision Function.**
4. Click **Save.**

To run ad-hoc tests from test templates, see How to Run Ad-hoc Tests from Test Templates.

How to Run Test Suites or Cases

When you run a test, a new tab is opened, and you can see the diagnostic comments, exceptions and test results. Tests can be run either as a suite, multiple test cases or as individual test cases. Tests are executed via RL generation.

For detailed documentation of fields and other UI controls, click **Help, Help for This Page** from within SOA Composer.

To run test suites or cases:

1. Select a Test Suite or Test Template to run, and click **Execute.**

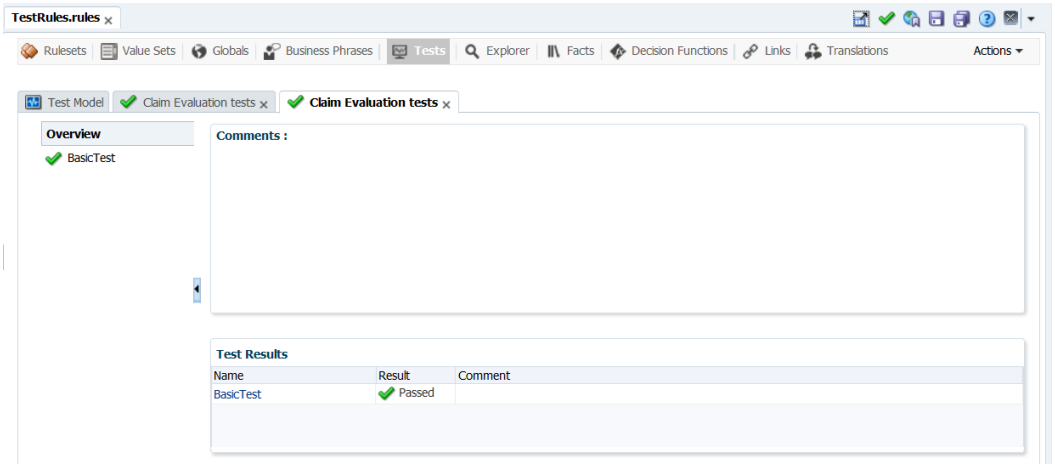
   The execute button is enabled only when a test suite or test case (or test template) is selected from the table and as long as there are no validation warnings in the current dictionary.

2. A new tab, Results appears. Click it to see the test results.

   For test suite execution, the tab shows a summary of the test results by default, but you can double-click each test case to see its test results. For test case execution, the tab shows the test results.

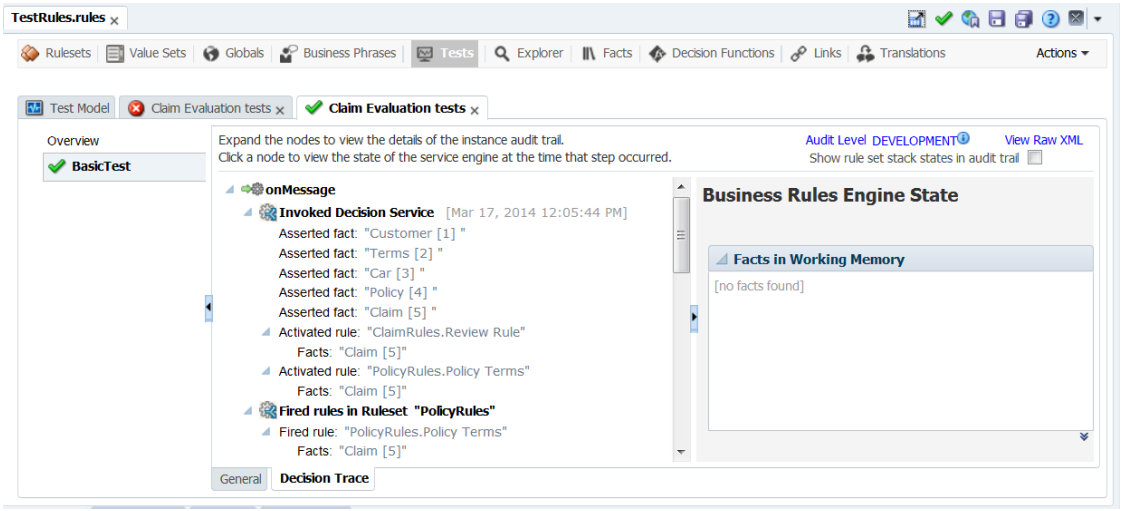
   If a test fails, the test results will show diagnostic comments and output differences or exceptions depending on the cause of the failure.

Figure 7-10 Diagnostic Comments for a Test Suite



If you select a test suite from the Test Model tree and run it, you can see the Decision Trace tab, as shown in Figure 7-11.

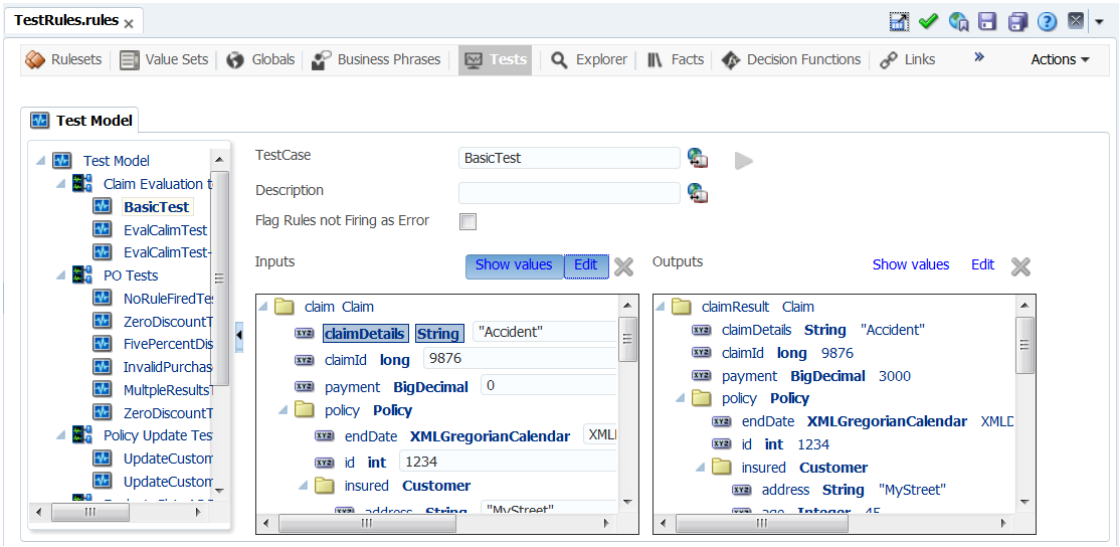
Figure 7-11 Decision Trace UI



How to Run Ad-hoc Tests from Test Templates

The Input and Output trees are loaded with input and output facts from the associated Decision Function, as shown in Figure 7-12. If you modify facts in a Decision Function, those changes are automatically synced to the Input and Output facts.

Figure 7-12 Input and Output Facts



For detailed documentation of fields and other UI controls, click **Help, Help for This Page** from within SOA Composer.

To run ad-hoc tests from test templates:

1. To see the **Input** and **Output** facts, click to choose a template from the Test Model tree.

   This is where you can edit values to compare the input with the expected output. You can add dynamic values here or check **Mark as Parameter** to be able to enter values when the rule is executed.

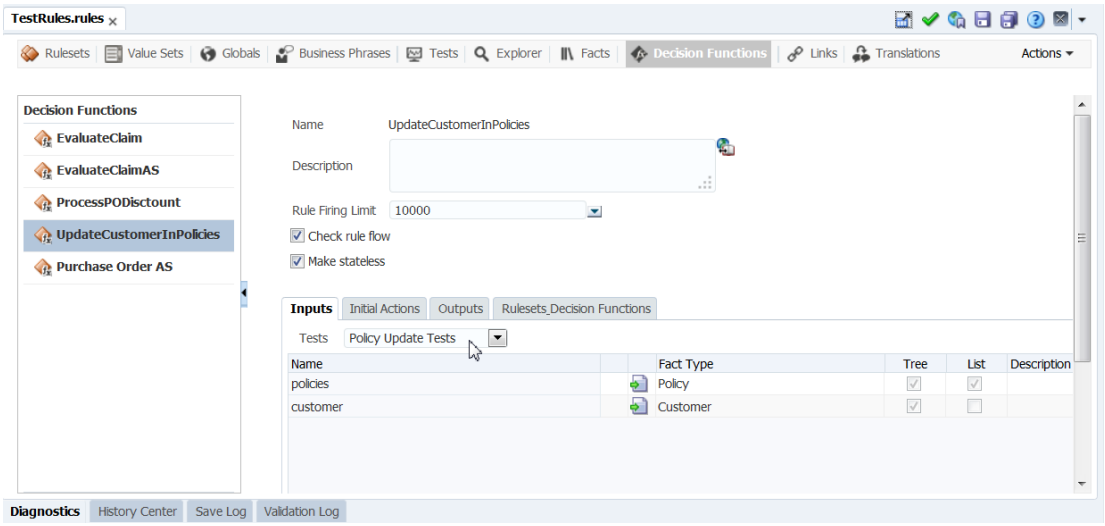
For more information about how to use the Expression Builder, see Working with Tree Mode Rules.

2. Click the **Execute Test Template** button to run the template. A dialog box appears.
3. Enter values for those variable or parameter fields and click **Run** from the dialog.
4. From the new **Results** tab, check the test results. Click the **Decision Trace** tab to see the audit trail.

How to Run Tests for a Specific Decision Function

You can run tests for specific Decision Functions, as shown in Figure 7-13:

Figure 7-13 Decision Functions Tab



For detailed documentation of fields and other UI controls, click Help from within JDeveloper.

To run tests for a specific Decision Function:

1. Click the **Decision Functions** tab and select the appropriate Decision Function from the list.
2. In the **Tests** field, use the dropdown to select the appropriate test.
3. The test is opened in the **Tests** tab. Click the test, and then click **Execute** to run your test.
4. The results tab appears. Click the new tab to view test results.

Testing Decision Functions Using a Rules Function

You can test rulesets by creating a decision function and calling the decision function from Rules Designer with an Oracle Business Rules function. In the body of the Oracle Business Rules function you create input facts, call a decision function, and validate the facts output from the decision function. For more information, see Introduction to Decision Functions and Introduction to Oracle Business Rules Functions.

To test a decision function using an Oracle Business Rules function:

1. Confirm that your dictionary is valid.

   For more information on dictionary validation, see How to Validate a Dictionary

2. In Rules Designer, select the **Functions** navigation tab.
3. In the **Functions** area click the **Create...** button.
4. Enter the function name in the **Name** field, or use the default name.
5. Select the return type from the **Return Type** list.

   For a test function, select `boolean`.

6. In the **Arguments** table, confirm that there are no arguments. For a test function, you cannot specify any arguments.
7. In the **Body** area, enter the test function body.

   In the body of the test function you can call a decision function using `assign new` to call and get the return value of the decision function (in the body of the test function you create input facts, call a decision function, and validate the facts output from the decision function).

   A decision function call returns a `List`. Thus, to test a decision function in a test function you do the following:

   - You create the input data as required for the decision function input arguments.
   - You call the decision function with the arguments you create in the test function.
   - You place results in a `List`, for example, in the following:

     ```
     assign new List resultsList = DecisionFunction_1(testScore)
     ```

8. Select the function and click the **Test Function** button.

   The function is executed. The output is shown in a Function Test Result dialog.

9. Click **OK** to dismiss the Function Test Result dialog.

What You Need to Know About Testing Decision Functions

You can use Oracle Business Rules Functions to test decision functions from within Rules Designer. Keep the following points in mind when using a test function:

- The **Test Function** button is gray if the dictionary associated with the test Oracle Business Rules Function contains any validation warnings. The **Test Function** button is only shown when the dictionary validates without warnings.
- To enable logging you can call `RL.watch.all()`. For more information on RL Language functions, see *Rules Language Reference for Oracle Business Process Management*. In this guide, `RL.watch.all()` is an alias for the RL Language function `watchAll()`.
- As an alternative to the example above, you can enter the function body that is shown in the example below:

```
call RL.watch.all()
assign new TestScore testScore = new TestScore()
modify (testScore, name: "Bill Reynolds", testName: "Math Test", testScore: 81)
assign new TestGrade testGrade = (TestGrade)DecisionFunction_1(testScore).get(0)
return testGrade.grade == Grade.B
```

For the `testScore` value 81, this function returns "Test Passed." For the `testScore` value 91, this returns "Test Failed."

This function runs and shows the `RL.watch.all()` output. The dialog shows "Test Passed" when the grade is in the B range. The dialog shows "Test Failed" when the grade asserted is not in the B range.

Testing Decision Services in SOA Composites

In a BPM or SOA application that uses Oracle Business Rules with a Decision Service, you can test rules at runtime with Oracle Enterprise Manager Fusion Middleware Control Console Test Web Service page, where you can create an instance of your composite for testing.

For more information about how to create a test instance of your composite after you have finished designing and deploying it, see Initiating a SOA Composite Application Test Instance in *Oracle Fusion Middleware Administrating Oracle SOA Suite and Oracle Business Process Management Suite*.

You can see the audit trail for the Decision Service execution. For more information, see Monitoring Business Rule Tracing in *Oracle Fusion Middleware Administrating Oracle SOA Suite and Oracle Business Process Management Suite*.

8

Working with Rules in Standalone (Non SOA/BPM) Scenarios

When using rules in standalone (non SOA or BPM) scenarios, you can create RuleSession rules, or you can use the Decision Point API.

For more information about using a RulesSession object, see Using a RuleSession in *Rules Language Reference for Oracle Business Process Management*.

For information about using the Decision Point API, see Introduction to the Rules SDK Decision Point API.

This chapter includes the following sections:

- Loading a Dictionary from the Repository
- Executing a Rule Dictionary
- Introduction to the Rules SDK Decision Point API
- Creating a Dictionary for Use with a Decision Point
- Creating a Java Application Using Rules SDK Decision Point
- Running the Car Rental Sample
- What You Need to Know About Using Decision Point in a Production Environment
- What You Need to Know About Decision Point and Decision Tracing

For more information on APIs that are referred to in this chapter, see *Java API Reference for Oracle Business Rules*.

Loading a Dictionary from the Repository

Non-SCA (SOA/BPM) applications typically package the rule dictionaries used by the application such that they are copied to the MDS repository when the application is deployed. In order to use the dictionaries at runtime, they will need to be retrieved from MDS. The basic access method is to use the RuleRepository API to access the dictionaries. A simple usage example is shown below.

```
RuleRepository rr = RepositoryManager.getMDSRuleRepository(null);
 // pkg and name are the dictionary package and name (Strings).
 // Alternatively you could construct a DictionaryFQN
 RuleDictionary rd = rr.load(pkg, name);
 // if this is an editing session, edit session occurs and
 // then save it when done.
 rr.save(rd);
 // when the RuleRepository will not be used again it must be closed.
 rr.close();
```

It is typical that an application will want to react when a dictionary has been modified, reload the dictionary and begin using the new rule definitions. Detecting when a dictionary must be reloaded is complicated when linked dictionaries are used since it may not be the root dictionary that was modified.

In this situation, the oracle.rules.sdk2.repository.DictionaryLoader class is the recommended mechanism for loading dictionaries for rule execution. DictionaryLoader tracks the dictionaries that are loaded including linked dictionaries and can determine when a dictionary it has loaded needs to be reloaded. DictionaryLoader uses DictionaryFinder instances to load all dictionaries. DictionaryFinders are added to a DictionaryLoader instance with the addFinder method in the order in which they will be invoked to find a dictionary. That is, the first finder added will be the first one to attempt to load a dictionary. When all the desired DictionaryFinders have been added to the DictionaryLoader, the loadDictionary() method is used to load a dictionary. The reloadNeeded method can be invoked to determine if a dictionary needs to be reloaded.

```
//
 // A RuleRepository instance is needed for its built-in DictionaryFinder
 //
 RuleRepository rr = RepositoryManager.getMDSRuleRepository(null);
 DictionaryFinder rrf = rr.getDictionaryFinder();

 //
 // Create the dictionary loader
 //
 DictionaryLoader dloader = new DictionaryLoader();
 dloader.addFinder(rrf);
 //
 // If the DecisionPointDictionaryFinder is required, add it
 //
 dloader.addFinder(new DecisionPointDictionaryFinder());

 //
 // Load the dictionary
 //
 DictionaryFQN fqn = new DictionaryFQN("somepackage", "myDictionary");
 RuleDictionary rd = dloader.loadDictionary(fqn);

 //
 // Check if the dictionary needs to be reloaded
 //
 if (dloader.reloadNeeded(fqn))
     rd = dloader.loadDictionary(fqn);

 //
 // When usage is complete, the RuleRepository must be closed.
 //
 rr.close();
```

Executing a Rule Dictionary

There are two approaches to initializing the RuleSessionPool from the RuleDictionary: initiallizing the RuleSessioPool, and using the Decision Point API.

For information, see Introduction to the Rules SDK Decision Point API

When initializing for rule execution, once a RuleDictionary has been loaded a RuleSessionPool should be initialized with RL generated from the dictionary.

For more information, see How to Use a RuleSession Pool in the *Rules Language Reference for Oracle Business Process Management*.

If a single decision function will be invoked, then it is most efficient to load only the rule sets that are referenced by that decision function. Sample code to build the list of RL text which is passed to the RuleSessionPool constructor is shown below.

Note that error checking and exception handling not shown for brevity.

```
RuleDictionary rd;  // previously loaded dictionary
        String dfAlias = "Alias for my Decision Function";
        DecisionFunction df =
 rd.getCombinedDataModel().getDecisionFunctionByAlias(dfAlias);
        List<String> rlList = new ArrayList<String>();
        //
        // Add the RL for the data model
        //
        rlList.add(rd.dataModelRL());
        //
        // Add the RL for each rule set referenced by the decision function
        //
        Collection<String> rsal = df.getRuleSets();
        for (String alias : rsal)
        {
            rlList.add(rd.ruleSetRL(alias));
        }
```

If multiple decision functions will be invoked from the same RuleSessionPool, then adding the RL for the rulesets referenced by each decision function will enable this. Care must be taken to avoid adding the same ruleset twice.

In other scenarios such as using rules to dynamically selecting other rule sets to execute at run time, it will be necessary to load all of the rule sets in the dictionary. Sample code to do this is shown below.

```
RuleDictionary rd;  // previously loaded dictionary
        List<String> rlList = new ArrayList<String>();
        //
        // Add the RL for the data model
        //
        rlList.add(rd.dataModelRL());
        //
        // Add the RL for each rule set referenced by the decision function
        //
        Collection<String> rsal = rd.getRuleSetAliases(true);
        for (String alias : rsal)
        {
            rlList.add(rd.ruleSetRL(alias));
        }
```

Introduction to the Rules SDK Decision Point API

This section describes how to use Oracle Business Rules SDK (Rules SDK) to write applications that access, create, modify, and execute rules in Oracle Business Rules dictionaries (and work with the contents of a dictionary). It also provides a brief description of Rules SDK and shows how to work with the Rules SDK Decision Point API. The Rules SDK consists of four areas:

- Engine: provides for rules execution

- Storage: provides access to rule dictionaries and repositories
- Editing: provides a programatic way to create and modify dictionary components
- Decision Point: provides an interface to access a dictionary and execute a decision function

Other than for explanation purposes, there is not an explicit distinction between these areas in Rules SDK. For example, to edit rules you also need to use the storage area of Rules SDK to access a dictionary. These parts of the Rules SDK are divided to help describe the different modes of usage, rather than to describe distinct Rules SDK APIs.

Working with Decision Point API

The Decision Point API provides a concise way to execute rules. Most users create Oracle Business Rules artifacts, including data model elements, rules, Decision Tables, and rulesets using the Rules Designer extension to Oracle JDeveloper. Thus, most users do not need to work directly with the engine, storage, or editing parts of Rules SDK.

To work with the Rules SDK Decision Point package you need to understand three important classes:

- `DecisionPoint`: is a helper class that follows the factory design pattern to create instances of `DecisionPointInstance`. In most applications there should be one `DecisionPoint` object that is shared by all application threads. A caller uses the `getInstance()` method of `DecisionPoint` to get an instance of `DecisionPointInstance` which can be used to call the defined Decision Point.
- `DecisionPointBuilder`: follows the Builder design pattern to construct a Decision Point.
- `DecisionPointInstance`: users call `invoke()` in this class to assert facts and execute a decision function.

The `DecisionPoint` classes support a fluent interface model so that methods can be chained together. For more information, see

`http://www.martinfowler.com/bliki/FluentInterface.html`

A Decision Point manages several aspects of rule execution, including:

- Use of `oracle.rules.rl.RuleSession` objects
- Reloading of a dictionary when the dictionary is updated

To create a Decision Point in a Java application you need the following:

- Either the name of a dictionary to be loaded from an MDS repository or a pre-loaded `oracle.rules.sdk2.dictionary.RuleDictionary` instance.
- The name of a decision function stored in the specified dictionary.

How to Obtain the Car Rental Sample Application

This chapter shows a car rental application that demonstrates the use of Rules SDK and the Decision Point API. You can obtain the sample application in a ZIP file, `CarRentalApplication.zip`. This ZIP contains a complete JDeveloper application and project.

The source code for Oracle Business Rules-specific samples and SOA samples are available online in the Oracle SOA Suite samples page.

To work with the sample unzip `CarRentalApplication.zip` into an appropriate directory. The car rental application project contains a rules dictionary and several Java examples using Rules SDK.

For more examples, see Introduction to the Grades Sample Application

How to Open the Car Rental Sample Application and Project

The Car Rental sample application shows you how to work with the Rules SDK Decision Point API.

To open the car rental sample application:

1. Start Oracle JDeveloper.
2. Open the car rental application in the directory where you unzipped the sample. For example, from the File menu select **Open...** and in the Open dialog navigate to the CarRentalApplication folder.
3. In the Open dialog select **CarRentalApplication.jws** and click **Open**.
4. In the Application Navigator, expand the **CarRentalApplication**, expand **Application Sources** and **Resources**. This displays the Oracle Business Rules dictionary named `CarRental.rules` and several Java source files.

Creating a Dictionary for Use with a Decision Point

The car rental sample uses the Rules SDK Decision Point API with either a pre-loaded Oracle Business Rules dictionary or a repository stored in MDS. When you are working in a development environment you can use the Decision Point API with the pre-loaded dictionary signature. In a production environment you would typically use a Decision Point with the MDS repository signature.

The CarRental dictionary is pre-defined and is available in the car rental sample application.

To work with the Decision Point API you need to create a dictionary that contains a decision function (the car rental sample application comes with a predefined dictionary and decision function).

You perform the following steps to create a dictionary and a decision function:

- How to Create Data Model Elements for Use with a Decision Point
- How to View a Decision Function to Call from the Decision Point
- How to Create Rules or Decision Tables for the Decision Function

How to Create Data Model Elements for Use with a Decision Point

Note:

Note that the screen shots reflect a previous version, however, the content is applicable to the current release.

You need the following to add to a decision function when you create an application with a Decision Point.

- A dictionary containing data model elements that you use to create rules or Decision Tables and when working with ADF Business Components fact types, you need to add links for the Decision Point support dictionary. For more information, see Working with Data Model Elements. For more information, see Working with Oracle Business Rules and ADF Business Components.
- A dictionary containing fact definitions. For more information, see Working with Facts and Value Sets.

To view the data model in the sample application:

1. In Rules Designer, click the **Facts** navigation tab.
2. Select the **Java Facts** tab, as shown in Figure 8-1.

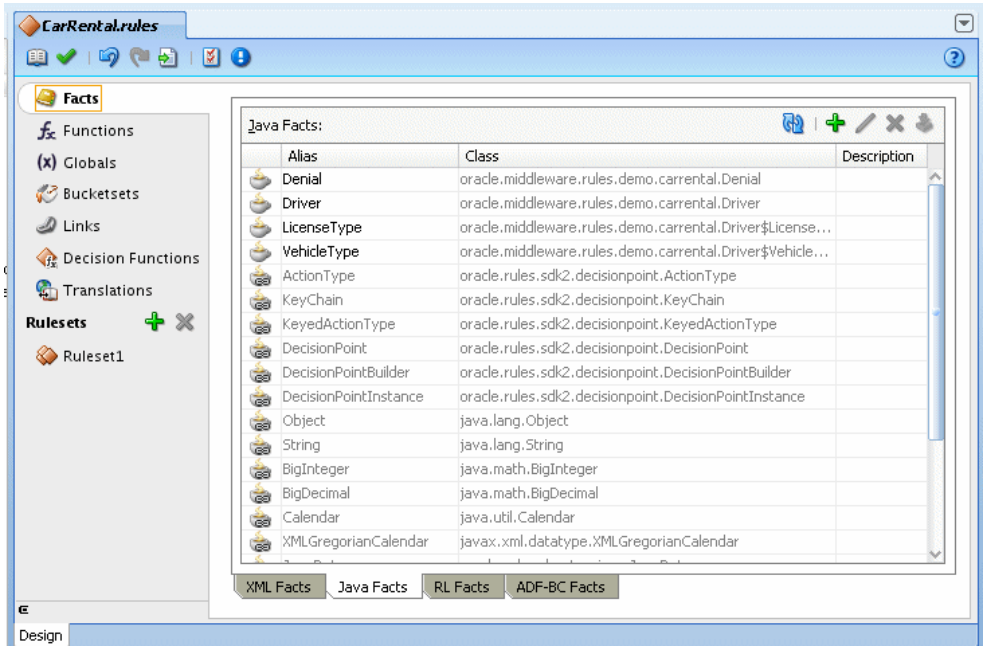
   The **Java Facts** tab shows four fact types imported, in addition to the fact types provided as built-in to the dictionary.

   The `Driver` Java Fact is imported from the `Driver` Java class in the project.

   The `Denial` Java Fact is imported from `Denial` Java class in the project.

   The `LicenseType` and `VehicleType` facts are imported from the nested `enum` classes defined in the `Driver` class.

Figure 8-1 Defined Java Facts for the Car Rental Sample Application



When you use a Decision Point with Rules SDK, you call a decision function in a specified dictionary. The decision function that you call can contain one or more rulesets that are executed as part of the Decision Point.

Similarly, to view the ruleset in the supplied car rental sample application, expand the **CarRentalApplication** in Rules Designer. In the CarRentalApplication, expand **Resources** and double-click the **CarRental.rules**

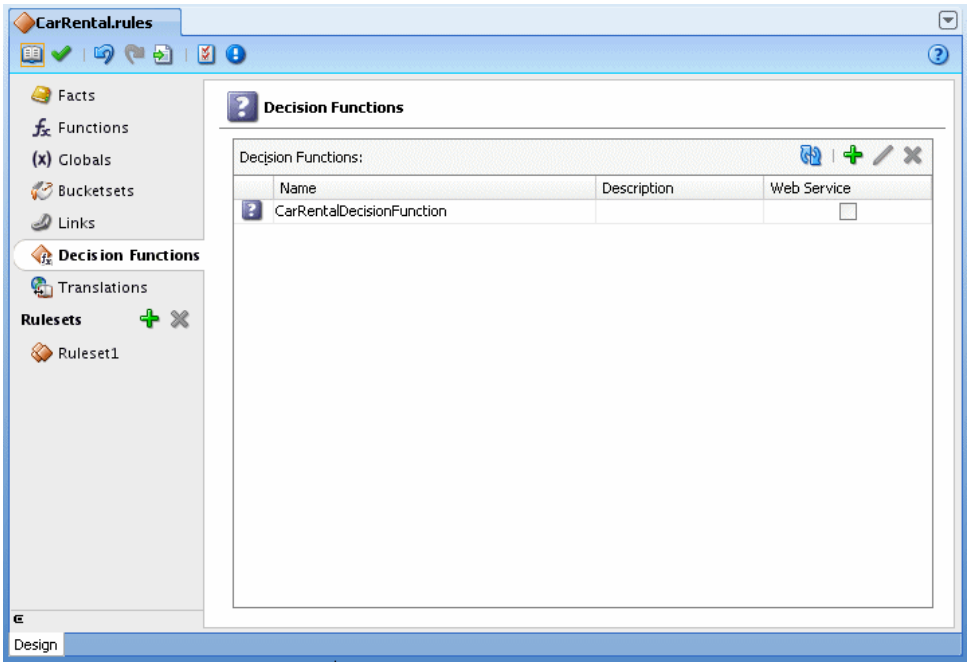
How to View a Decision Function to Call from the Decision Point

When you work with the Decision Point API you use decision functions to expose an Oracle Business Rules dictionary. For more information on decision functions, see Working with Decision Functions.

To view the decision function in the car rental sample application:

1. In Rules Designer, click the **Decision Functions** navigation tab. This displays the available decision functions in the CarRental dictionary, as shown in Figure 8-2.

Figure 8-2 Car Rental Sample Decision Function



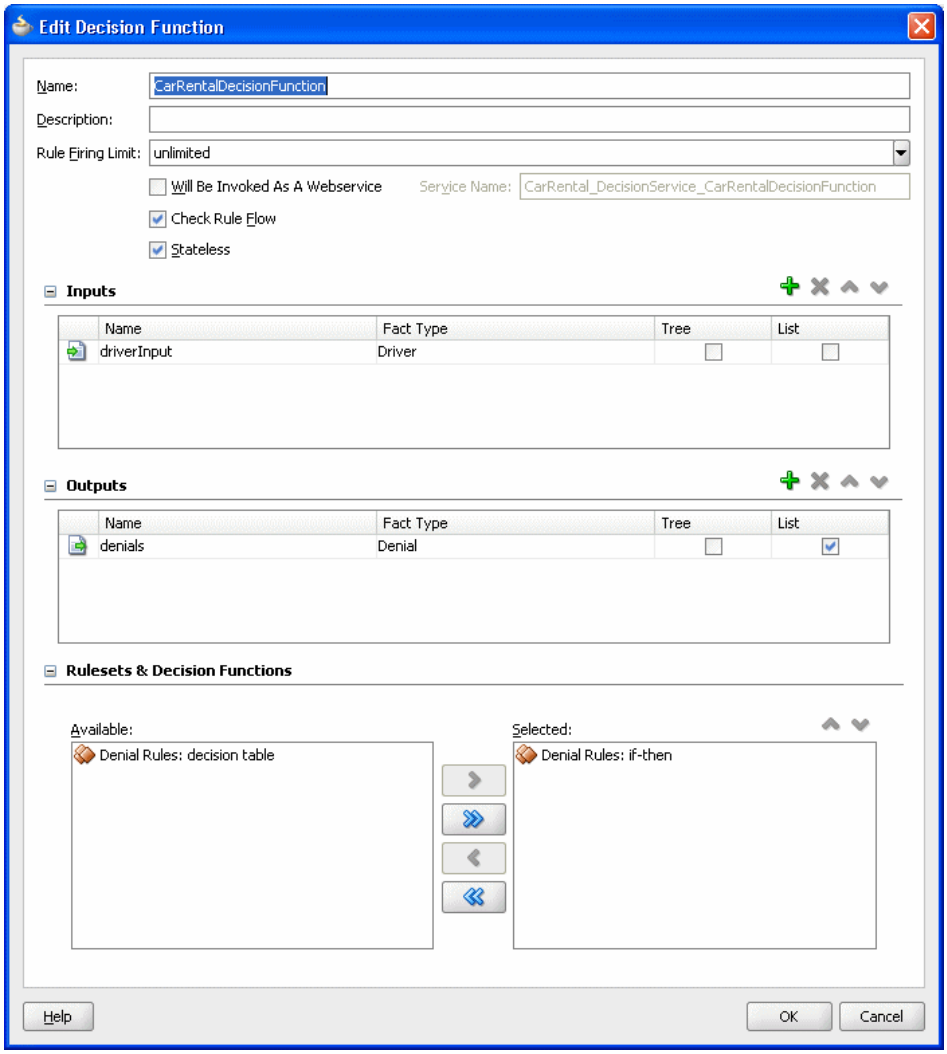
2. Select the row with CarRentalDecisionFunction and double-click the decision function icon. This opens the Edit Decision Function dialog as shown in Figure 8-3.

   The decision function **Inputs** table includes a single argument for a Driver fact type.

   The decision function **Outputs** table includes a single argument for a Denial fact type.

   The decision function **Rulesets and Decision Functions** area shows **Denial Rules:if-then** in the **Selected** box.

Figure 8-3 Car Rental Decision Function for the Car Rental Sample Application



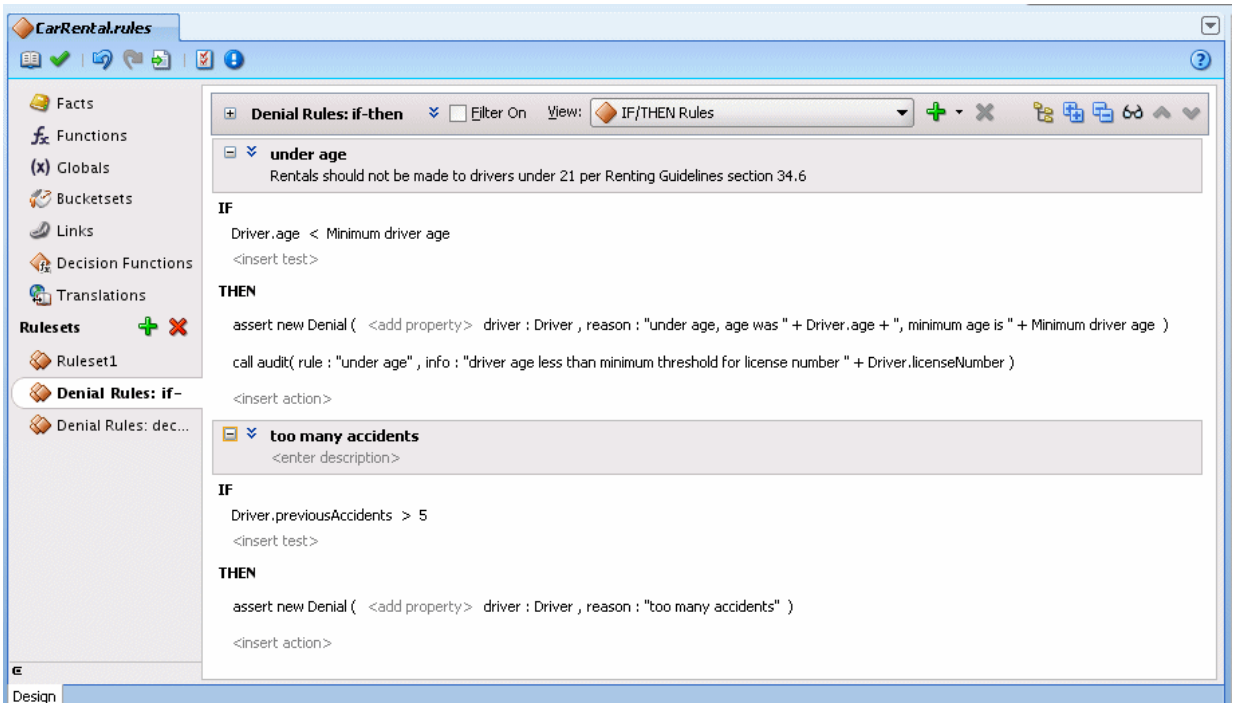
How to Create Rules or Decision Tables for the Decision Function

The car rental sample includes two rulesets, one with IF/THEN rules and another containing a Decision Table. You can use either IF/THEN rules or Decision Tables or both in your application if you are using a Decision Point.

To view the rules in the car rental sample application:

1. In Rules Designer click the **Denial Rules:if-then** ruleset, as shown in Figure 8-4.

Figure 8-4 Ruleset with IF/THEN Rules for the Car Rental Sample Application

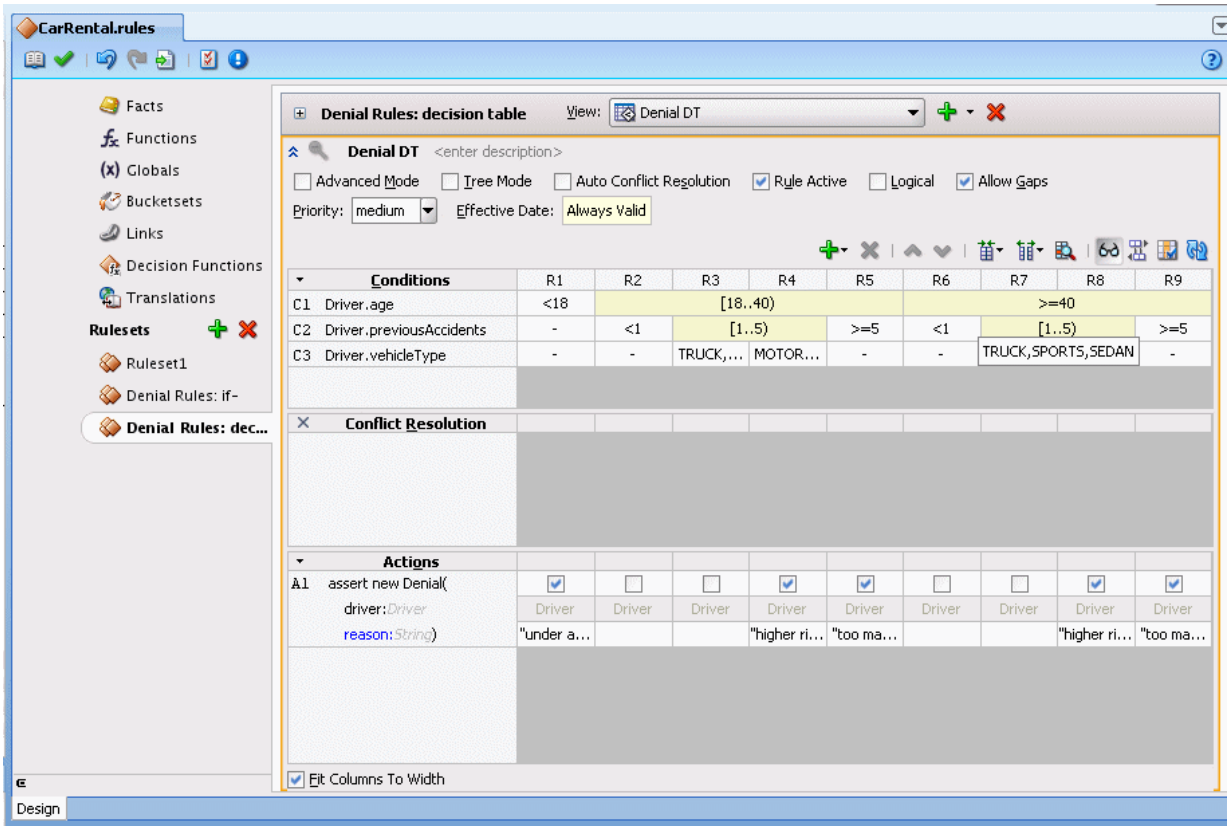


The **Denial Rules:if-then** ruleset includes two rules:

- **under age**: this rule defines the minimum age of the driver. The rule compares the `Driver` instance `age` property to the global `Minimum driver age`. If the driver is under this age, then a new `Denial` fact is asserted. A call to the decision function collects this `Denial` fact, as defined in its output.
- **too many accidents**: this rule defines an upper threshold for the number of accidents a driver can have before a rental for the driver is denied. The rule also calls a user-defined function, `audit`, to provide some auditing output about why the `Denial` is created.

2. To view the Decision Table in the car rental application, click the **Denial Rules:decision table** ruleset in the Rules Designer, as shown in Figure 8-5.

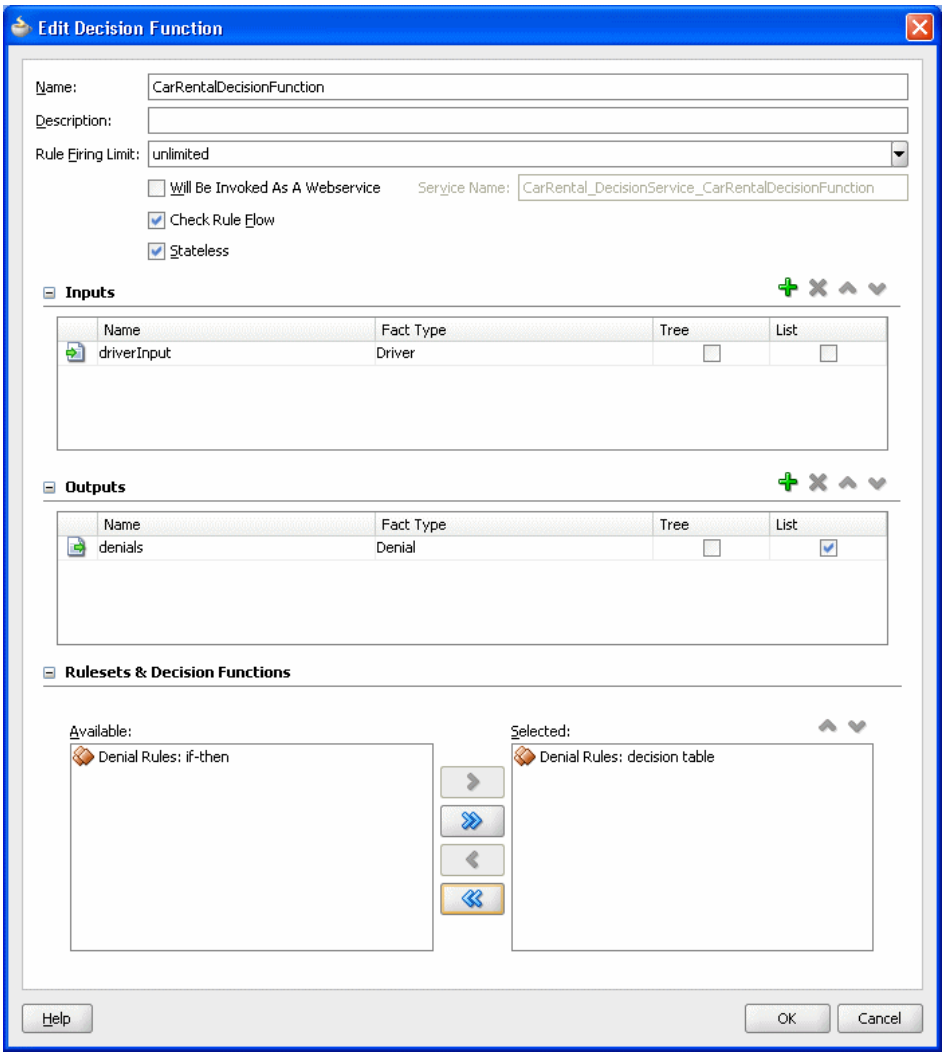
Figure 8-5 Ruleset with Decision Table for the Car Rental Sample Application



What You Need to Know About Using Car Rental Sample with a Decision Table

The car rental sample application includes the **Denial Rules: decision table** ruleset. To switch to use a Decision Table in the supplied decision function sample, move the **Denial Rules:if-then** from the **Selected** area in the decision function and add the **Denial Rules: decision table** ruleset, which uses a Decision Table to define similar rules, as shown in Figure 8-6.

Figure 8-6 Decision Function for Car Rental Sample with Decision Table Ruleset



Creating a Java Application Using Rules SDK Decision Point

When you use Rules SDK in a development environment, you have the option of using Decision Point API with a pre-loaded dictionary. In a production environment you typically use the Decision Point API with the MDS repository signature and the dictionary is stored in MDS. For more information on using a Decision Point with, see What You Need to Know About Using Decision Point in a Production Environment.

The source code for Oracle Business Rules-specific samples and SOA samples are available online in the Oracle SOA Suite samples page.

The CarRentalProject project includes the `com.example.rules.demo` package that includes the car rental sample file, `CarRentalWithDecisionPointUsingPreloadedDictionary.java`. The project also includes several `.java` source files that support different variations for using Decision Point. Table 8-1 provides a summary of the different versions of the car rental sample.

Table 8-1 Java Files in the Decision Point Sample CarRentalProject

Base Java Filename	Description
`CarRental`	This is the base class for all of the examples. It contains constant values for using the CarRental dictionary and a method `createDrivers` which creates instances of the `Driver` class.
`CarRentalWithDecisionPoint`	Contains a static attribute of type `DecisionPoint` and a method `checkDriver()` that invokes a Decision Point with a specified instance of the `Driver` class. This class includes these methods for the sample application so that both the MDS repository and pre-loaded dictionary examples can share the same `checkDriver()` implementation.
`CarRentalWithDecisionPointUsingMdsRepository`	Contains an example of creating a Decision Point that uses MDS to access and load the rule dictionary. In a production environment, most applications use the Decision Point API with MDS.
`CarRentalWithDecisionPointUsingPreloadedDictionary`	Contains an example of creating a Decision Point from an instance of the `RuleDictionary` class. This example also contains code for manually loading the dictionary to create a `RuleDictionary` instance.
`CarRentalWithRuleSession`	Contains an advanced usage of the Engine API that is documented further in the comments.
`CarRentalWithRuleSessionPool`	Contains an advanced usage of the Engine API that is documented further in the comments.
`Denial`	Contains the class that defines the `Denial` fact type used to create the rules and Decision Table.
`Driver`	Contains the class that defines the `Driver` fact type used to create the rules and Decision Table.
`DriverCheckerRunnable`	Contains the class which can be used as a thread for simulating concurrent users invoking the Decision Point.

How to Add a Decision Point Using Decision Point Builder

To use a Decision Point you create a `DecisionPoint` instance using `DecisionPointBuilder`, as shown in example below:

```
static {
    try {
        // specifying the Decision Function and a pre-loaded
        // RuleDictionary instance
        m_decisionPoint =  new DecisionPointBuilder()
                           .with(DF_NAME)
                           .with(loadRuleDictionary())
                           .build();
    } catch (SDKException e) {
        System.err.println("Failed to build Decision Point: " + e.getMessage());
        e.printStackTrace();
    }
}
```

The above example shows the `DecisionPointBuilder` supports a fluent interface pattern, so all methods can easily be chained together when you create a Decision

Point. The three most common methods for configuring the Decision Point with `DecisionPointBuilder` are overloaded to have the name `with()`. Each `with()` method takes a single argument of type `RuleDictionary`, `DictionaryFQN`, or `String`. The `DecisionPointBuilder` also supports similar set and get methods: `getDecisionFunction()`, `setDecisionFunction()`, `getDictionary()`, `setDictionary()`, `getDictionaryFQN()`, `setDictionaryFQN()`.

This chain shown in example above includes the following steps:

1. The first step is to create a `DecisionPointBuilder` instance with code such as the following:

   ```
   new DecisionPointBuilder()
   ```

2. The `with()` method using a `String` argument defines the name of the decision function that the Decision Point executes. Calling this method is mandatory.

   ```
   .with(DF_NAME)
   ```

   The `DF_NAME` specifies the name of the decision function you define for your application. For example for the sample car rental application `DF_NAME` is defined in `CarRental.java` as `CarRentalDecisionFunction`.

3. Call only one of the other two `with()` methods. In this case the sample code uses a pre-loaded Rule Dictionary instance, containing the specified decision function. The `loadDictionary()` method loads an instance of `RuleDictionary` from a file. Example 8-1 shows the `loadDictionary()` method. For more information, see How to Use a Decision Point with a Pre-loaded Dictionary.

   ```
   .with(loadRuleDictionary())
   ```

4. Call the `build()` method to construct and return a `DecisionPoint` instance.

The `DecisionPoint` instance is shared among all instances of the application, which is why it is a static attribute and created in a static block. Another way of initializing the `DecisionPoint` would be to initialize the `m_decisionPoint` attribute with a static method that created and returned a `DecisionPoint` instance.

How to Use a Decision Point with a Pre-loaded Dictionary

Example 8-1 shows the `loadRuleDictionary()` method that loads an instance of RuleDictionary from a file.

When reading or writing a dictionary directly from a file as shown in Example 8-1, ensure to set the encoding to `UTF-8`. If this is not done, Unicode characters used in the dictionary are corrupted. The `UTF-8` option must be set explicitly in the `FileInputStream` or `OutputStreamWriter` constructor. Do not use Java classes such as `FileReader` and `FileWriter`, as these classes always use the platform default encoding which is usually an ASCII variant rather than a Unicode variant.

Example 8-1 Load Rule Dictionary Method

```
private static RuleDictionary loadRuleDictionary(){
        RuleDictionary dict = null;
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(
                        new InputStreamReader(
                            new FileInputStream(
                                new File(DICT_LOCATION)), "UTF-8"));
```

```
            dict = RuleDictionary.readDictionary(reader,
                                                 new
DecisionPointDictionaryFinder(null));

            List<SDKWarning> warnings = new ArrayList<SDKWarning>();

            dict.update(warnings);
            if (warnings.size() > 0) {
                System.err.println("Validation warnings: " + warnings);
            }
        } catch (SDKException e){
            System.err.println(e);
        } catch (FileNotFoundException e){
            System.err.println(e);
        } catch (IOException e){
            System.err.println(e);
        } finally {
            if (reader != null) { try { reader.close(); } catch (IOException
ioe) {ioe.printStackTrace();}}
        }
        return dict;
    }
```

How to Use Executor Service to Run Threads with Decision Point

The car rental sample allows you to use Oracle Business Rules and simulate multiple concurrent users. The following code example shows use of the Java `ExecutorService` interface to execute multiple threads that invoke the Decision Point. The `ExecutorService` is not part of the Rules SDK Decision Point API.

```
ExecutorService exec = Executors.newCachedThreadPool();
List<Driver> drivers = createDrivers();

for (int i = 0; i < NUM_CONCURRENT; i++) {
    Driver driver = drivers.get(i % drivers.size());
    exec.execute(new DriverCheckerRunnable(driver));
}
```

The above example includes the following code for the sample application:

- Create the Executor Service:

  ```
  ExecutorService exec = Executors.newCachedThreadPool();
  ```

- Call method `createDrivers()`, defined in `CarRental.java`, to create a list of `Driver` instances.

  ```
  List<Driver> drivers = createDrivers();
  ```

- A loop through a list of `Driver` instances to fill the driver list with drivers.
- A loop to start multiple threads from `DriverCheckerRunnable` instances. These instances open a Decision Point and run the rules on each driver. For information on this code, see How to Create and Use Decision Point Instances.

The following code example shows the code that waits for the threads to complete.

```
try {
    exec.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

```
        exec.shutdown();
    }
```

How to Create and Use Decision Point Instances

The `DriverCheckerRunnable` instances call the `checkDriver()` method. Example 8-2 shows the `checkDriver()` method that is defined in `CarRentalWithDecisionPoint`. The `checkDriver()` method handles invoking Decision Point with a `Driver` instance.

Example 8-2 shows the following:

- Getting a `DecisionPointInstance` from the static `DecisionPoint` defined with the `DecisionPointBuilder`, with the following code.

  ```
  DecisionPointInstance instance = m_decisionPoint.getInstance();
  ```

- Add inputs according to the signature of the decision function associated with the Decision Point. This defines one argument of type `List` as the input. This `List` contains the `Driver` instances:

  ```
  instance.setInputs(new ArrayList<Object>() {
          {
              add(driver);
          }
      });
  ```

- Invoke the Decision Point and store the return value. The return type follows the same pattern as the decision function which is being called in the Decision Point.

  ```
  List<Object> outputs = instance.invoke();
  ```

  In this case the `invoke()` returns a `List` of length one, containing a `List` of `Denial` instances.

- If the return is a `List` of any other size than one, then this is an error:

  ```
  if (outputs.isEmpty())
    System.err.println("Oops, no results");
  ```

- The first entry that is returned from the Decision Point is cast to a List of type `List<Denial>`:

  ```
  java.util.List<Denial> denials =
      (java.util.List<Denial>)outputs.get(0);
  ```

- If the denials list is empty, then no `Denial` instances were asserted by the rules. This indicates that it is OK to rent a car to the driver. Otherwise, print the reasons why the driver rental was rejected:

  ```
  if (denials.isEmpty()) {
      System.out.println("Rental is allowed for " +
                         driver.getName());
  } else {
      for (Denial denial : denials) {
          System.out.println("Rental is denied for " +
                             denial.getDriver().getName() +
                             " because " + denial.getReason());
      }
  }
  ```

Sample Code to Create a Decision Point Instance with getInstance()

The `DriverCheckerRunnable` instances call the `checkDriver()` method. Example 8-2 shows the `checkDriver()` method that is defined in `CarRentalWithDecisionPoint`. The `checkDriver()` method handles invoking Decision Point with a `Driver` instance.

Example 8-2 Code to Create a Decision Point Instance with getInstance()

```
public class CarRentalWithDecisionPoint extends CarRental {

    protected static DecisionPoint m_decisionPoint;

    public static void checkDriver(final Driver driver) {
        try {
            DecisionPointInstance instance = m_decisionPoint.getInstance();
            instance.setInputs(new ArrayList<Object>() {
                    {
                        add(driver);
                    }
                });
            List<Object> outputs = instance.invoke();

            if (outputs.isEmpty())
                System.err.println("Oops, no results");

            java.util.List<Denial> denials =
                (java.util.List<Denial>)outputs.get(0);
            if (denials.isEmpty()) {
                System.out.println("Rental is allowed for " +
                                   driver.getName());
            } else {
                for (Denial denial : denials) {
                    System.out.println("Rental is denied for " +
                                       denial.getDriver().getName() +
                                       " because " + denial.getReason());
                }
            }
        } catch (RLException e) {
            e.printStackTrace();
        } catch (SDKException e) {
            e.printStackTrace();
        }
    }

}
```

Running the Car Rental Sample

In the car rental sample installed on your system, for the code shown in Example 8-1, modify the value of `DICT_LOCATION` to match the location of the dictionary on your system.

To run the car rental sample on your system:

1. In the Application Navigator, select the dictionary and from the **Edit** menu select **Copy Path**.
2. In the `CarRental.java` file, paste the path value into the `DICT_LOCATION` value.

3. In the **CarRentalProject** select the **CarRentalWithDecisionPointUsingPreloadedDictionary.java** file.

4. Right-click and in the list select **Run**.

Sample Output from Car Rental

Example 8-3 shows sample output from car rental application.

Example 8-3 Output from Car Rental Sample

```
Rental is allowed for Carol
Rental is allowed for Alice
Rental is allowed for Alice
Rental is allowed for Carol
Rental is denied for Bob because under age, age was 15, minimum age is 21
Mar 13, 2009 11:18:00 AM oracle.rules.rl.exceptions.LogWriter flush
INFO: Fired: under age because driver age less than minimum threshold for license
number d222
Mar 13, 2009 11:18:00 AM oracle.rules.rl.exceptions.LogWriter flush
INFO: Fired: under age because driver age less than minimum threshold for license
number d222
Rental is denied for Bob because under age, age was 15, minimum age is 21
Rental is allowed for Alice
Rental is allowed for Eve
```

What You Need to Know About Using Decision Point in a Production Environment

In a production environment you can use an MDS repository to store Oracle Business Rules dictionaries. When you use an MDS repository to store the dictionary, the steps shown in How to Add a Decision Point Using Decision Point Builder and How to Use a Decision Point with a Pre-loaded Dictionary change to access the dictionary. The `CarRentalWithDecisionPointUsingMdsRepository` shows sample code for using Decision Point with MDS.

To see a complete example with deployment steps showing the use of a Decision Point to access a dictionary in MDS, see Adding a Servlet with Rules SDK Calls for Grades Sample Application.

The following code example shows the use of `DictionaryFQN` with `DecisionPointBuilder` to access a dictionary in an MDS repository. The complete example is shown in the sample code in `CarRentalWithDecisionPointUsingMdsRepository`.

```
static {
    try {
        // specifying the Decision Function and Dictionary FQN
        // loads the rules from the MDS repository.
        m_decisionPoint = new DecisionPointBuilder()
                            .with(DF_NAME)
                            .with(DICT_FQN)
                            .build();
    } catch (SDKException e) {
        System.err.println("Failed to build Decision Point: " +
                           e.getMessage());
```

Similar to the steps in section How to Add a Decision Point Using Decision Point Builder, the above example shows the following:

1. The first step is to create a `DecisionPointBuilder` instance with.

   ```
   new DecisionPointBuilder()
   ```

2. The `with()` method using a `String` argument defines the name of the decision function that the Decision Point executes. Calling this method is mandatory.

   ```
   .with(DF_NAME)
   ```

   The `DF_NAME` specifies the name of the decision function you define for your application. For example for the car rental application this is defined in `CarRental.java` a `CarRentalDecisionFunction`.

3. Call only one of the other two `with()` methods. In this case the sample code calls a `DictionaryFQN` to access an MDS repository. The code example in step 4 shows the routing that uses the dictionary package and the dictionary name to create the `DictionaryFQN`.

   ```
   .with(DICT_FQN)
   ```

4. Call the `build()` method to construct and return a `DecisionPoint` instance.

```
protected static final String DICT_PKG = "com.example.rules.demo";
protected static final String DICT_NAME = "CarRental";

protected static final DictionaryFQN DICT_FQN =
    new DictionaryFQN(DICT_PKG, DICT_NAME);
protected static final String DF_NAME = "CarRentalDecisionFunction";
```

What You Need to Know About Decision Point and Decision Tracing

The Rules SDK API contains methods to assist with processing a decision trace. These methods process a decision trace to replace the RL names used in the trace with the aliases used in the associated dictionary. This makes the decision trace naming consistent with the naming used in the Oracle Business Rules dictionary.

The basic API for processing a decision trace requires a `RuleDictionary` object and a `DecisionTrace` object:

```
RuleDictionary dict = ...;
DecisionTrace trace = ...;
dict.processDecisionTrace(trace);
```

This code shows the processing call that converts the naming in the decision trace to use the same names, with aliases, as in the dictionary.

The Rules SDK Decision Point API contains methods that allow you configure decision tracing and retrieve the resulting trace when you invoke a decision point.

Table 8-2 shows the Decision Point API methods for setting decision trace options.

Table 8-2 Decision Point Decision Tracing Methods

Method	Description
`decisionTrace`	Get the decision trace produced from the call to invoke. Returns `DecisionTrace`

Table 8-2 (Cont.) Decision Point Decision Tracing Methods

Method	Description

`getDecisionTraceLevel`	Get the decision trace level to be used by the RuleSession. This value defaults to `DECISION_TRACE_OFF`, which means no trace information is gathered. Possible values are: `DECISION_TRACE_OFF`
	`DECISION_TRACE_DEVELOPMENT`
	`DECISION_TRACE_PRODUCTION`
	Return Type: String

`getDecisionTraceLimit`	Get the decision trace limit, or maximum number of trace elements which are retrieved for the trace. Return Type: int

`setDecisionTraceLevel`	Set the decision trace level to be used by the RuleSession. This parameter value is a String. Possible values are: `DECISION_TRACE_OFF`
	`DECISION_TRACE_DEVELOPMENT`
	`DECISION_TRACE_PRODUCTION`

`setDecisionTraceLimit`	Set the decision trace limit, or maximum number of trace elements which are retrieved for the trace.

Sample Usage of Decision Tracing

Example 8-4 shows sample usage of decision tracing with DecisionPoint API.

For more information on decision tracing, see Tracing Rule Execution in Fusion Middleware Control Console in *Administering Oracle SOA Suite and Oracle Business Process Management Suite*.

Example 8-4 Using Decision Trace from Decision Point API

```
DecisionPoint dp = new DecisionPointBuilder()
  .with(new DictionaryFQN("com.foo", "Bar"))
  .with("MyDecisionFunction")
  .setDecisionTraceLevel(DecisionPointBuilder.DECISION_TRACE_DEVELOPMENT)
  .setDecisionTraceLimit(24000)
  .build();

...

DecisionPointInstance dpi = dp.getInstance();

dpi.invoke();

DecisionTrace trace = dpi.decisionTrace();
```

9

Creating a Rule-enabled Non-SOA Java EE Application

This chapter describes how to use Oracle JDeveloper to create a rule-enabled non-SOA Java Enterprise Edition (EE) application. It also shows a sample application, a Java Servlet, which runs as a Java EE application using Oracle Business Rules (this describes using Oracle Business Rules without a SOA composite).

The chapter includes the following sections:

The source code for Oracle Business Rules-specific samples and SOA samples are available online in the Oracle SOA Suite samples page.

Introduction to the Grades Sample Application

The Grades application provides a sample use of Oracle Business Rules in a Java Servlet. The servlet uses Rules SDK Decision Point API.

This sample demonstrates the following:

- Creating rules in an Oracle Business Rules dictionary using an XSD schema that defines the input and the output data, and the facts for the data model. In this case you provide the XSD schema in the file `grades.xsd`.
- Creating a servlet that uses Oracle Business Rules to determine a grade for each test score that is input.
- Creating a test page to supply input test scores and to submit the data to the grades servlet.
- Deploying the application, running it, submitting test values, and seeing the output.

There is another example--for more information, see How to Open the Car Rental Sample Application and Project.

Creating an Application and a Project for Grades Sample Application

You can create Grades sample application by following the steps below.

Note:

The screen shots reflect a previous version, however, the content is applicable to the current release.

To create the application and the project for the grades sample application, do the following:

- Create a Fusion Web Application (ADF)
- Create a project in the application
- Add the schema to define the inputs, outputs, and the objects for the data model
- Create an Oracle Business Rules dictionary in the project

How to Create a Fusion Web Application for the Grades Sample Application

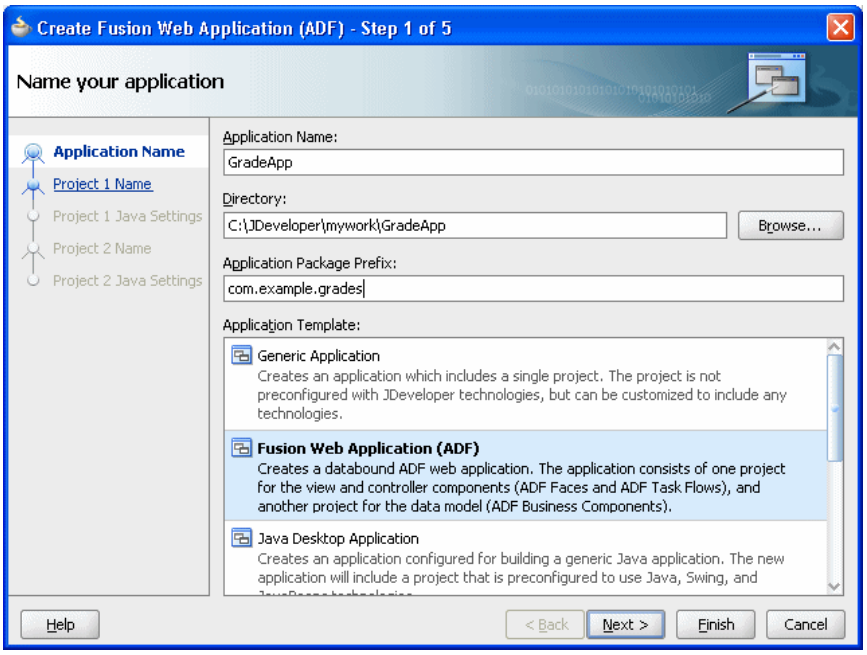
To work with Oracle Business Rules and create a Java EE application, you first need to create the application in Oracle JDeveloper.

To create a fusion web application (ADF) for grades:

1. Create an application. You can do this in the Application Navigator by selecting **New Application...**, or from the **Application** menu list by selecting **New Application...**.
2. In the Name your application dialog enter the application options, as shown in Figure 9-1:
   a. In the **Application Template** area, select **Fusion Web Application**.
   b. In the **Application Name** field, enter an application name. For example, enter `GradeApp`.
   c. In the **Directory** field, specify a directory name or accept the default.
   d. In the **Application Package Prefix** field, enter an application package prefix. For example, `com.example.grades`.

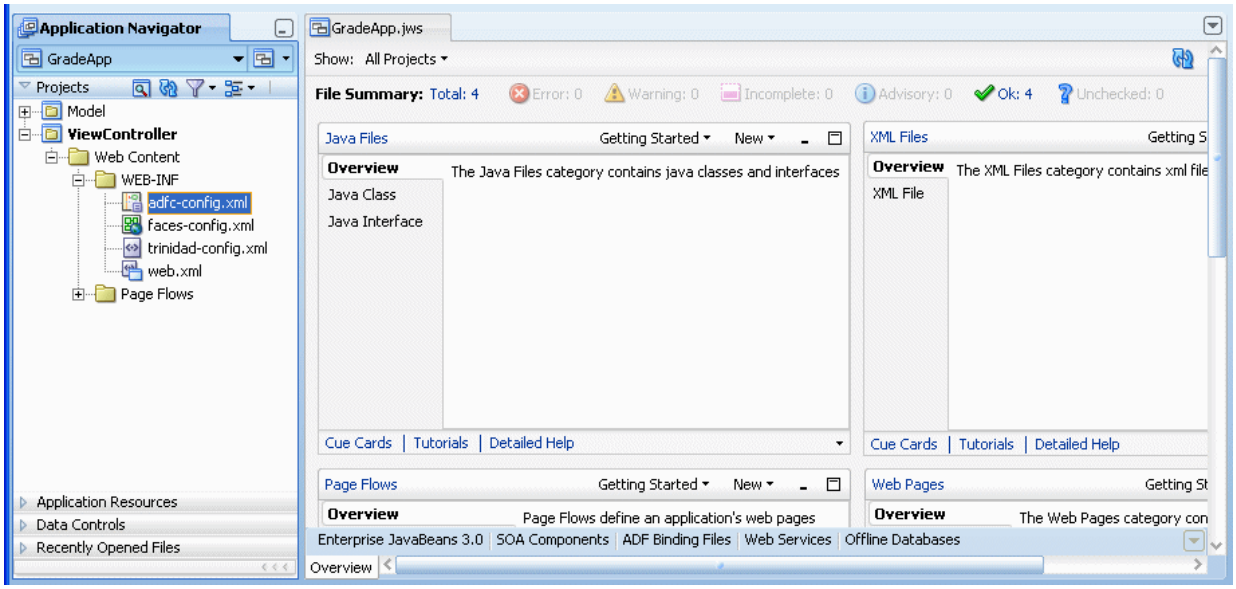
      The prefix, followed by a period applies to objects created in the initial project of an application.

Figure 9-1 Adding GradeApp Application



3. Click **Finish**. After creating the application Oracle JDeveloper displays the file summary, as shown in Figure 9-2.

Figure 9-2 New Grades Application Named GradeApp



How to Develop Accessible ADF Faces Pages

Oracle software implements the standards of the Web Content Accessibility Guidelines (WCAG) 1.0 Level AA using an interpretation of the standards at `http://www.oracle.com/accessibility/standards.html`

ADF Faces user interface components have built-in accessibility support for visually and physically impaired users. User agents such as a web browser rendering to nonvisual media such as a screen reader can read component text descriptions to

provide useful information to impaired users. Access key support provides an alternative method to access components and links using only the keyboard. ADF Faces accessibility audit rules provide direction to create accessible images, tables, frames, forms, error messages and popup windows using accessible HTML markup.

For information on how to develop accessible ADF Faces pages, see Developing Accessible ADF Faces Pages in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

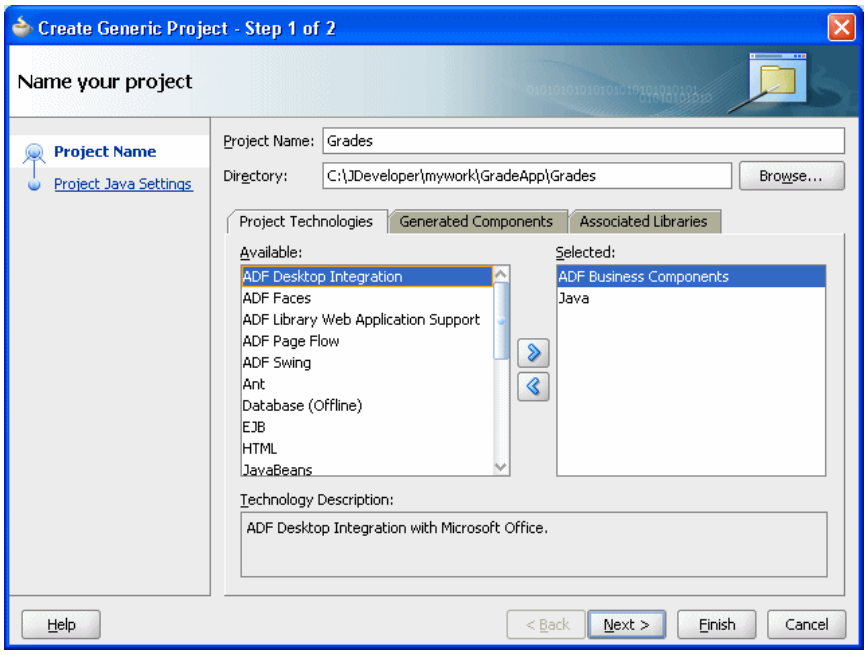
How to Create the Grades Project

In the Grades sample application you do not use the Model or ViewController projects. You create a project in the application for the grades sample project.

To create a grades project:

1. In the GradeApp application, in the Application Navigator, from the Application Menu select **New Project...**.
2. In the New Gallery, in the **Items** area select **Generic Project**.
3. Click **OK**.
4. In the Name your project page enter the values as shown in Figure 9-3:
   a. In the **Project Name** field, enter a name. For example, enter `Grades`.
   b. Enter or browse for a directory name, or accept the default.
   c. Select the **Project Technologies** tab.
   d. In the **Available** area double-click **ADF Business Components** to move this item to the **Selected** area. This also adds Java to the **Selected** area as shown in Figure 9-3.

Figure 9-3 Adding Generic Project to the Grades Application



5. Click **Finish**. This adds the Grades project.

How to Add the XML Schema and Generate JAXB Classes in the Grades Project

To create the Grades sample application you need to use the `grades.xsd` file, as shown in example `grades.xsd` schema below. You can create and store the schema file locally and then use Oracle JDeveloper to copy the file to your project.

```
<?xml version= '1.0' encoding= 'UTF-8' ?>
<xs:schema targetNamespace="http://example.com/grades"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://example.com/grades"
        attributeFormDefault="qualified" elementFormDefault="qualified"
        xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:extensionBindingPrefixes="xjc"
        jaxb:version="2.0">

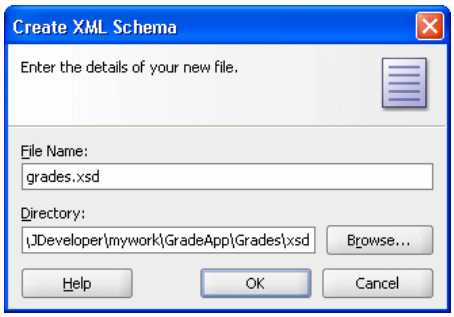
    <xs:element name="TestScore">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="testName" type="xs:string"/>
                <xs:element name="testScore" type="xs:double"/>
                <xs:element name="testCurve" type="xs:double"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="TestGrade">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="grade" type="tns:Grade"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:simpleType name="Grade">
        <xs:restriction base="xs:string">
            <xs:enumeration value="A"/>
            <xs:enumeration value="B"/>
            <xs:enumeration value="C"/>
            <xs:enumeration value="D"/>
            <xs:enumeration value="F"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

To add the XML schema to the grades project:

1. Save the schema file as shown in example `grades.xsd` schema to a local file named `grades.xsd`.
2. In the Application Navigator select the **Grades** project.
3. Right-click and in the context menu select **New...**.
4. In the New Gallery select the **All Technologies** tab.
5. In the **Categories** area, expand **General** and select **XML**.
6. In the **Items** area, select **XML Schema**. Click **OK**.

7. In the Create XML Schema dialog, in the **File Name** field enter `grades.xsd`.
8. In the Create XML Schema dialog, in the **Directory** field add the `xsd` directory to the Grades project path name, as shown in Figure 9-4.

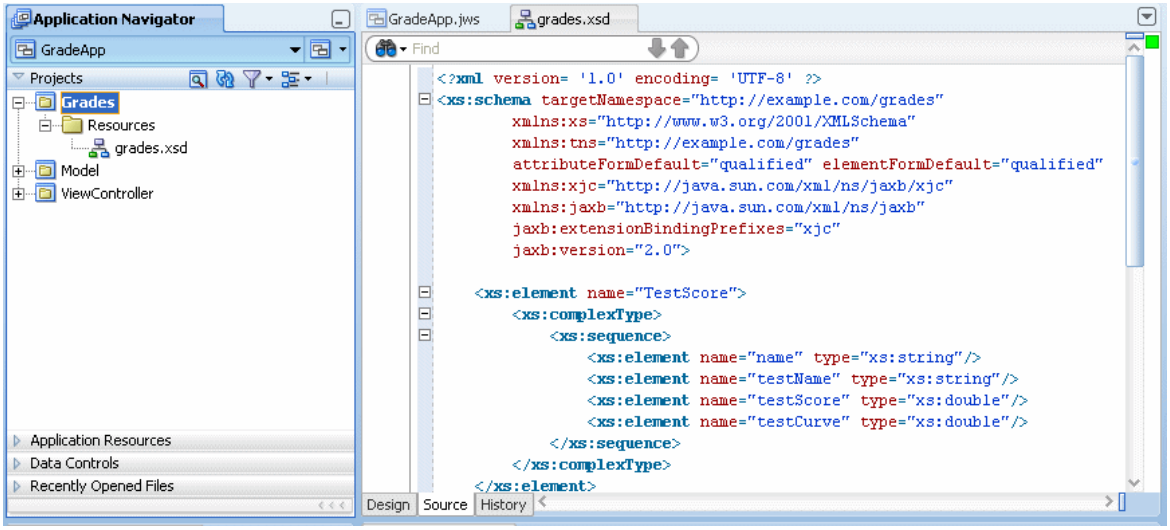
Figure 9-4 Adding Schema to Grades Project in xsd Directory



Click **OK**.

9. In the `grades.xsd` file, select the **Source** tab.
10. Copy the schema shown in example above to the `grades.xsd` in the Grades project, as shown in Figure 9-5.

Figure 9-5 Shows the Grades.xsd Schema File in the Grades Project



How to generate JAXB 2.0 content model from grades schema

To generate JAXB 2.0 content model from grades schema:

1. In the Application Navigator, in the **Grades** project expand **Resources** and select **grades.xsd**.
2. Right-click and in the context menu select **Generate JAXB 2.0 Content Model**.
3. In the JAXB 2.0 Content Model from XML Schema dialog, click **OK**.

How to Create an Oracle Business Rules Dictionary in the Grades Project

After creating a project in Oracle JDeveloper create business rules within the Grades project.

To use business rules with Oracle JDeveloper, you do the following:

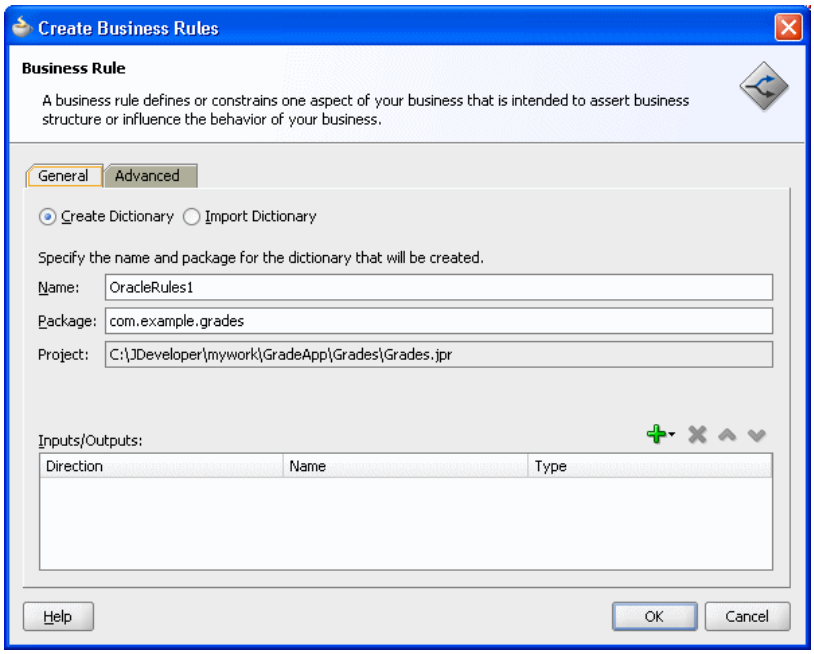
- Add a business rule to the project and import `grades.xsd` schema
- Create input and output variables
- Create an Oracle Business Rules dictionary in the project

To create a business rules dictionary in the business tier:

1. In the Application Navigator, select the Grades project.
2. Right-click and in the context menu select **New...**. and select the **All Technologies** tab.
3. In the New Gallery, in the **Categories** area, expand Business Tier and select **Business Rules**.
4. In the New Gallery, in the **Items** area, select **Business Rules**. Click **OK**.

   Oracle JDeveloper displays the Create Business Rules dialog, as shown in Figure 9-6.

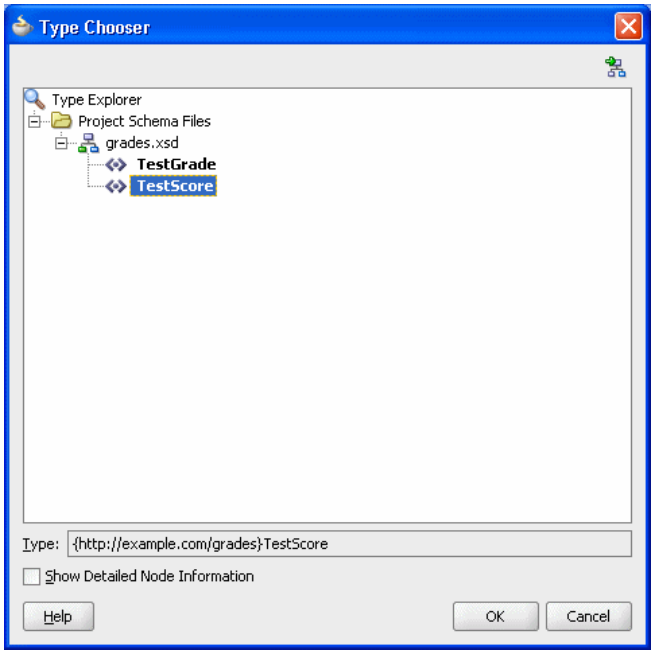
   Figure 9-6 Adding a Business Rule to Grades with the Create Business Rules Dialog



5. In the **Name** field, enter a name to name the dictionary. For example, enter `GradingRules`.
6. To add an input, from the list next to the **Add** button select **Input...**.

7. In the Type Chooser, expand the Project Schemas Files folder and expand `grades.xsd`. Select the input **TestScore**, as shown in Figure 9-7.

Figure 9-7 Shows the Type Chooser Dialog with TestScore Element

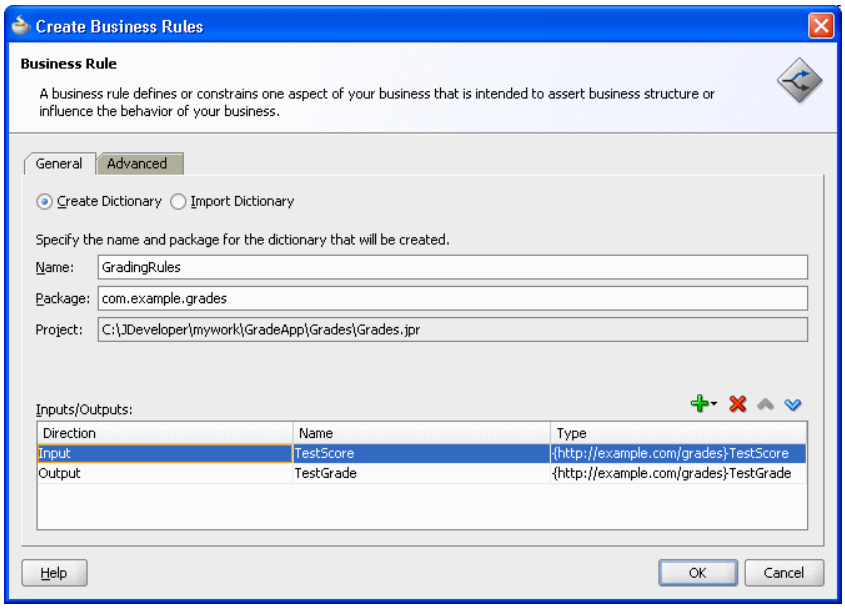


8. On the Type Chooser window, click **OK**. This displays the Create Business Rules dialog.

9. In the Create Business Rules dialog, in a similar manner to the input add the output by selecting **Output...** to add the output element `TestGrade` from the `grades.xsd` schema.

   The resulting Create Business Rules dialog is shown in Figure 9-8.

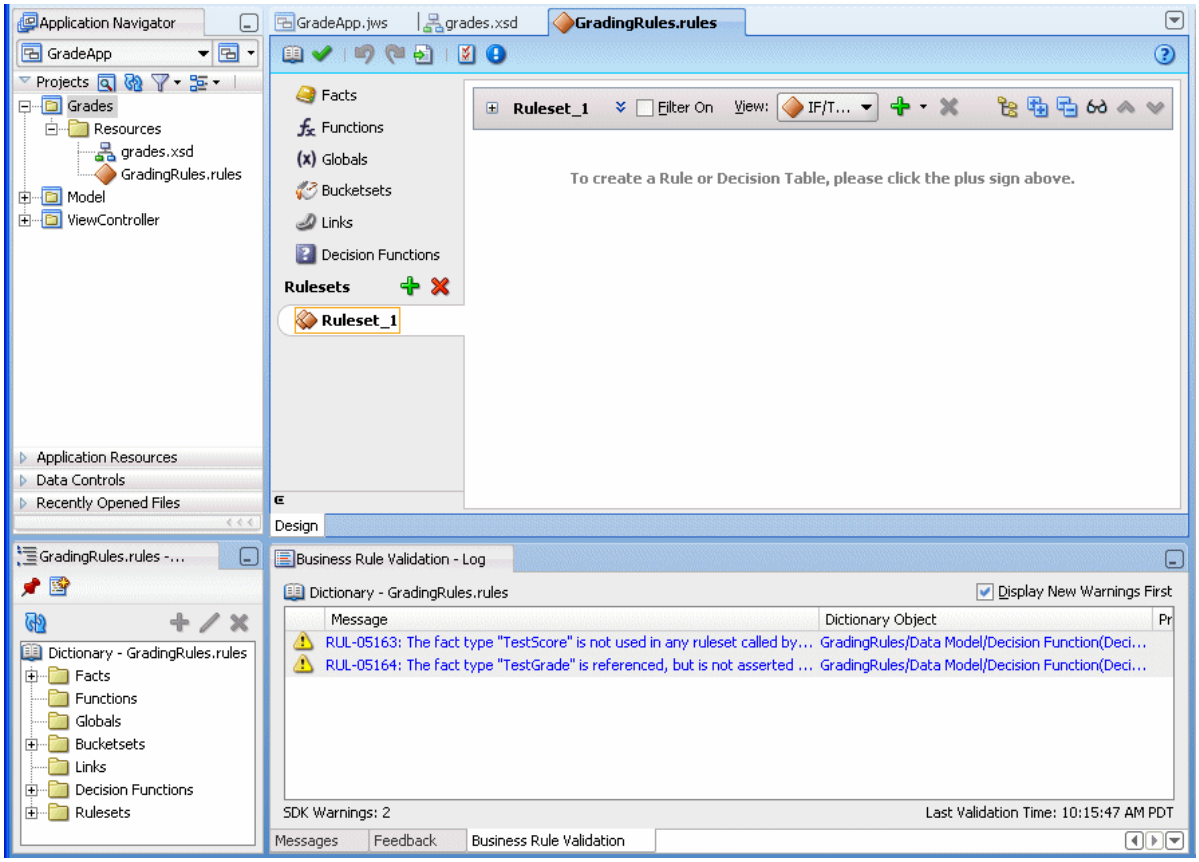
Figure 9-8 Create Business Rules Dialog with Grades Input and Output



Click **OK**. Oracle JDeveloper creates the GradingRules dictionary as shown in Figure 9-9.

10. In the **File** menu, select **Save All** to save your work.

Figure 9-9 Shows the New Grading Rules Dictionary



Note that the business rule validation log area for the new dictionary shows several validation warnings. These validation warning messages are cleared as you modify the dictionary in later steps.

Creating Data Model Elements and Rules for the Grades Sample Application

Create data model elements and rules for the grades sample application by following the steps below.

To create the data model and the business rules for the Grades sample application, do the following:

- Create value sets for grades
- Create rules by adding a Decision Table for grades
- Split the Decision Table and add actions for rules
- Rename the default decision function

How to Create Value Sets for Grades Sample Application

In this example you associate a value set with a fact type. This supports using a Decision Table where you need value sets that specify how to draw values for each cell in the Decision Table (for the conditions in the Decision Table).

To create the value set for the grades sample application:

1. In Rules Designer, select the **Value Sets** navigation tab.
2. From the list next to the **Create Value Set...** button, select **List of Ranges**.
3. For the value set, double-click in the Name field to select the default name.
4. Enter `Grade Scale`, and press **Enter** to accept the value set name.
5. In the **Value Set** table, double-click the icon for the Grade Scale value set to display the **Edit Value Set** dialog.
6. In the **Edit Value Set** dialog, click **Add** to add a value and click **Add** three times to add three more values.
7. In the **Endpoint** field, enter 90 for the top endpoint and press **Enter** to accept the new value.
8. For the next value, in the **Endpoint** field enter 80 and press **Enter** to accept the new value. Similarly, for the next two values enter values in the **Endpoint** field, values 70 and 60.
9. In the **Included Endpoint** field for each value select each check box.
10. Modify the **Alias** field for each value to enter the values A, B, C, D, and F, for each corresponding range, (press **Enter** after you add each alias).

How to Associate a Value Set with a Fact Property

To prepare for creating Decision Tables you can associate a global value set with fact properties in the data model. In this way condition cells in a Decision Table **Conditions** area can use the value set when you create a Decision Table.

To associate a value set with a fact property:

1. In Rules Designer, select the **Facts** navigation tab.
2. In the **Facts** navigation tab select the **XML Facts** tab.
3. Double-click the **XML fact** icon for the **TestScore** fact. This displays the Edit XML Fact dialog.
4. In the Edit XML Fact dialog select the **testScore** property.
5. In the **Value Set** field, from the list select **Grade Scale**.
6. Click **OK**.

How to Add a Decision Table for Grades Sample Application

You create rules in a Decision Table to process input facts and to produce output facts, or to produce intermediate conclusions that Oracle Business Rules can further process using additional rules or in another Decision Table.

To use a Decision Table for rules in this application you work with facts representing a test score. Then, you use a Decision Table to create rules based on the test score to produce a grade.

To add a decision table for Grades application:

1. In Rules Designer, select **Ruleset_1** under the **Rulesets** list.
2. In **Ruleset_1**, click **Create** from the Decision Table area on the **Overview** tab. This creates **DecisionTable_1**. You can ignore the warning messages shown in the Business Rule Validation log area. You remove these warning messages in later steps.
3. In the Decision Table, **DecisionTable_1**, click the **Add** button and from the list select **Condition**.
4. In the Decision Table, double-click **<edit condition>**. Then, in the variables navigator expand **TestScore** and select `testScore`. This enters the expression `TestScore.testScore` for condition C1.

If you view the rules validation log, you should see warning messages. You remove these warning messages as you modify the Decision Table in later steps.

How to Add an Action to a Decision Table

To add an action to a decision table:

You add an action to the Decision Table to assert a new Grade fact.

1. In the Decision Table, click the **Add** button and from the list select **Action** and select **Assert New**.
2. In the **Actions** area, double-click **Assert New**.

   This displays the Action Editor dialog.
3. In the Action Editor dialog, in the **Facts** area select **TestGrade**.
4. In the Action Editor dialog, in the Properties table for the property **grade**, select the **Parameterized** check box and the **Constant** check box.

   This specifies that each rule independently sets the grade.
5. In the Action Editor dialog select the **Always Selected** check box.
6. In the Action Editor dialog click **OK**.
7. Select **Save All** from the **File** main menu to save your work.

Next you add rules to the Decision Table and specify an action for each rule.

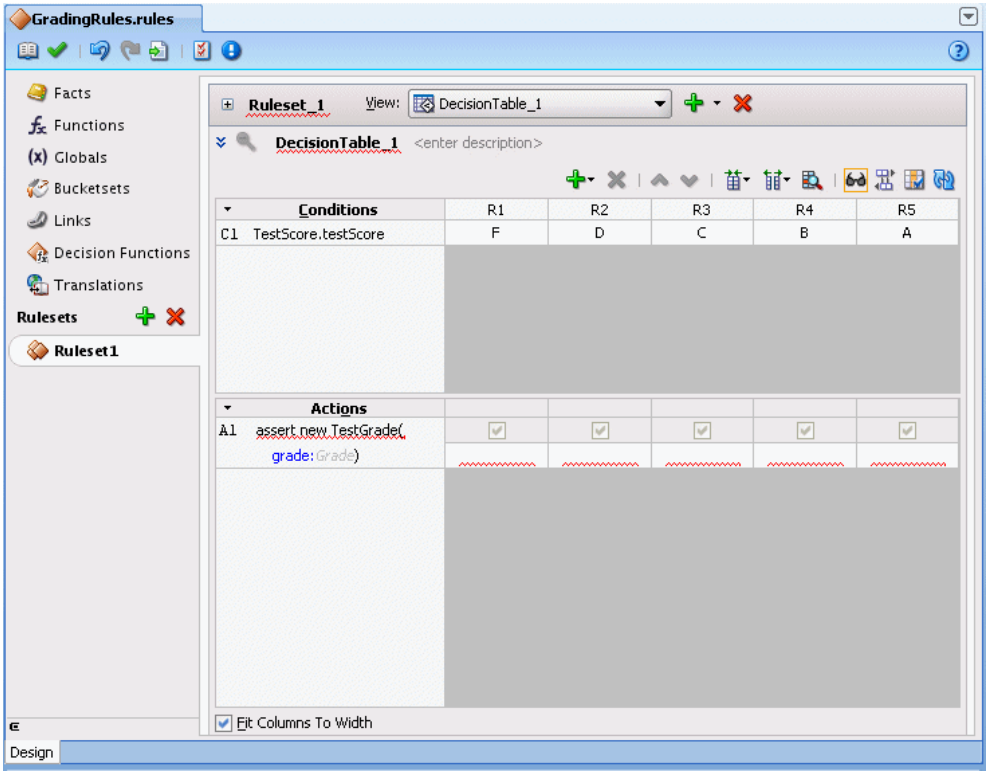
How to Add Rules in the Decision Table for Grades Sample Application

You can use the Decision Table split operation to create rules for the value set associated with the conditions row in the Decision Table. This creates one rule for every value.

To split the decision table, from the Decision Table, click the **Split Table** button from the list select **Split Table**. The split operation eliminates the "do not care" cells from the table. The table now shows five rules that cover all ranges, as shown in Figure 9-10.

These steps produce validation warnings for action cells with missing expressions. You fix these problems in later steps when you define actions for each rule.

Figure 9-10 Splitting a Decision Table Using Split Table Operation for Grades

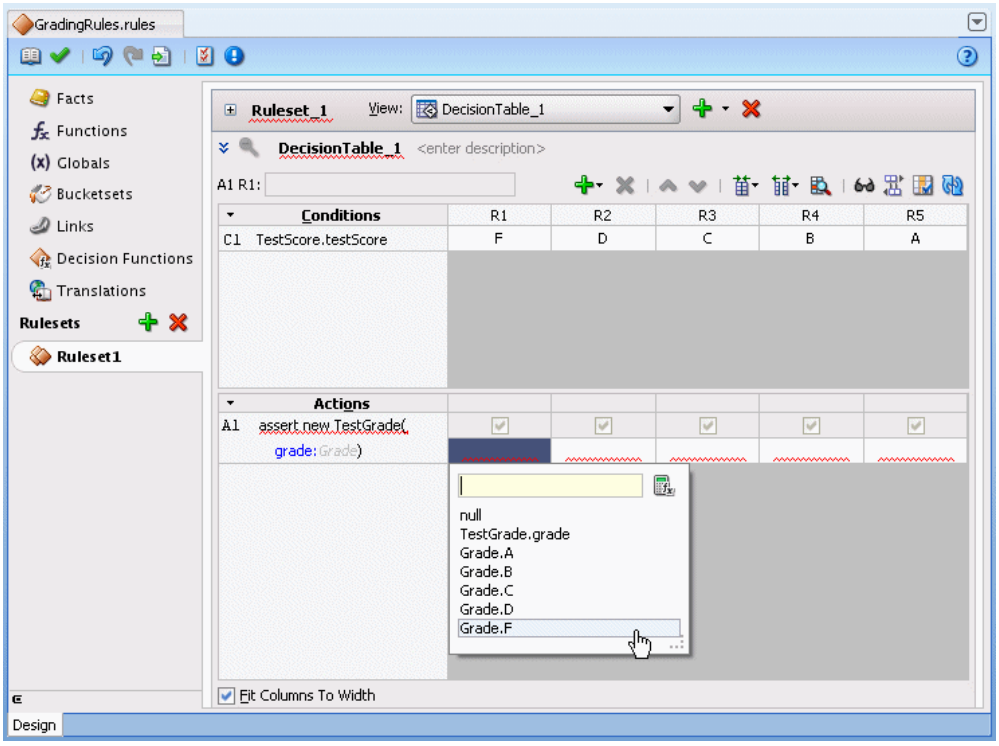


To add actions for each rule in the decision table:

In the Decision Table you specify a value for the result, a grade property, associated with TestGrade for each action cell in the **Actions** area. The possible choices for each grade property are the valid grades. In this step you fill in a value for each of the rules. The values you enter correspond to the conditions that form each rule in the Decision Table.

1. In the **Actions** area, double-click the action cell for rule **R1** as shown in Figure 9-11.

Figure 9-11 Adding Action Cell Values to Grades Decision Table



2. In the list select the corresponding value for the action cell. For example, select **Grade.F**.

3. For each of the remaining action cells select the appropriate value for **TestScore**: D, C, B, and A.

How to Rename the Decision Function for Grades Sample Application

The name you specify when you use a decision function with a Rules SDK Decision Point must match the name of a decision function in the dictionary. To make the name match, you can rename the decision function to any name you like. Thus, for this example you rename the default decision function to use the name `GradesDecisionFunction`.

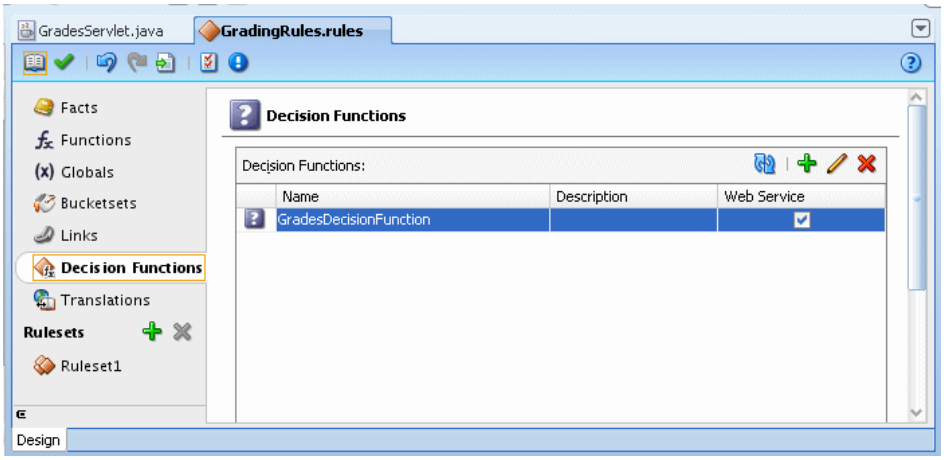
To rename the decision function:

1. In the Application Navigator, in the **Grades** project, expand the **Resources** folder and double-click the dictionary **GradingRules.rules**.

2. Select the **Decision Functions** navigation tab.

3. In the **Name** field in the **Decision Functions** table edit the decision function name to enter the value `GradesDecisionFunction`, and then press **Enter**, as shown in Figure 9-12.

Figure 9-12 Renaming Decision Function in Rules Designer



Adding a Servlet with Rules SDK Calls for Grades Sample Application

The Grades sample application includes a servlet that uses the Rules Engine.

To add this servlet with Oracle Business Rules you need to understand the important Rules SDK methods. Thus, to use the Oracle Business Rules dictionary you created with Rules Designer, you do the following:

- Create initialization steps that you perform one time in the servlet `init` routine.
- Create a servlet `service` routine using the Rules SDK Decision Point API.
- Perform steps to add the servlet code in the project.

For more information on Rules SDK Decision Point API, see Working with Rules in Standalone (Non SOA/BPM) Scenarios.

How to Add a Servlet to the Grades Project

You add a servlet to the grades project using the Create HTTP Servlet wizard.

To add a servlet to the Grades project with Oracle JDeveloper:

1. In the Application Navigator, select the **Grades** project.
2. Right-click the **Grades** project and in the context menu select **New...**.
3. In the New Gallery, select the **All Technologies** tab.
4. In the New Gallery, in the **Categories** area expand **Web Tier** and select **Servlets**.
5. In the New Gallery, in the **Items** area select **HTTP Servlet**. Click **OK**.

   Oracle JDeveloper displays the Create HTTP Servlet Welcome page.
6. Click **Next**.

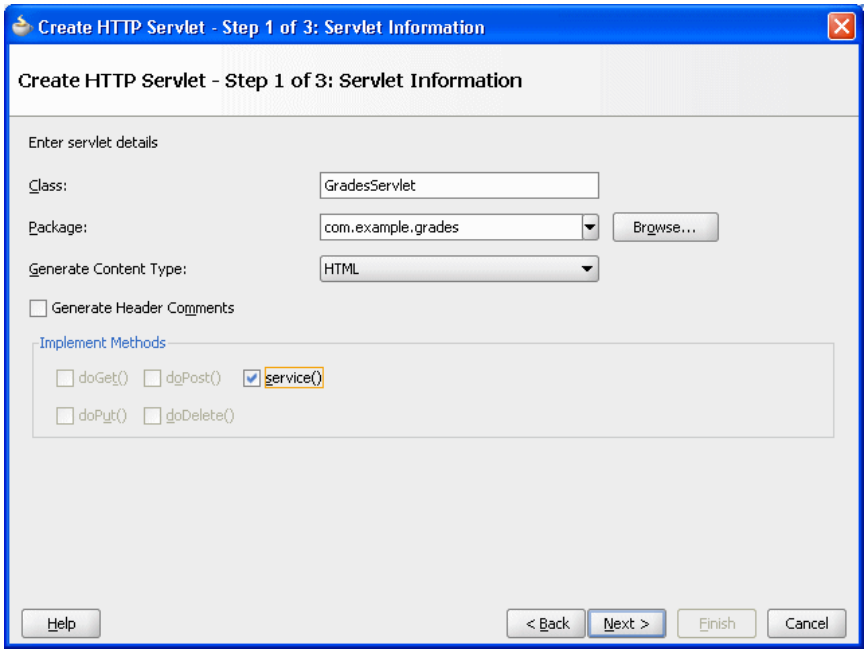
   This displays the Web Application page.
7. Select **Servlet 2.5\JSP 2.1 (Java EE 1.5)** and click **Next**.

   This displays the Create HTTP Servlet - Step 1 of 3: Servlet Information page.

8. Enter values in Create HTTP Servlet - Step 1 of 3: Servlet Information page, as follows, and as shown in Figure 9-13.

   - **Class**: `GradesServlet`
   - **Package**: `com.example.grades`
   - **Generate Content Type**: `HTML`
   - **Generate Header Comments**: `unchecked`
   - **Implement Methods**: **service()** checked and all other check boxes unchecked

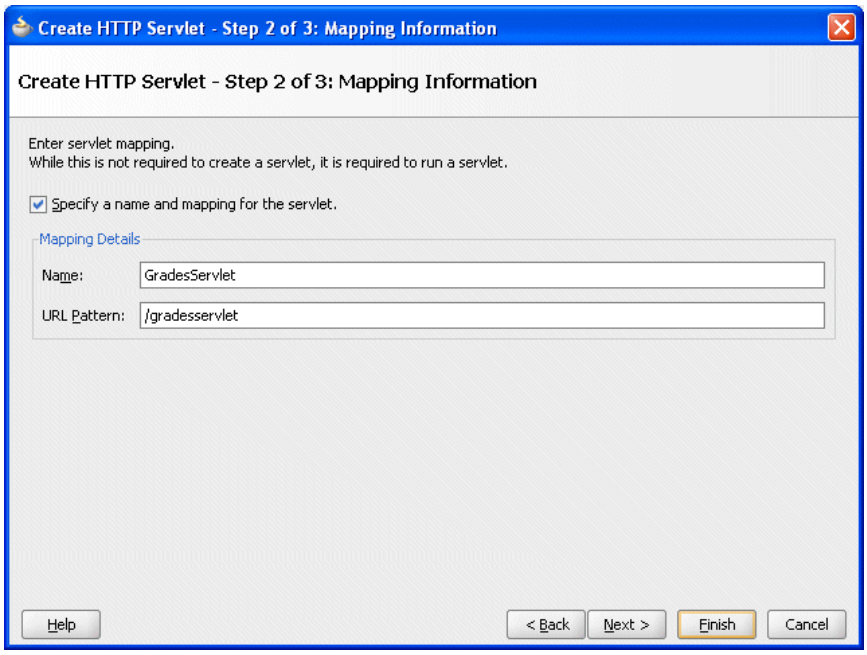
Figure 9-13 Create HTTP Servlet Wizard - Step 1 of 3: Servlet Information



Click **Next**.

This displays the Create HTTP Servlet: Step 2 of 3: Mapping Information dialog as shown in Figure 9-14.

Figure 9-14 Create HTTP Servlet Wizard - Step 2 of 3: Mapping Information



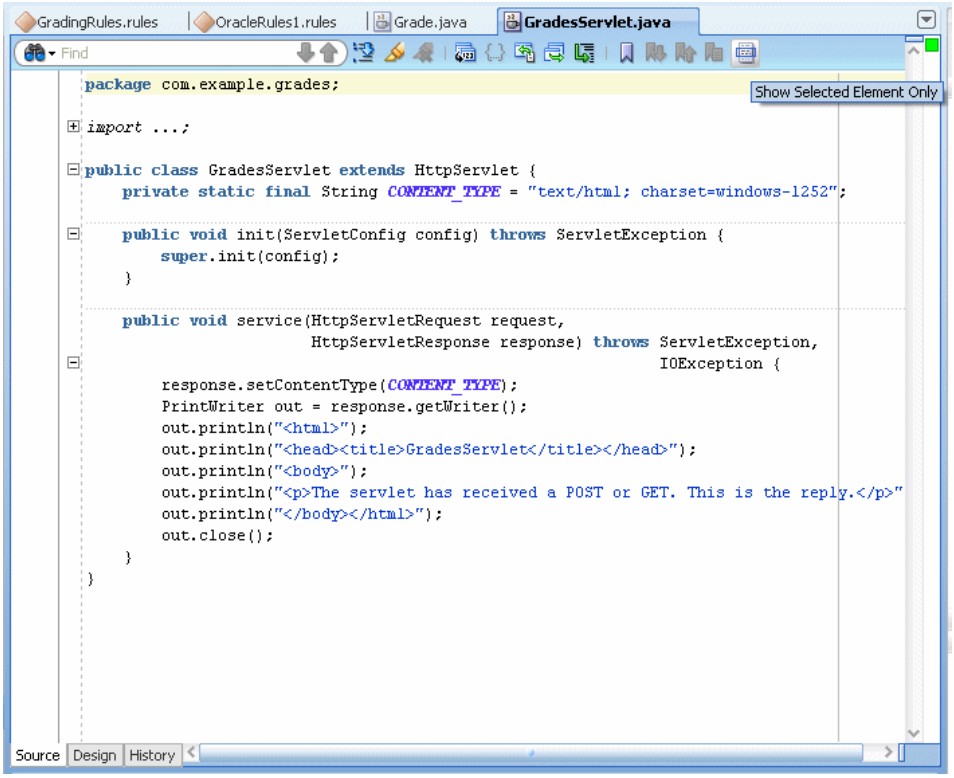
9. Configure this dialog as follows:

   - **Name**: `GradesServlet`
   - **URL Pattern**: `/gradesservlet`

   Click **Finish** when done.

   JDeveloper adds a Web Content folder to the project and creates a `GradesServlet.java` file and opens the file in the editor as shown in Figure 9-15.

Figure 9-15 Generated GradesServlet.java



10. Replace the generated servlet with the source as shown in the grades application example below.

```
package com.example.grades;

import java.io.IOException;
import java.io.PrintWriter;

import java.util.ArrayList;
import java.util.List;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import oracle.rules.rl.exceptions.RLException;
import oracle.rules.sdk2.decisionpoint.DecisionPoint;
import oracle.rules.sdk2.decisionpoint.DecisionPointBuilder;
import oracle.rules.sdk2.decisionpoint.DecisionPointInstance;
import oracle.rules.sdk2.exception.SDKException;
import oracle.rules.sdk2.repository.DictionaryFQN;

public class GradesServlet extends HttpServlet {

    private static final String CONTENT_TYPE = "text/html";
    private static final String DICT_PKG = "com.example.grades";
    private static final String DICT_NAME = "GradingRules";
    private static final DictionaryFQN DICT_FQN =
        new DictionaryFQN(DICT_PKG, DICT_NAME);
    private static final String DF_NAME = "GradesDecisionFunction";

    private DecisionPoint m_decisionPoint = null; // init in init()
```

```
public void init(ServletConfig config) throws ServletException {
    super.init(config);

    try {

        // specifying the Decision Function and Dictionary FQN
        // load the rules from the MDS repository.
        m_decisionPoint = new DecisionPointBuilder()
                          .with(DF_NAME)
                          .with(DICT_FQN)
                          .build();
    } catch (SDKException e) {
        System.err.println("Failed to build Decision Point: " +
                           e.getMessage());
        throw new ServletException(e);
    }
}

public void service(HttpServletRequest request,
                    HttpServletResponse response) throws ServletException,
                                                         IOException {
    // retrieve parameters
    String name = request.getParameter("name");
    String strScore = request.getParameter("testScore");

    // open output document
    StringBuilder doc = new StringBuilder();
    addHeader(doc);

    // create TestScore object to assert
    final TestScore testScore = new TestScore();
    testScore.setName(name);

    try {
        testScore.setTestScore(Integer.parseInt(strScore));
    } catch (NumberFormatException e){ /* use default val */ }

    // get DecisionPointInstance for invocation
    DecisionPointInstance point = m_decisionPoint.getInstance();

    // set input parameters
    point.setInputs(new ArrayList() {{ add(testScore); }});

    // invoke decision point and get result value
    TestGrade testGrade = null;
    try {

        // invoke the decision point with our inputs
        List<Object> result = point.invoke();
        if (result.size() != 1){
            error(doc, testScore.getName(), "bad result", null);
        }
        // decision function returns a single TestGrade object
        testGrade = (TestGrade)result.get(0);
    } catch (RLException e) {
        error(doc, testScore.getName(), "RLException occurred: ", e);
    } catch (SDKException e) {
        error(doc, testScore.getName(), "SDKException occurred", e);
    }

    if (testGrade != null){
        // create output table in document
        openTable(doc);
        addRow(doc, testScore.getName(), strScore, testGrade.getGrade());
        closeTable(doc);
    }

    addFooter(doc);
```

```
        // write document
        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();
        out.println(doc);
        out.close();
    }


    public static void addHeader(StringBuilder doc) {
        doc.append("<html>");
        doc.append("<head><title>GradesServlet</title></head>");
        doc.append("<body>");
        doc.append("<h1>Test Results</h1>");
    }

    public static void addFooter(StringBuilder doc) {
        doc.append("</body></html>");
    }

    public static void openTable(StringBuilder doc) {
        doc.append("<table border=\"1\"");
        doc.append("<tr>");
        doc.append("<th>Name</th>");
        doc.append("<th>Score</th>");
        doc.append("<th>Grade</th>");
        doc.append("</tr>");
    }

    public static void closeTable(StringBuilder doc) {
        doc.append("</table>");
    }

    public static void addRow(StringBuilder doc, String name, String score, Grade grade){
        doc.append("<tr>");
        doc.append("<td>"+ name +"</td>");
        doc.append("<td>"+ score +"</td>");
        doc.append("<td>"+ grade.value() +"</td>");
        doc.append("</tr>");
    }

    public static void error(StringBuilder doc, String name, String msg, Throwable t){
        doc.append("<tr>");
        doc.append("<td>"+ name +"</td>");
        doc.append("<td colspan=2>"+ msg + " " + t +"</td>");
        doc.append("</tr>");
    }
}
```

The above example includes a Oracle Business Rules Decision Point, that uses an MDS repository to access the dictionary. For more information, see What You Need to Know About Using Decision Point in a Production Environment.

When you add the Servlet as shown in the grades application example, note the following:

- In the `init()` method the servlet uses the Rules SDK Decision Point API for Oracle Business Rules. For more information on using the Decision Point API, see Working with Rules in Standalone (Non SOA/BPM) Scenarios.
- The `DecisionPointBuilder()` requires arguments including a decision function name and, in a production environment a dictionary FQN to access a dictionary in an MDS repository, as shown:

```
m_decisionPoint = new DecisionPointBuilder()
                      .with(DF_NAME)
                      .with(DICT_FQN)
```

For more information on using the Decision Point API, see Working with Rules in Standalone (Non SOA/BPM) Scenarios.

Adding an HTML Test Page for Grades Sample Application

Add an HTML test page for the grades application by following the steps below.

The Grades sample application includes an HTML test page that you use to invoke the servlet you created in Adding a Servlet with Rules SDK Calls for Grades Sample Application.

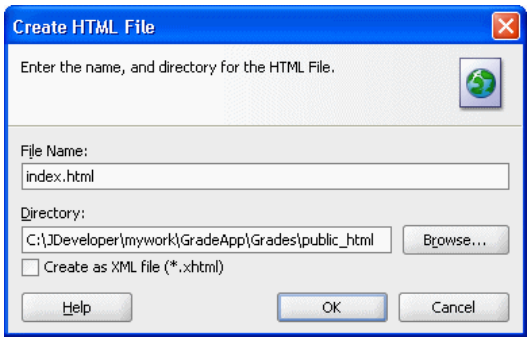
To add an HTML page to the servlet you use the Create HTML File wizard.

To add an HTML test page:

1. In the Application Navigator, in the **Grades** project select the **Web Content** folder.
2. Right-click the **Web Content** folder project and in the context menu select **New...**.
3. In the New Gallery, select the **All Technologies** tab.
4. In the New Gallery, in the **Categories** area expand **Web Tier** and select **HTML**.
5. In the New Gallery, in the **Items** area select **HTML Page**. Click **OK.**

   Oracle JDeveloper displays the Create HTML File dialog.
6. Configure this dialog as follows and as shown in Figure 9-16:
   - **File Name**: `index.html`
   - **Directory**: `C:\JDeveloper\mywork\GradeApp\Grades\public_html`

   Figure 9-16 Create HTML File Dialog

   Click **OK**.

   JDeveloper adds `index.html` to the Web Content folder and opens the editor.
7. In the editor for `index.html`, select the **Source** tab.
8. Copy and paste the HTML code from the HTML test page example below to replace the contents of the `index.html` file. Note that in the form element action attribute uses the URL Pattern you specified in Figure 9-14.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"></meta>
```

```
    <title>Test Grade Example Servlet</title>
  </head>
  <body>
    <form name="names_and_scores"
          method="post"
          action="/grades/gradesservlet" >
          <p>Name: <input type="text" name="name" /></p>
          <p>Test Score: <input type="text" name="testScore"/></p>
       <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

9. Select **Save All** from the **File** main menu to save your work.

Preparing the Grades Sample Application for Deployment

Business rules are deployed as part of the application for which you create a deployment profile in Oracle JDeveloper.

You deploy the application to Oracle WebLogic Server.

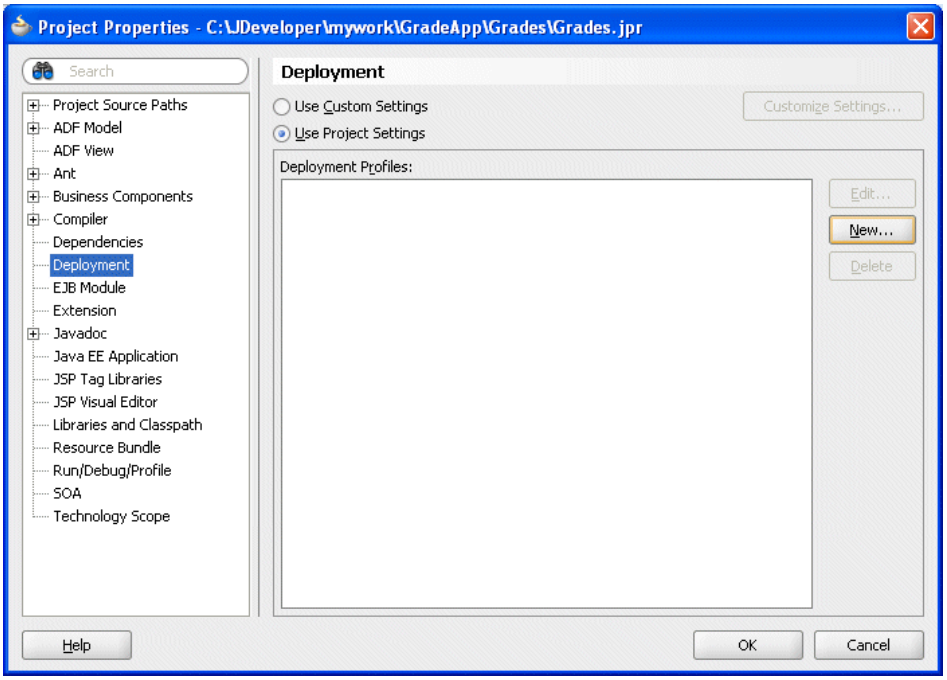
How to Create the WAR File for the Grades Sample Application

You deploy the GradeApp sample application using JDeveloper with Oracle WebLogic Server.

To create the WAR file for the grades sample application:

1. In the Application Navigator, select the **Grades** project.
2. Right-click the **Grades** project and in the context menu select **Project Properties...**. This displays the Project Properties dialog for the project.
3. In the Project Properties navigator, select the **Deployment** item as shown in Figure 9-17.

Figure 9-17 Project Properties - Deployment



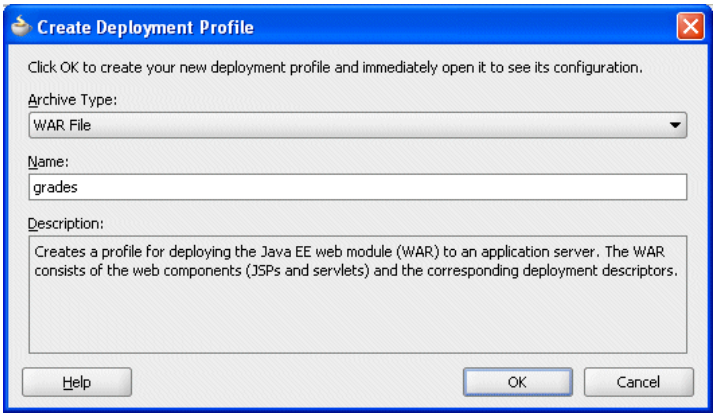
4. In the Project Properties dialog, click **New...**.

   This displays the Create Deployment Profile dialog.

5. In the Create Deployment Profile dialog, in the **Archive Type** list, select **WAR File**.

6. In the Create Deployment Profile dialog, in the **Name** field enter `grades`, as shown in Figure 9-18. Note the **Name** value uses the package value that you specified in the `form` element `action` attribute in step 8 of Adding an HTML Test Page for Grades Sample Application.

   Figure 9-18 Create Deployment Profile Dialog for WAR File



7. Click **OK**.

   This displays the Edit WAR Deployment Profile Properties dialog.

8. In the Edit War Deployment Profile Properties dialog, select **General** and configure the General page as follows, as shown in Figure 9-19:

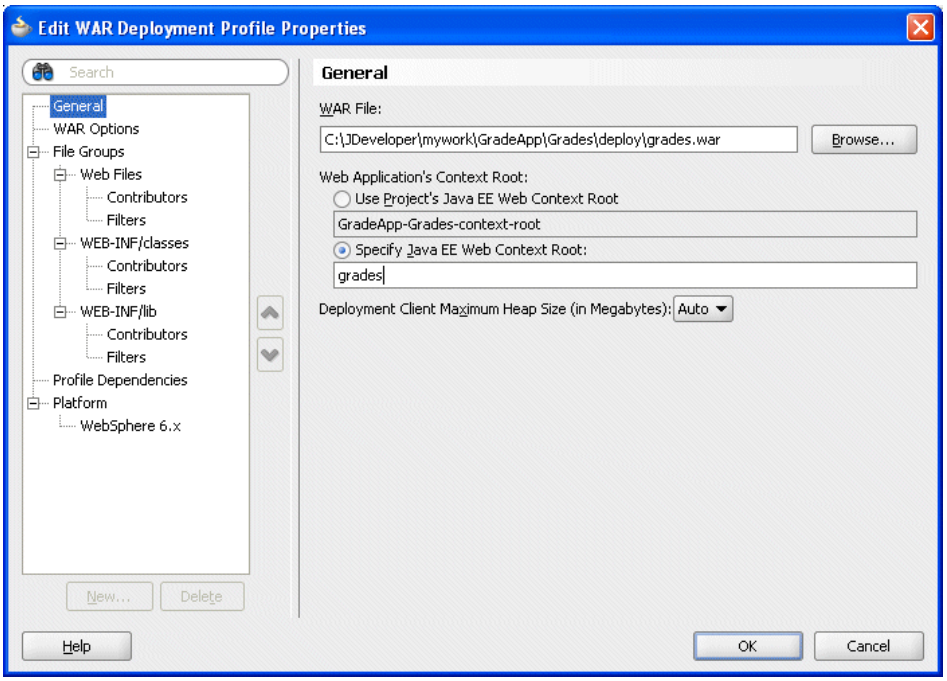
   a. Set the **WAR File**: `C:\JDeveloper\mywork\GradeApp\Grades\deploy\grades.war`

   b. In the **Web Application Context Root** area, select **Specify Java EE Web Context Root**:

   c. In the **Specify Java EE Web Context Root**: text entry area, enter `grades`.

   d. In the **Deployment Client Maximum Heap Size (in Megabytes)**: list select **Auto**

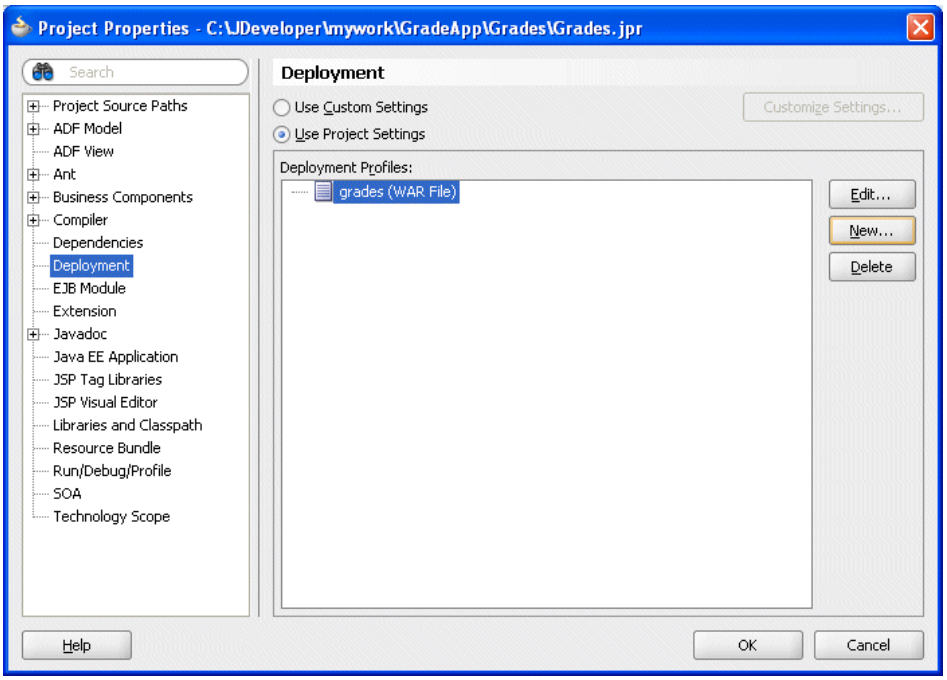
Figure 9-19 Edit WAR Deployment Properties - General



9. In the Edit WAR Deployment Profile Properties dialog, click **OK**.

JDeveloper creates a deployment profile named `grades (WAR File)` as shown in Figure 9-20.

Figure 9-20 Project Properties - Deployment Profile Created



10. In the Project Properties dialog, click **OK**.

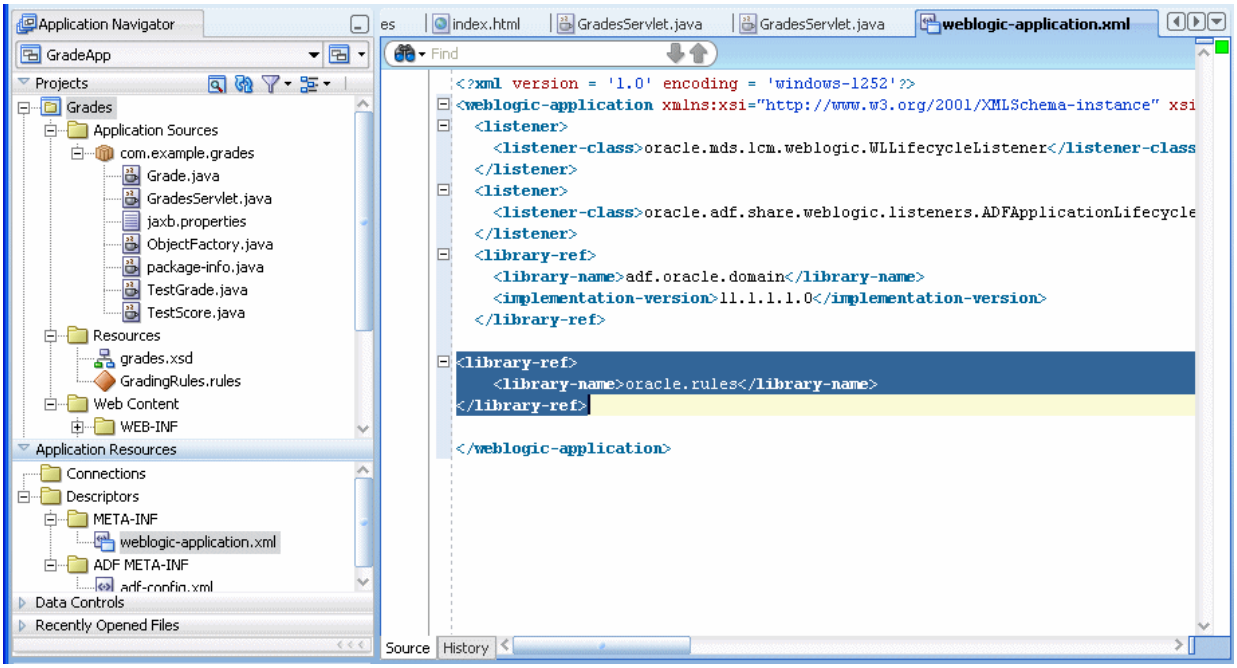
How to Add the Rules Library to the Grades Sample Application

To add the rules library to the weblogic-application file:

1. In the GradeApp application, in the Application Navigator expand **Application Resources**.
2. Expand **Descriptors** and expand **META-INF** and double-click to open **weblogic-application.xml**.
3. Add the `oracle.rules` library reference to the `weblogic-application.xml file`. Add the following lines, as shown in Figure 9-21.

```
<library-ref>
    <library-name>oracle.rules</library-name>
</library-ref>
```

Figure 9-21 Adding Oracle Rules Library Reference to WebLogic Descriptor



4. Save the `weblogic-application.xml` file.

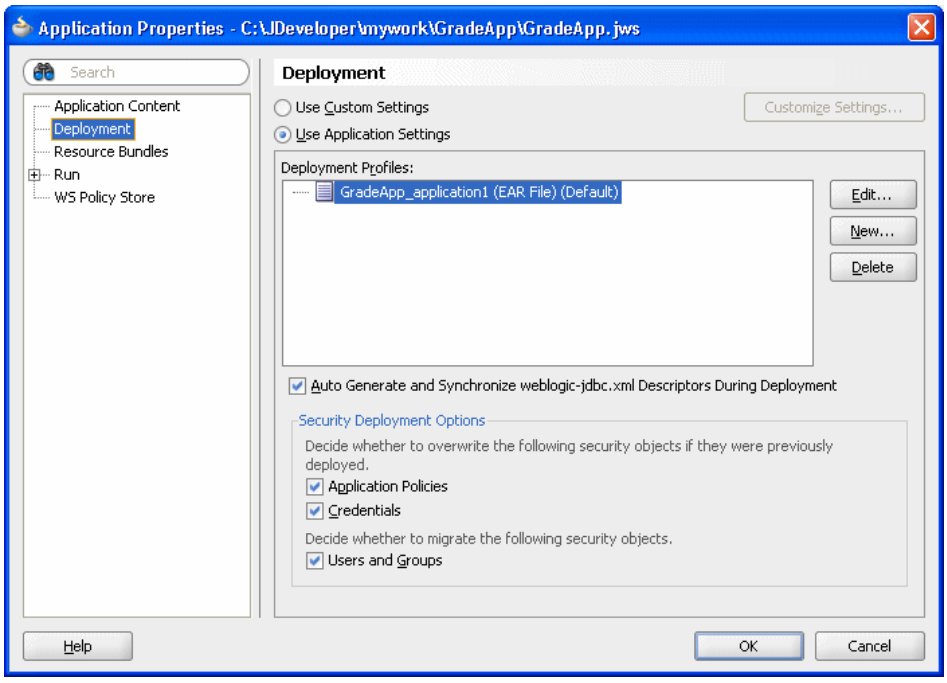
How to Add the MDS Deployment File to the Grades Sample Application

To add the MDS deployment file:

1. In the Application Navigator, select the **GradeApp** application.
2. Right-click the **GradeApp** application and in the context menu select **Application Properties...**.

   This displays the Application Properties dialog.
3. In the Application Properties navigator select the **Deployment** item, as shown in Figure 9-22.

Figure 9-22 Application Properties - Deployment



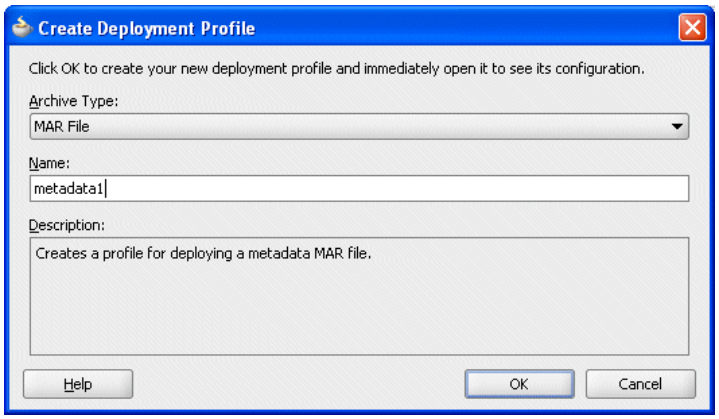
4. In the Application Properties dialog, click **New...**.

   This displays the Create Deployment Profile dialog.

5. Configure this dialog as follows, as shown in Figure 9-23:

   - **Archive Type**: `MAR File`
   - **Name**: `metadata1`

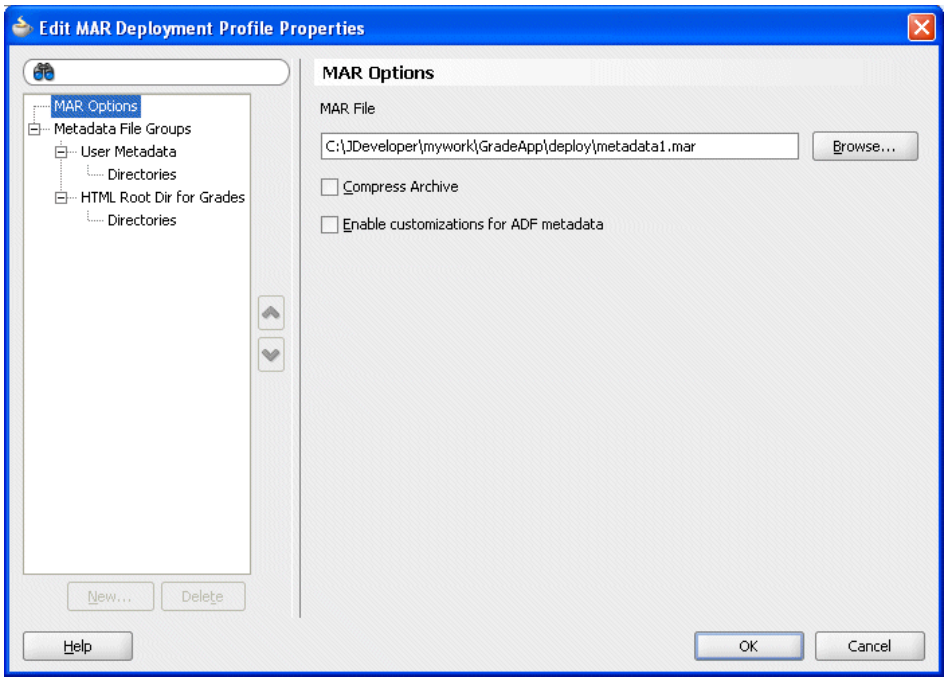
Figure 9-23 Create Deployment Profile Dialog for MAR File



Click **OK**.

This displays the Edit MAR Deployment Properties dialog as shown in Figure 9-24.

Figure 9-24 Edit MAR Deployment Profile Properties - MAR Options



6. Expand the **Metadata File Groups** item and select the **User Metadata** item and click **Add**.

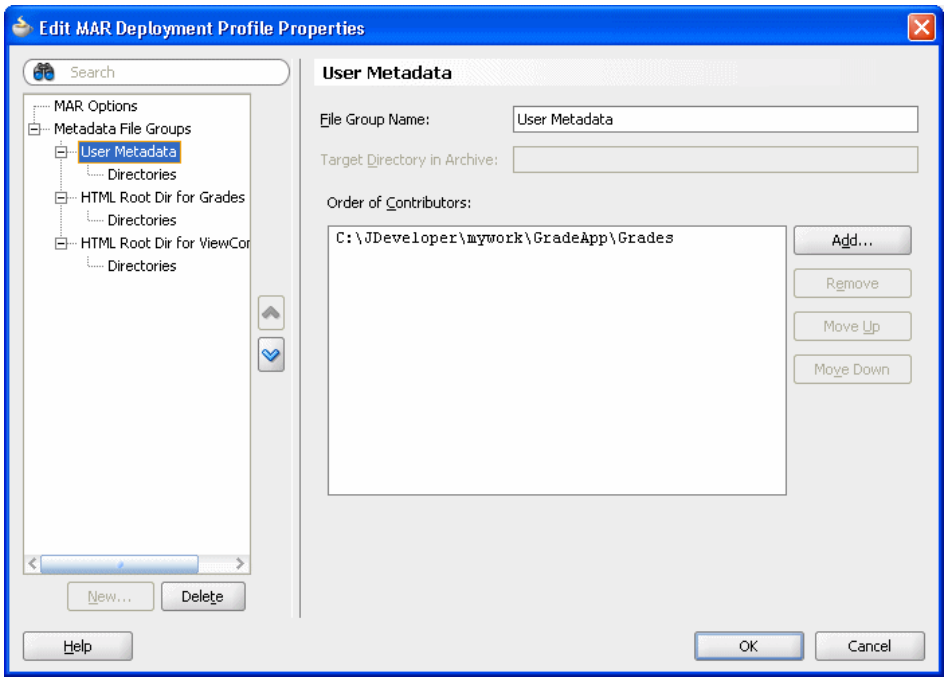
   This displays the Add Contributor dialog.

7. In the Add Contributor dialog, click the **Browse** button and navigate to the directory for the project that contains the `GradingRules.rules` dictionary file.

   In this example, navigate to `C:\JDeveloper\mywork\GradeApp\Grades` and click **Select**.

8. In the Add Contributor dialog, click **OK** to close the dialog. This displays the Edit MAR Deployment Properties dialog as shown in Figure 9-25

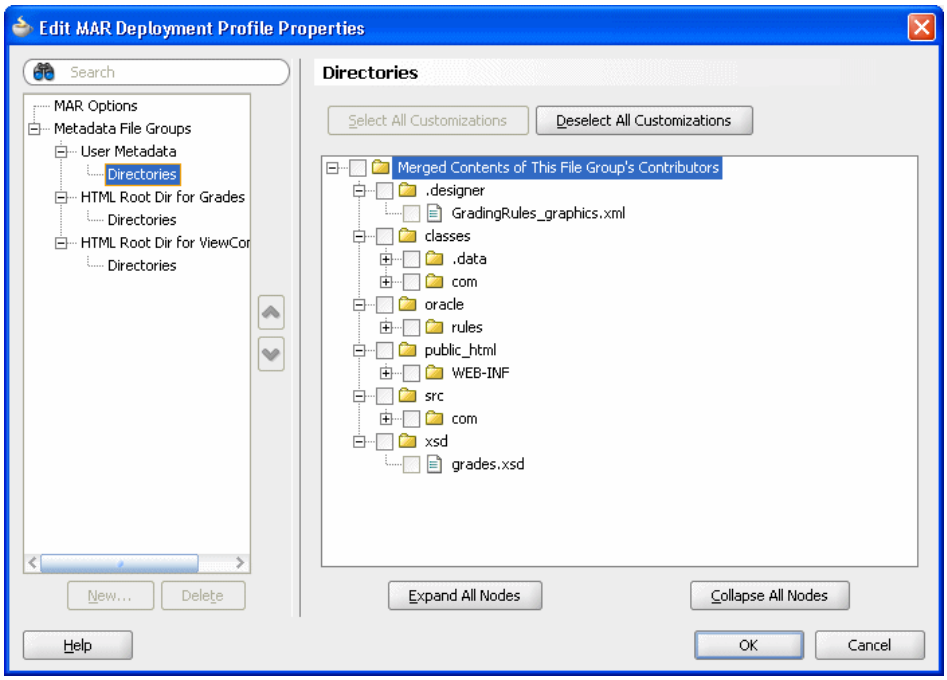
Figure 9-25 Edit MAR Deployment Profile Properties - User Metadata



9. In the Edit MAR Deployment Profile Properties dialog, expand the **Metadata File Groups** and expand the **User Metadata** item and select **Directories**.

   This displays the Directories page as shown in Figure 9-26.

Figure 9-26 Edit MAR Deployment Profile Properties - Directories

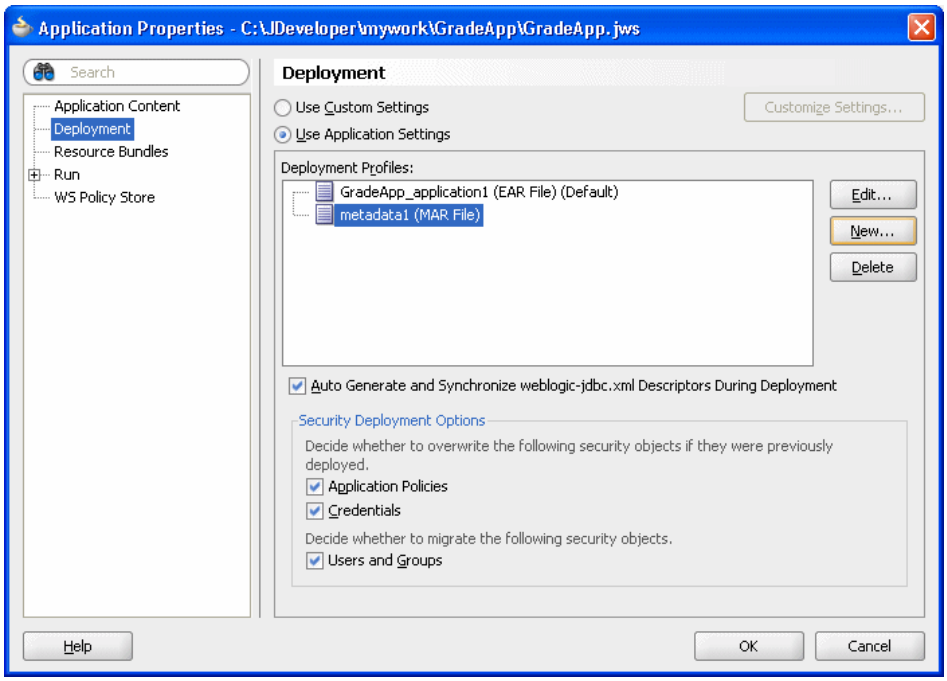


10. Select the **oracle** directory check box. This selects the **GradingRules.rules** dictionary to be included in the MAR.

    Click **OK**.

JDeveloper creates an application deployment profile named `metadata1 (MAR File)` as shown in Figure 9-27.

Figure 9-27 Application Properties - Deployment - MAR



Click **OK** in the Application Properties dialog.

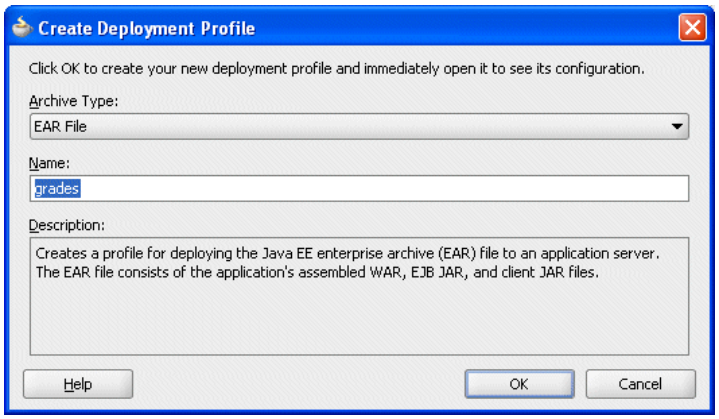
How to Add the EAR File to the Grades Sample Application

Add an EAR file to the Grades sample application.

To add the ear file to the grades sample application:

1. In the Application Navigator, select the **GradeApp** application.
2. Right-click and in the context menu select **Application Properties...**.
3. In the Application Properties dialog, select **Deployment** and click **New...**. This displays the Create Deployment Profile dialog.
4. Configure this dialog as follows, as shown in Figure 9-28.
   - **Archive Type**: EAR
   - **Name**: `grades`

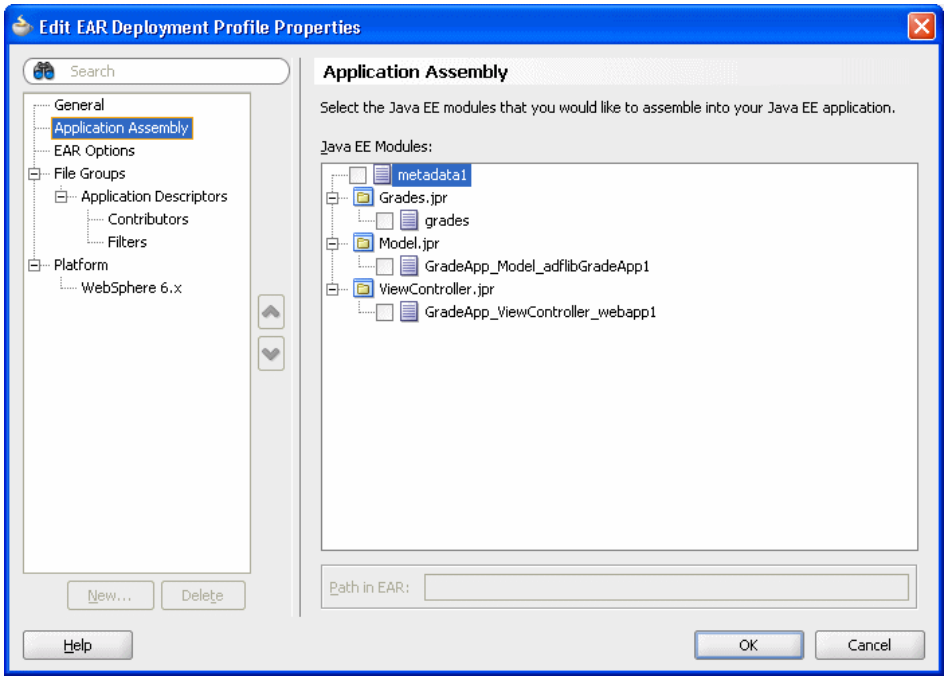
Figure 9-28 Create Deployment Profile Dialog for EAR File



5. Click **OK**. This displays the Edit EAR Deployment Profile Properties dialog.

6. In the Edit Ear Deployment Profile Properties dialog, in the navigator select **Application Assembly** as shown in Figure 9-29.

Figure 9-29 Edit EAR Deployment Profile Properties - Application Assembly



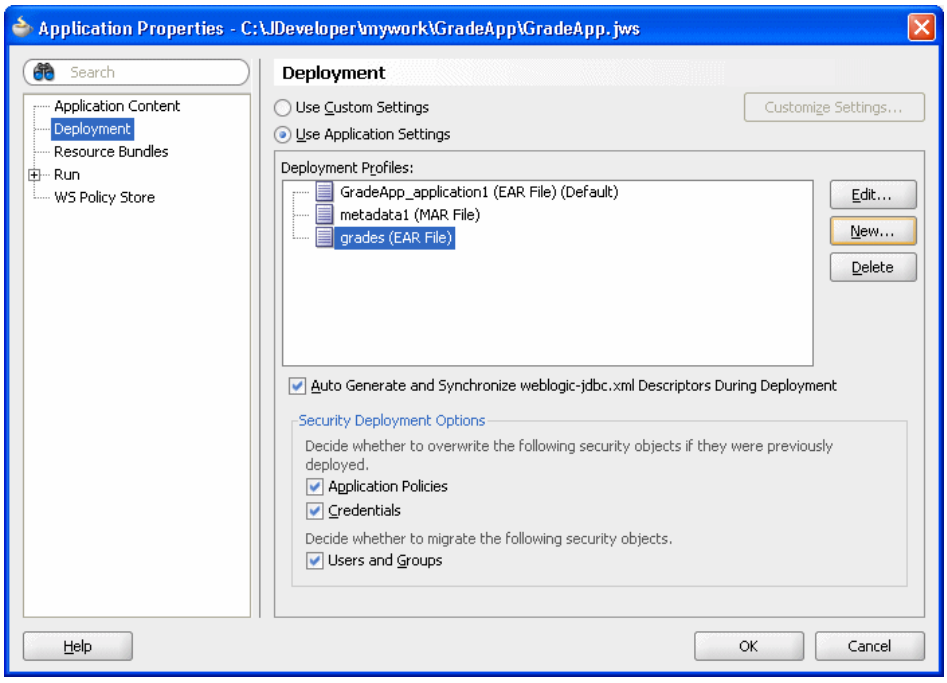
7. Configure this dialog as follows:

   - Select the **metadata1** check box.
   - Expand the **Grades.jpr** item and select the **grades** check box.

8. In the Edit EAR Deployment Profile Properties dialog, click **OK**.

   JDeveloper creates an application deployment profile named `grades(EAR File)` as shown in Figure 9-30.

Figure 9-30 Application Properties - Deployment - EAR



9. Click **OK** to close the Application Properties dialog.

10. Select **Save All** from the **File** main menu to save your work.

Deploying and Running the Grades Sample Application

You can now deploy and run the grades sample application on Oracle WebLogic Server.

How to Deploy to Grades Sample Application

To deploy the grades sample application:

1. In the Application Navigator, select the GradeApp application.

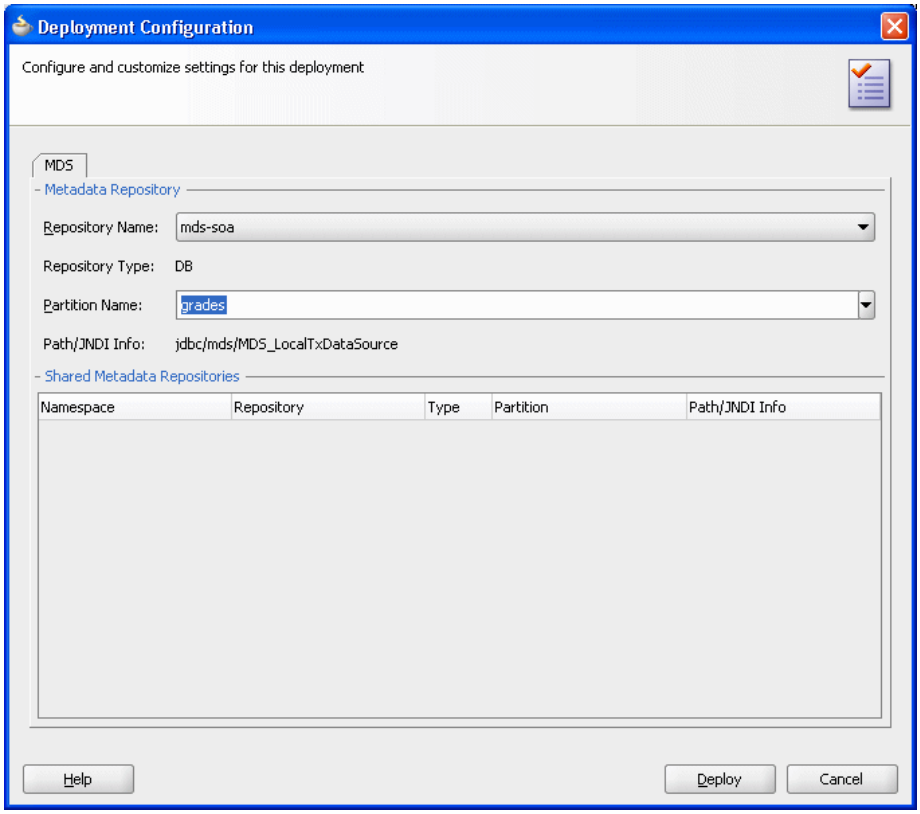
2. Right-click the **GradeApp** application and in the context menu select **Deploy > grades > to >** and select either an existing connection or **New Connection...** to create a connection for the deployment. This starts the deployment to the specified Oracle WebLogic Server.

3. As the deployment proceeds, Oracle JDeveloper shows the Deployment Configuration dialog.

4. In the Deployment Configuration dialog enter the following values, as shown in Figure 9-28:

   - In the **Repository Name** field, from the list, select: **mds-soa**
   - In the **Partition Name** field, enter **grades**

Figure 9-31 Deployment Configuration Dialog for MDS with Repository and Partition



5. In the Deployment Configuration dialog, click **Deploy**.

How to Run the Grades Sample Application

After you deploy the grades sample application, you can run the application.

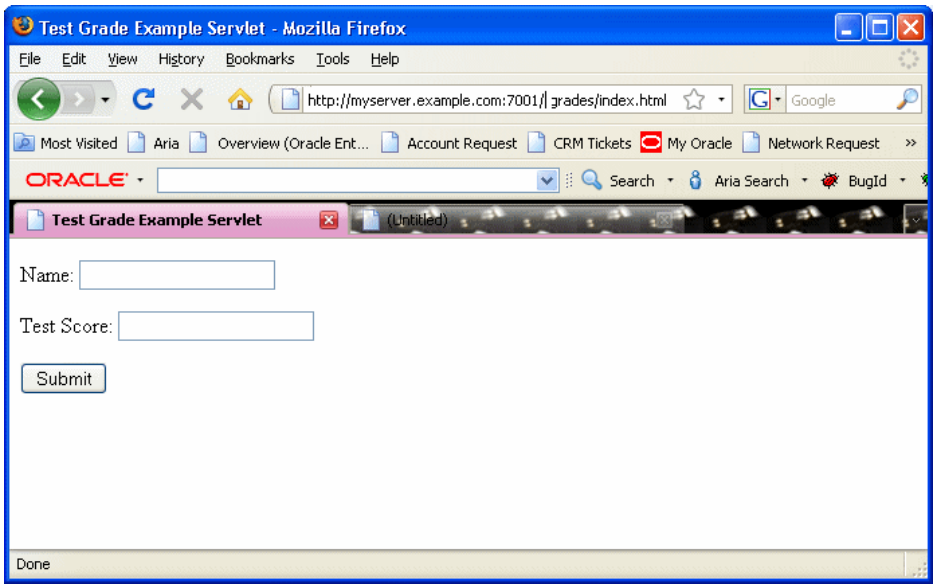
To run the grades sample application:

1. Point a web browser at,

   http://*yourServerName*:*port*/grades/

   This displays the test servlet as shown in Figure 9-32.

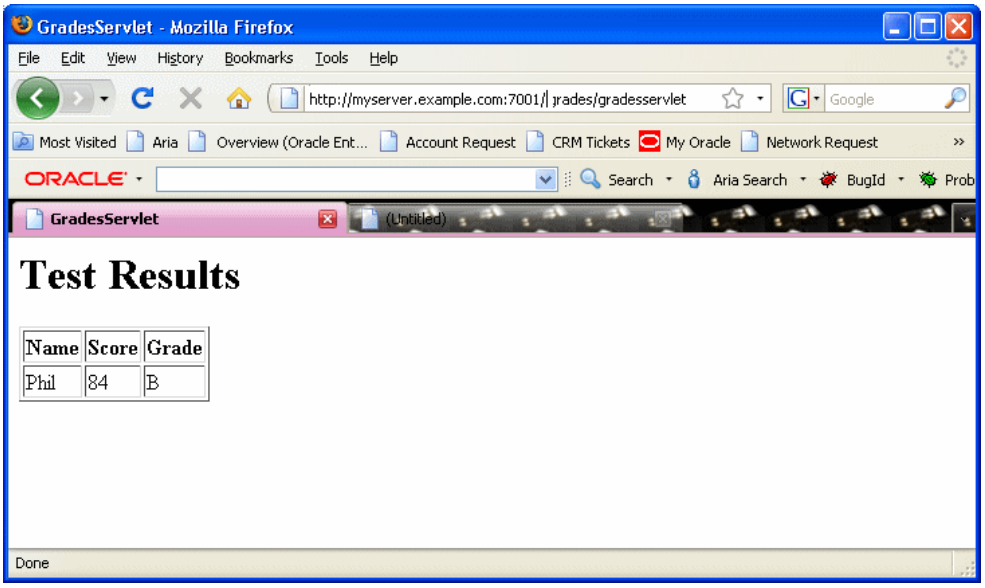
Figure 9-32 Grades Sample Application Servlet



2. Enter a name and test score and click **Submit**. This returns results as shown in Figure 9-33.

The first time you run the servlet there may be a delay before any results are returned. The first time the servlet is invoked, during servlet initialization the runtime loads the dictionary and creates a rule session pool. Subsequent invocations do not perform these steps and should run much faster.

Figure 9-33 Grades Sample Application Servlet with Results



10

Working with Oracle Business Rules and ADF Business Components

This chapter describes how Oracle Business Rules enables you to use Oracle ADF Business Components view objects as facts. As with all fact types, trees of facts, representing master-detail data, can be asserted as a unit. Oracle Business Rules has built-in tests (RLcontains) and optional "tree mode" syntax (for example, master-detail) to support navigating one-to-many relationships.

The chapter includes the following sections:

Introduction to Using Business Rules with ADF Business Components

The ADF Business Components rule development process can be summarized as below.

Using ADF Business Components rule, you can:

1. Create view object definitions.
2. Create action types.
3. Create rule dictionary.
4. Register view object fact types.
5. Register Java fact types for actions.
6. If you are invoking from Java:
    - If the view object is already instantiated at the Decision Point, code the Decision Point invocation passing the view object instance.
    - If the view object is not instantiated at the Decision Point, code the Decision Point invocation passing the view object key values.

Understanding Oracle Business Rules ADF Business Components Fact Types

When an ADF Business Components view object is imported into an Oracle Business Rules data model, an ADF Business Components fact type is created which has a property corresponding to each attribute of the view object.

Additionally, the ADF Business Components fact type contains the following:

- A property named **ViewRowImpl** which points directly to the `oracle.jbo.Row` instance that each fact instance represents.
- A property named **key_values** which points to an `oracle.rules.sdk2.decisionpoint.KeyChain` object. You can use this property to retrieve the set of key-values for this row and its parent rows.

Note the following:

- Relationships between view object definitions are determined by introspection of attributes on the View Definition, specifically, those attributes which are View Link Accessors.

  The ADF Business Components fact type importer correctly determines which relationships are 1-to-1 and which are 1-to-many and generates definitions in the dictionary accordingly. For 1-to-many relationships the type of the property generated is a `List` which contains facts of the indicated type at runtime.

- ADF Business Components fact types are not Java fact types and do not allow invoking methods on any explicitly created implementation classes for the view object.

  If you need to call such methods then add the view object implementation to the dictionary as a Java fact type instead of as an ADF Business Components fact type. In this case, all getters and setters and other methods become available but the trade-off is that related view objects become inaccessible and, should related view object access be required, these relationships must be explicitly managed.

- Internally in Oracle Business Rules, when you use ADF Business Components fact types these fact types are created as instances of RL fact types. Thus, you cannot assert ADF Business Components view object instances directly to a Rule Session, but must instead use the helper methods provided in the `MetadataHelper` and `ADFBCFactTypeHelper` classes. For more information, see *Java API Reference for Oracle Business Rules*.

Understanding Oracle Business Rules Decision Point Action Type

With Rules SDK, the primary way to update a view object within a Decision Point is with an action type. An action type is a Java class that you import into the rule dictionary data model in the same way you import a rule pattern fact type Java class. A new instance of this action type is then asserted in the action of a rule and then processed by the Postprocessing Ruleset in the `DecisionPointDictionary`.

A Java class to be used as an action type must conform to the following requirements:

- The Java fact type class must subclass `oracle.rules.sdk2.decisionpoint.ActionType` or `oracle.rules.sdk2.decisionpoint.KeyedActionType`.

  By subclassing `KeyedActionType` the Java class inherits a standard `oracle.rules.sdk2.decisionpoint.KeyChain` attribute, which may be used to communicate the rule fact's primary keys and parent-keys to the `ActionType` instance.

- The class has a default constructor.
- The class implements abstract `exec` method for the `ActionType`. The `exec` method should contain the main action which you want to perform.

- The Java class must have properties which conform to the `JavaBean` interface (that is, each property must have a getter and setter method).

See Example 10-1 for a sample `ActionType` implementation.

Table 10-1 shows the methods in `DecisionPointInstance` that an application developer might need when implementing the `ActionType exec`.

Table 10-1 DecisionPointInstance Methods

Method	Description
`getProperties`	Supplies a `HashMap<String,Object>` object containing any runtime-specified parameters that the action types may need. If you intend to use the decision function from a Decision service, use only String values.
`getRuleSession`	Gives access to the Oracle Business Rules RuleSession object from which static configuration variables in the Rule Dictionary may be accessed.
`getActivationID`	If populated by the caller, supplies a String value to be used for Set Control indirection.
`getTransaction`	Provides a transaction object so that action types may make persistent changes in the back end.
`addResult`	Adds a named result to the list of output values in the form of a String key and Object value. Output is assembled as a `List` of `oracle.rules.sdk2.decisionpoint.DecisionPointInstance.NamedValue` objects as would be the case in a pure map implementation. The `NamedValue` objects are simple data-bearing classes with a getter each for the name and value. Output values from one action types instance are never allowed to overwrite each other, and in this regard, the action type implementations should be considered completely independent of each other.

Sample ActionType Implementation

Example 10-1 shows a sample `ActionType` implementation and an `oracle.rules.sdk2.decisionpoint.DecisionPointInstance` as a parameter to the `exec` method.

Example 10-1 Implementing an ActionType

```
package com.example;

import oracle.jbo.domain.Number;

import oracle.rules.sdk2.decisionpoint.ActionType;
import oracle.rules.sdk2.decisionpoint.DecisionPointInstance;

public class RaiseAction extends ActionType {
    private double raisePercent;

    public void exec(DecisionPointInstance dpi) {
        Number salary = (Number)getViewRowImpl().getAttribute("Salary");
        salary = (Number)salary.multiply(1.0d + getRaisePercent()).scale(100,2, new boolean[]
{false});
        dpi.addResult("raise for " + this.getViewRowImpl().getAttribute("EmployeeId"),
                      getRaisePercent() + "=>" + salary );
        getViewRowImpl().setAttribute("Salary", salary);
    }
```

```
    public void setRaisePercent(double raisePercent) {
        this.raisePercent = raisePercent;
    }

    public double getRaisePercent() {
        return raisePercent;
    }
}
```

Using Rules Designer you can select parameters appropriate for the `ActionType` you are configuring.

Using Decision Points with ADF Business Components Facts

You can use a Decision Point to execute a decision function. There are certain Decision Point methods that only apply when working with ADF Business Components Fact types.

For more information on decision functions, see Working with Decision Functions .

How to Call a Decision Point with ADF Business Components Facts

When you use ADF Business Components fact types you invoke a decision function using the Rules SDK Decision Point interface.

To call a decision function using the Rules SDK Decision Point interface:

1. Construct and configure the template `DecisionPoint` instance using the `DecisionPointBuilder`.

   For more information, see How to Add a Decision Point Using Decision Point Builder.

2. Create a `DecisionPointInstance` using the `DecisionPoint` method `getInstance`.

3. Add the fact objects you want to use to the `DecisionPointInstance` using `DecisionPointInstance` method `addInput`, `setInputs`, or `setViewObject`. These are either `ViewObject` or `ViewObjectReference` instances. These must be added in the same order as they are declared in the decision function input. For more information, see Calling the Invoke Method for an ADF Business Components Rule

4. Set the transaction to be used by the `DecisionPointInstance`.

   For more information, see Setting the Decision Point Transaction.

5. Set any runtime properties the consequent application actions may expect.

   For more information, see Setting Runtime Properties.

6. Call the `DecisionPointInstance` method `invoke`.

   For more information, see:

   - Calling the Invoke Method for an ADF Business Components Rule
   - What You Need to Know About Decision Point Invocation

Setting the Decision Point Transaction

The Oracle Business Rules SDK framework requires an `oracle.jbo.server.DBTransactionImpl2` instance to load a `ViewObject` and to provide `ActionType` instances within a transactional context. The class `oracle.jbo.server.DBTransactionImpl2` is the default JBO transaction object returned by calling the `ApplicationModule` method `getTransaction`. Setting the transaction requires calling the `DecisionPointInstance` method `setTransaction` with the `Transaction` object as a parameter.

Should a `DBTransaction` instance not be available for some reason, the Oracle Business Rules SDK framework can bootstrap one using any of the three provided overrides of the `setTransaction` method.

These require one of:

- A JDBC URL, user name, and password.
- A JDBC connection object.
- A `javax.sql.DataSource` object and a flag to specify whether the `DataSource` represents a JTA transaction or a local transaction.

Setting Runtime Properties

Runtime properties may be provided with the `setProperty` method. These can then be retrieved by `ActionType` instances during their execution. If no runtime properties are needed, you may safely omit these calls.

Calling the Invoke Method for an ADF Business Components Rule

The `ViewObject` to be used in a Decision Point invocation can be specified in one of two ways, as shown in Table 10-2.

Table 10-2 Setting the View Object for a Decision Point Invocation

ViewObject Set Method	Description
`setViewObject`	The decision function is invoked once for each `ViewObject` row. This the preferred way to use view objects. Between each invocation of the decision function, the rule session is not reset so any asserted facts from previous invocations of the decision function are still in working memory. In most cases, users should write rules that retract the asserted facts before the decision function call completes. For example, you can have a cleanup ruleset that retracts the `ViewObject` row that runs before the Postprocessing decision function is called. How to Add Retract Employees Ruleset shows this usage. To use `setViewObject`, the `ViewObject` must be the first entry in the decision function InputTable.
`addInput` `setInputs`	The decision function is invoked once with all of the `ViewObject` rows loaded at the same time. This is generally not a scalable operation, since hundreds of thousands of rows can be loaded at the same time. There are some cases where there are a known small number of rows in a `ViewObject` that this method of calling the `ViewObject` can be useful.

What You Need to Know About Decision Point Invocation

Care must be taken when invoking Decision Points using a view object that loads large amounts of data, since the default behavior of the JBO classes is to load all data eagerly. If a view object with many rows and potentially very many child rows is loaded into memory, not only is there risk of memory-exhaustion, but DML actions taken based on such large data risk using all rollback segments.

Sample to Invoke a Decision Point Using setinputs Method

Example 10-2 shows how to invoke a Decision Point with a `ViewObject` instance using the `setInputs` method. For the complete example, see the example shown in How to Add the Outside Manager Finder Class.

Example 10-2 Invoking a Decision Point Using setInputs Method

```
public class OutsideManagerFinder {
    private static final String AM_DEF = "com.example.AppModule";
    private static final String CONFIG = "AppModuleLocal";
    private static final String VO_NAME = "EmployeesView1";

    private static final DictionaryFQN DICT_FQN =
                 new DictionaryFQN("com.example", "Chapter10Rules");

    private static final String DF_NAME = "FindOutsideManagers";

    private DecisionPoint dp = null;

    public OutsideManagerFinder() {
        try {
            dp = new DecisionPointBuilder()
                            .with(DICT_FQN)
                            .with(DF_NAME)
                            .build();
        } catch (SDKException e) {
            System.err.println(e);
        }
    }


    public void run() {
        final ApplicationModule am =
                  Configuration.createRootApplicationModule(AM_DEF, CONFIG);
        final ViewObject vo = am.findViewObject(VO_NAME);
        final DecisionPointInstance point = dp.getInstance();
        point.setTransaction((DBTransactionImpl2)am.getTransaction());
        point.setAutoCommit(true);
        point.setInputs(new ArrayList<Object>(){{ add(vo); }});
        try {
            List<Object> invokeList = point.invoke();

List<DecisionPoint.NamedValue> results = point.getResults();

        } catch (RLException e) {
            System.err.println(e);
        } catch (SDKException e) {
            System.err.println(e);
        }
    }
```

Sample to Invoke a Decision Point Using setViewObject Method

Example 10-3 shows how to invoke a `DecisionPoint` using the `setViewObject` method to set the `ViewObject`.

Example 10-3 Invoking a Decision Point Using setViewObject Method

```
    public void run() {
        final ApplicationModule am =
                Configuration.createRootApplicationModule(AM_DEF, CONFIG);
        final ViewObject vo = am.findViewObject(VO_NAME);
        final DecisionPointInstance point = dp.getInstance();

        point.setTransaction((DBTransactionImpl2)am.getTransaction());
        point.setAutoCommit(true);
        point.setViewObject(vo);
        try {
            List<Object> invokeList = point.invoke();

List<DecisionPoint.NamedValue> results = point.getResults();

        } catch (RLException e) {
            System.err.println(e);
        } catch (SDKException e) {
            System.err.println(e);
        }
    }
```

How to Call a Decision Function with Java Decision Point Interface

To call a decision function with a ruleset using ADF Business Components fact types with the Oracle Business Rules SDK Decision Point interface you must configure the decision function with certain options. For more information on using decision functions, see Working with Decision Functions.

To define a decision function using the Java Decision Point interface:

1. In the Decision Functions tab, select the appropriate Decision Function and click the **Edit** button. The Edit Decision Function dialog appears.
2. In the Edit Decision Function dialog, configure the decision function:

   - **Inputs**: names the fact types to use in the configured business rules.

     The inputs, when working with an application using ADF Business Components fact types, are the ADF Business Components view objects used in your rules.

     When you use the `setViewObject` method with a Decision Point, the **List** attribute should be cleared. Each Input fact type should have the **List** attribute selected when you are using `addInput` or `setInputs` methods with the Decision Point. Optionally, depending on the usage of the view objects, select the **Tree** attribute:

     List: defines that a list of ADF Business Components fact types are passed to the decision function.

     Tree: defines that all objects in the master-detail hierarchy should be asserted, instead of only the top-level object.

For more information, see How to Call a Decision Point with ADF Business Components Facts.

- **Initial Actions**: click <insert action> to add actions that can be used to change input facts before they are asserted, change the ruleset stack, set the effective date, or even assert output facts. These actions can be used instead of rules, or to "set up" the environment for running rules.

  For more information on using decision functions, see Working with Decision Functions.

- **Output Fact Types**: defines the fact types that the caller returns.

  When calling a decision function using the Java Decision Point interface for a decision function that uses ActionTypes, two ways of returning output are available:

  – Output fact types can be used, as with any decision function. These results are returned from DecisionPointInstance.invoke().

  – The ActionType's exec method can be overridden to call dpi.addResult (see Example 10-1). These results are returned from DecisionPointInstance.getResults().

  Using ActionTypes is optional. Typically ActionTypes would not be used unless the rules need to modify the ADF data. If ActionTypes are not used, then DecisionPointDictionary.Preprocessing and DecisionPointDictionary.Postprocessing are not needed.

  For more information, see Understanding Oracle Business Rules Decision Point Action Type.

- **RuleSets and Decision Functions**: an ordered list of the rulesets and other decision functions that this decision function executes. The rulesets **DecisionPointDictionary.Preprocessing** and **DecisionPointDictionary.Postprocessing** from the **DecisionPoint** dictionary must be added so that they run before and after, respectively, the application-specific rulesets and decision functions.

What You Need to Know About Decision Function Configuration with ADF Business Components

Both rulesets and decision functions may be included in the definition of a decision function. It is common for an application to require some rules or decision functions which act as "plumbing code". This plumbing code is only needed if you are using ActionType.

Such applications include components that perform transformations on the input data, assert auxiliary facts, or process output facts. The plumbing code may need to run before or after the rules that contain the core business rules of the application.

You can separate these application concerns and their associated rules from the application functional concerns using *nested decision functions*. Using nested decision functions, the inner decision function does not contain the administrative, plumbing-oriented concerns, and thus only presents those rules which define the core logic of the application. This design eliminates the need for the user to understand the administrative rules and prevents a user from inappropriately modifying these rules (and possibly rendering the system inoperable due to these changes).

To create a configuration using multiple rulesets and nested decision functions, create two decision functions and add one to the other. A good naming scheme is to suffix the nested inner decision function with the name `Core`. The user specified rulesets can be added to the inner `Core` decision function. For example, **DecisionFunction_1** can be defined to run the **DecisionPointDictionary.Preprocessing** decision function, the **DecisionFunction_1Core** decision function, and the **DecisionPointDictionary.Postprocessing** decision function. For this example, **DecisionFunction_1Core** contains the core business logic rulesets.

It is also common for the input of a Decision Point to be an ADF Business Components fact type that is the root of a tree of ADF Business Components objects. However, the user might only write business rules that match on a subset of the types found in the tree. In this case, it is a good practice to define the inputs of the nested decision functions to be only the types which are actually matched in the contained rulesets. For example, consider a Decision Point calling a decision function whose input is an `Employee` fact type with the **Tree** option selected; if this decision function includes a nested decision function with rulesets that only matched on the `Department` fact type. In this case, the nested decision function could either have as its input specified as an `Employee` fact type with the **Tree** option selected, or a `Department` fact type with the **List** option selected. For this example, the **Tree** option causes the children of the `Employee` instances, including the `Department` instances to be asserted (due to the one-to-many relationship between these types). If `Employee` is an input to the outer decision function and the **Tree** option is selected, the then `Department` fact type instances are asserted, and you can identify the signature on the inner decision function as a list of `Department` instances (these are the exact types which are being matched on for this decision function).

Creating a Business Rules Application with ADF Business Components Facts

The ADF Business Components sample application shows the use of ADF Business Component fact types.

The source code for Oracle Business Rules-specific samples and SOA samples are available online in the Oracle SOA Suite samples page.

How to Create an Application That Uses ADF Business Components Facts

To work with Oracle Business Rules with ADF Business Components facts, you first need to create an application and a project in Oracle JDeveloper.

To create an application that uses ADF Business Components facts:

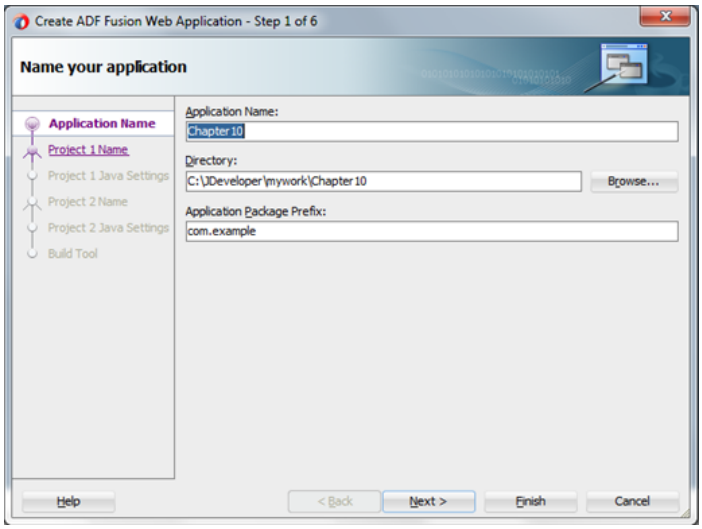
1. Start Oracle JDeveloper. This displays the Oracle JDeveloper start page.
2. In the Application Navigator, in the application menu click **New Application...**.
3. In the **Name your application** page enter the name and location for the new application:
   a. In the **Application Name** field, enter an application name. For example, enter `Chapter10`.
   b. In the **Directory** field, enter or browse for a directory name or accept the default.

c. In the **Application Package Prefix** field, enter an application package prefix. For example, enter `com.example`.

This should be a globally unique prefix and is commonly a domain name owned by your company. The prefix, followed by a period, applies to objects created in the initial project of an application.

In this sample, use the prefix `com.example`, as shown in Figure 10-1

Figure 10-1 Step 1 of 6



Click **Next.**

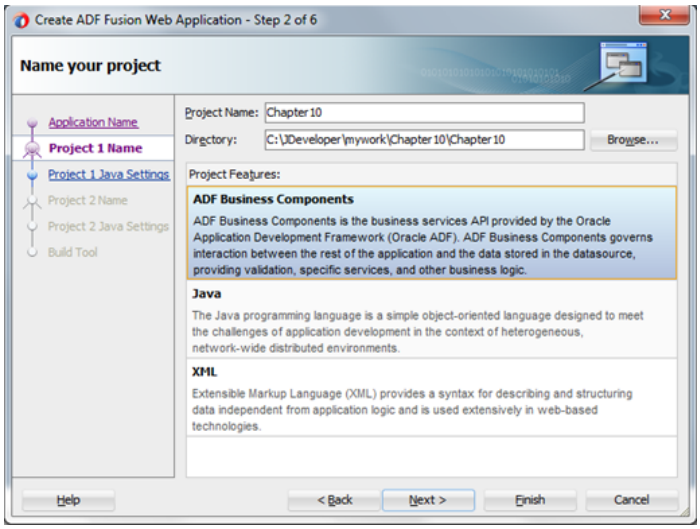
4. On the **Name your project** page:

a. In the **Project Name** field, enter `Chapter10`.

b. In the **Directory** field, enter or browse for a directory name or accept the default.

c. In the **Project Features** area, select **ADF Business Components** as shown in Figure 10-2.

Figure 10-2 Step 2 of 6



Click **Next.**

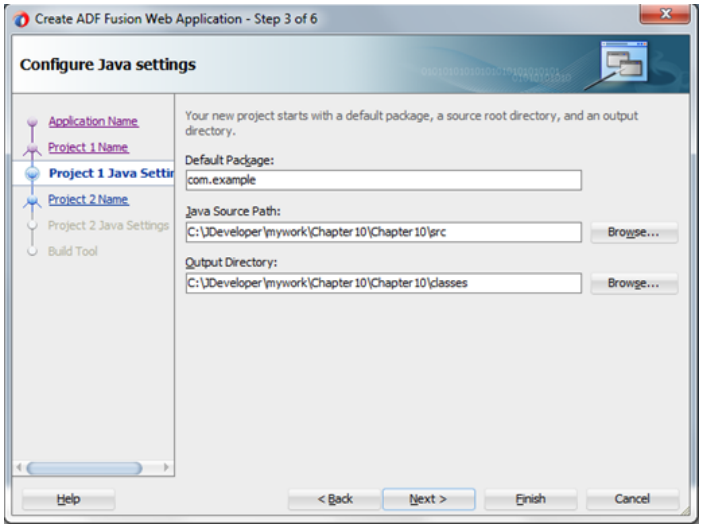
5. On the **Configure Java settings** page:

   a. In the **Default Package** field, enter `com.example`.

   b. In the **Java Source Path** field, enter or browse for a directory name or accept the default.

   c. In the **Output Directory** field, enter or browse for a directory name or accept the default as shown in Figure 10-3:

Figure 10-3 Step 3 of 6



   Click **Next**.

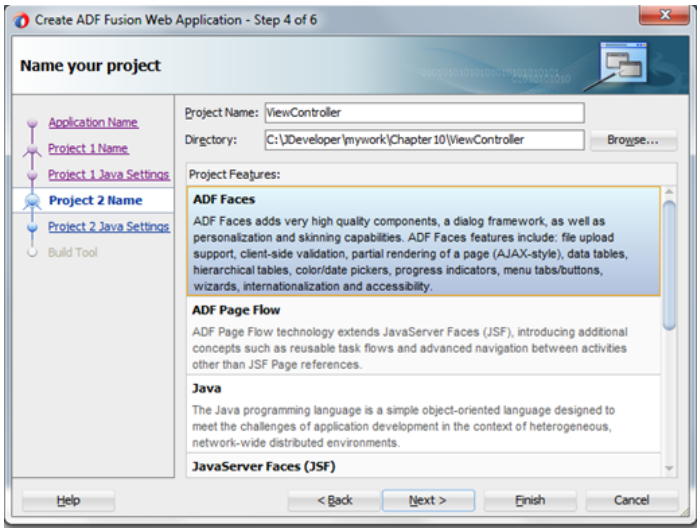
6. For Project 2, on the **Name your project** page:

   a. In the **Project Name** field, enter `ViewController`.

   b. In the **Directory** field, enter or browse for a directory name or accept the default.

   c. In the **Project Features** area, select **ADF Faces** as shown in Figure 10-4:

Figure 10-4 Step 4 of 6



Click **Next**.

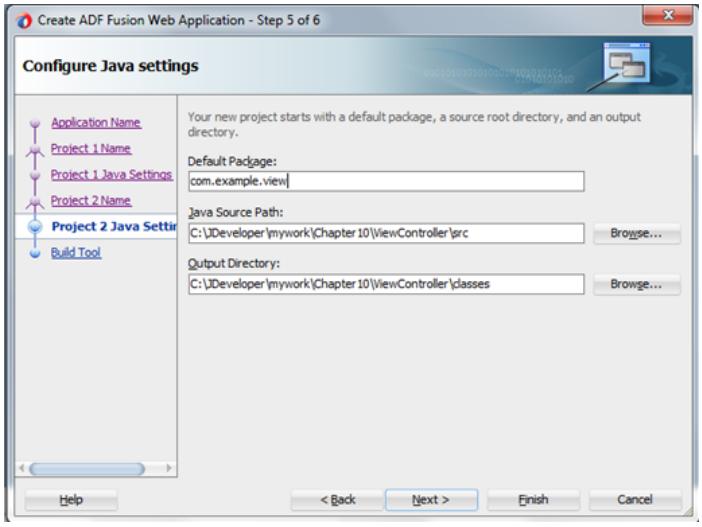
7. For Project 2, on the **Configure Java settings** page:

   a. In the **Default Package** field, enter `com.example.view`.

   b. In the **Java Source Path** field, enter or browse for a directory name or accept the default.

   c. In the **Output Directory** field, enter or browse for a directory name or accept the default, as shown in Figure 10-5:

Figure 10-5 Step 5 of 6



8. Click **Finish** when done.

How to Create ADF Business Components Application for Business Rules

You need to add ADF Business Components from a database table. For this example we use the standard HR database tables.

To add ADF Business Components:

1. In the Application Navigator, select the **Chapter10** project.
2. Right-click and from the menu select **New...**.
3. In the New Gallery, in the **Categories** area expand **Business Tier** and select **ADF Business Components**.
4. In the **Items** area select **Business Components from Tables** and click OK.
5. In the Initialize Business Components Project dialog, enter the required connection information to add a connection. Click **OK**.

   This displays the Create Business Components from Tables wizard.
6. In the Entity Objects page, select the desired objects by moving objects from the **Available** box to the **Selected** box. You may need to click **Query** to see the complete list. For example, select **DEPARTMENTS** and **EMPLOYEES**.
7. Click **Next**. This displays the Updatable View Objects page.
8. In the Updatable View Objects page select **Departments** and **Employees**.
9. Click **Next**. This displays the Entity based view object page.
10. Click **Next**. This displays the Application Module page.
11. Click **Finish** when done.

How to Update View Object Tuning for Business Rules Sample Application

You should tune the `ViewObject` to meet the performance requirements of your application.

To set tuning options for EmployeesView or DepartmentsView:

1. In the Application Navigator, double-click **EmployeesView** to set tuning options for employees or **DepartmentsView** to set tuning options for departments.
2. In the **General** navigation tab, expand **Tuning**.
3. In the Tuning area, select **All Rows**.
4. In the Tuning area, in the **Batches of:** field, enter 128.
5. In the Tuning area, select **All at Once**.

How to Create a Dictionary for Oracle Business Rules

You use Oracle JDeveloper to create an Oracle Business Rules dictionary.

To create a dictionary:

1. In the Application Navigator, select the **Chapter10** project.
2. Right-click, and from the list select **New...**.
3. In the New Gallery, select the **All Technologies** tab and in the **Categories** area expand **Business Tier** and select **Business Rules**.

4. In the New Gallery, in the **Items** area select **Business Rules**.
5. Click **OK**.
6. In the Create Business Rules dialog enter the dictionary name and package.
   - For example, in the **Name** field enter `Chapter10Rules`.
   - For example, in the **Package** field enter `com.example`.
7. Click **OK**.

   JDeveloper creates the dictionary and opens the `Chapter10Rules.rule`s file in Rules Designer.

How to Add Decision Point Dictionary Links

You need to add a dictionary links to the Oracle Business Rules supplied Decision Point Dictionary. This dictionary supports features for working with the Decision Point interface with ADF Business Components objects.

Add decision point dictionary links:

1. In the Rules Designer, click the **Links** navigation tab.
2. From the menu next to the **Create** button, select **Decision Point Dictionary**. This operation can take a while to complete. After waiting, Rules Designer adds a link to the Decision Point Dictionary.

How to Import the ADF Business Components Facts

You import ADF Business Components facts with Rules Designer to make these objects available when you create rules.

Import the ADF Business Components facts:

1. In Rules Designer, select the **Facts** navigation tab.
2. Select the **ADF-BC Facts** tab.
3. Click the **Create...** button. This displays the ADF Business Components Fact page.
4. In the **Connection** field, from the list select the connection which your ADF Business Components objects use. The **Search Classpath** area shows a list of classpaths.
5. In the **View Definition** field, select the name of the view object to import. For example, select **com.example.EmployeesView**.
6. Click **OK**. This displays the **Facts** navigation tab.

ADF Business Components Facts can include a circular reference, as indicated with the validation warning:

```
RUL-05037: A circular definition exists in the data model
```

When this warning is shown in the Business Rule validation log, you need to manually resolve the circular reference. To do this you clear the **Visible** check box for one of the properties that is involved in the circular reference.

How to Mark a Property as Non-visible

To mark a property as non-visible:

1. Select the **Facts** navigation tab and select the ADF Business Components Facts tab.
2. Double-click the icon in the **DepartmentsView** row.
3. In the **Properties** table, in the **EmployeesView** row clear the **Visible** check box.
4. Click **OK**.

How to Set Alias for DepartmentsView and EmployeesView

To set alias for DepartmentsView and EmployeesView:

1. Select the **Facts** navigation tab and select the ADF Business Components Facts tab.
2. In the **Alias** column, replace **EmployeesView** with **Employee**.
3. In the **Alias** column, replace **DepartmentsView** with **Department.**

How to Add and Run the Outside Manager Ruleset

The sample code that runs the outside manager ruleset invokes the Decision Point with the view object set using the `setInputs` method. This invokes the decision function once, with all of the view object rows loaded in a `List`. Note that invoking the Decision Point this way is not scalable, because all of the view object rows must be loaded into memory at the same time, which can lead to OutOfMemory exceptions. Only use this invocation style when there are a small and known number of view object rows. You can also use a Decision Point with `setViewObject`. For more information, see How to Call a Decision Point with ADF Business Components Facts.

How to Add the Outside Manager Ruleset and Add a Decision Function

After the view objects are imported as facts, you can rename the ruleset and create the decision function for the application.

To rename the ruleset, select the **Ruleset_1** navigation tab in Rules Designer and then select the ruleset name and enter `Outside Manager Ruleset` to rename the ruleset.

To add a decision function:

1. Click the **Decision Functions** navigation tab.
2. In the Decision Functions area, click **Create...**. This displays the Edit Decision Function dialog.
3. Edit the decision function fields as follows:
   - Enter **Name** value `FindOutsideManagers`.
   - In the **Inputs** area, click the **Add Input** button and edit the input information as follows:
     – Click the **Fact Type** field and select **Employee** from the list.

  - Select the **List** check box.

    In this decision function you do not define any outputs because you use the `ActionType` API for taking action rather than producing output. For more information, see Understanding Oracle Business Rules Decision Point Action Type.

- In the **Rulesets & Decision Functions** area move the following items from the Available area to the Selected area, in the specified order:

  - **DecisionPointDictionary.Preprocessing**
  - **Outside Manager Ruleset**
  - **DecisionPointDictionary.Postprocessing**

4. Ensure that the items in the Selected area are in this order: DecisionPointDictionary.Preprocessing, Outside Manager Ruleset, DecisionPointDictionary.PostProcessing.

   If they are not, select an item and use the **Move Up** and **Move Down** buttons to correct the order.

5. Click **OK**.

Several warnings appear. These warnings are removed in later steps when you add rules to the ruleset.

How to Create the ActionType Java Implementation Class

To create the sample application and to modify the view object in a rule, you need to create a Java implementation class for abstract class `oracle.rules.sdk2.decisionpoint.ActionType`. All subclasses of `ActionType` must implement the abstract `exec` method.

To create the ActionType Java implementation class:

1. In Oracle JDeveloper, select the project named **Chapter10**.
2. In the Application Navigator, select the **Application Sources** folder.
3. Right-click and from the list select **New...**.
4. In the New Gallery, in the **Categories** area select **General**.
5. In the New Gallery, in the **Items** area select **Java Class**.
6. Click **OK**.
7. In the Create Java Class dialog, configure the following properties:

   - Enter the **Name** value `MessageAction`.
   - Enter the **Package** value `com.example`.
   - Enter the **Extends** value `oracle.rules.sdk2.decisionpoint.ActionType`.

8. Click **OK**.

   Oracle JDeveloper displays the Java Class.

9. Replace this code with the code shown in the example below showing actiontype java implementation.

```
package com.example;

import oracle.rules.sdk2.decisionpoint.ActionType;
import oracle.rules.sdk2.decisionpoint.DecisionPointInstance;

public class MessageAction extends ActionType {
    public MessageAction() {
        super();
    }

    public void exec(DecisionPointInstance decisionPointInstance) {
        System.out.println(message);
    }

    private String message = null;

    public void setMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

10. In the Application Navigator, right click the `MessageAction.java` and from the list select **Make**.

How to Import the Message Action Java Fact

You just created a new Java class and you need to add this class as a Java fact type in Rules Designer to use later when you create rules.

To create the Java fact type:

1. In Rules Designer, click the **Facts** navigation tab.
2. Select the **Java Facts** tab.
3. Click **Create...**.
4. In the Create Java Fact dialog, in the Classes area navigate in the tree and expand `com` and `example` to display the **MessageAction** check box.
5. Select the **MessageAction** check box.
6. Click **OK**. This adds the fact to the table.

How to Add the Find Managers Rule

You add the rule to find the managers that are in a different departments than their employees.

To add the find managers in different departments rule:

1. In Rules Designer, select the **Outside Manager Ruleset** tab.
2. Click **Add** and from the list select **Create Rule**.

3. Rename the rule by selecting the default rule name **Rule_1**. This displays a text entry area. You enter a name. For example, enter **Find managers in different department**. Press **Enter** to apply the name.

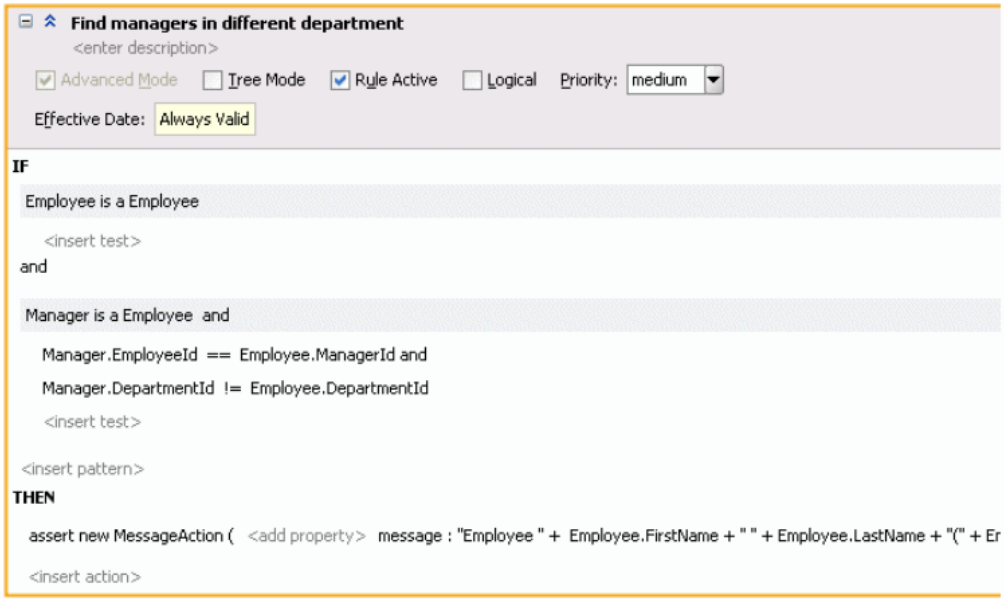
4. Click **Show Advanced Settings**. For more information, see How to Show and Hide Advanced Settings in a Rule or Decision Table.

5. In the rule select **Advanced Mode**.

6. Enter the rule as shown in Figure 10-6. The action for the rule shown in the **THEN** area is too long to show in the figure. The complete action that you build includes the following items:

```
IF Employee is EmployeeandManager is a Employee
andManager.EmployeeId == Employee.ManagerId
andManager.DepartmentId != Employee.DepartmentIDTHENassert
new MessageAction(<add property> message: "Employee " +
Employee.FirstName + " " + Employee.LastName + "(" +
Employee.EmployeeId + ")"+ " in dept " +
Employee.DepartmentId + " has manager outside of department,
" + Manager.FirstName + " " + Manager.LastName + "(" +
Manager.EmployeeId + ")" + " in dept " +
Manager.DepartmentId )
```

Figure 10-6 Find Managers in Different Departments Rule



How to Add the Outside Manager Finder Class

Add the outside manager finder class. This uses the Decision Point to execute a decision function.

To add the Outside Manager Finder Class:

1. Select the **Chapter10** project.

2. Right-click and select **New...**.

3. In the New Gallery, in the **Categories** area select **General**.
4. In the New Gallery, in the **Items** area select **Java Class**. Click **OK**.
5. In the **Name** field, enter `OutsideManagerFinder`. Click **OK**.
6. Replace the contents of this class with the code shown in example below showing Outside Manager Finder java class.

```
package com.example;

import java.util.ArrayList;

import oracle.jbo.ApplicationModule;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

import oracle.rules.rl.exceptions.RLException;
import oracle.rules.sdk2.decisionpoint.DecisionPoint;
import oracle.rules.sdk2.decisionpoint.DecisionPointBuilder;
import oracle.rules.sdk2.decisionpoint.DecisionPointInstance;
import oracle.rules.sdk2.exception.SDKException;
import oracle.rules.sdk2.repository.DictionaryFQN;

public class OutsideManagerFinder {
    private static final String AM_DEF = "com.example.AppModule";
    private static final String CONFIG = "AppModuleLocal";
    private static final String VO_NAME = "EmployeesView1";

    private static final DictionaryFQN DICT_FQN =
                  new DictionaryFQN("com.example", "Chapter10Rules");

    private static final String DF_NAME = "FindOutsideManagers";

    private DecisionPoint dp = null;

    public OutsideManagerFinder() {
        try {
            dp = new DecisionPointBuilder()
                              .with(DICT_FQN)
                              .with(DF_NAME)
                              .build();
        } catch (SDKException e) {
            System.err.println(e);
        }
    }

    public void run() {
        final ApplicationModule am =
                  Configuration.createRootApplicationModule(AM_DEF, CONFIG);
        final ViewObject vo = am.findViewObject(VO_NAME);
        final DecisionPointInstance point = dp.getInstance();
        point.setInputs(new ArrayList<Object>(){{ add(vo); }});
        try {
            point.invoke();
        } catch (RLException e) {
            System.err.println(e);
        } catch (SDKException e) {
            System.err.println(e);
        }
    }
```

```
    public static void main(String[] args) {
        OutsideManagerFinder omf = new OutsideManagerFinder();
        omf.run();
    }

}
```

How to Update ADF META INF for Local Dictionary Access

You need to update the `ADF-META-INF` file with MDS information for accessing the dictionary. You can use a local file with MDS to access the Oracle Business Rules dictionary. However, this procedure is not the usual dictionary access method with Oracle Business Rules in a production environment. For information on using a Decision Point to access a dictionary with MDS in a production environment, see What You Need to Know About Using Decision Point in a Production Environment.

To update ADF-META-INF:

1. In the Application Navigator, expand **Application Resources**.
2. Expand **Descriptors** and **ADF META-INF** folders.
3. Double-click **adf-config.xml** to open this file.
4. Click the **Source** tab to view the `adf-config.xml` source.
5. Add the MDS information to `adf-config.xml`, before the closing `</adf-config>` tag, as shown in the code example below:

```
   <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config version="11.1.1.000" xmlns="http://xmlns.oracle.com/mds/config">
      <persistence-config>
        <metadata-namespaces>
          <namespace metadata-store-usage="mstore-usage_1" path="/"/>
        </metadata-namespaces>
        <metadata-store-usages>
          <metadata-store-usage id="mstore-usage_1">
            <metadata-store class-
name="oracle.mds.persistence.stores.file.FileMetadataStore">
              <property name="metadata-path"
                        value="C:\jdevinstance\mywork\Chapter10\.adf\"/>
            </metadata-store>
          </metadata-store-usage>
        </metadata-store-usages>
      </persistence-config>
    </mds-config>
  </adf-mds-config>
```

6. In the `<property>` element with the attribute `metadata-path`, change the path to match `.adf` directory in the application on your system.

How to Copy Definitions/Dictionary to MDS Accessible Location

To copy definitions/dictionary to MDS accessible location:

1. In a file system navigator, outside of Oracle JDeveloper navigate to the **Chapter10** application, and in the **Chapter10** project, in the **src** folder select and copy the **com** folder. Or if you want to copy dictionary to MDS accessible location, copy the **oracle** directory that contains the Oracle Business Rules dictionary.

2. In the application directory for **Chapter10**, above the **Chapter10** project, navigate to the **.adf** directory.
3. Copy the **com** folder to this directory. or copy the **oracle** folder to this directory.

How to Build and Run the Project to Check the Outside Manager Finder

You can build and test the project by running the find managers with employees in different departments rule.

Build the OutsideManagerFinder configuration:

1. From the dropdown menu next to **Run** button, select **Manage Run Configurations...**.
2. In the Project Properties dialog, click **New...**.
3. In the Create Run Configuration dialog, enter a name. For example, enter `OutsideManagerFinder`.
4. Click **OK**.
5. With **OutsideManagerFinder** selected, click **Edit...**.
6. In the **Default Run Target** field, click **Browse...**.
7. Select **OutsideManagerFinder.java** from the `src\com\example` folder.
8. Click **Open**.
9. In the Edit Run Configuration dialog, click **OK**.
10. In the Project Properties dialog, click **OK**.

To run the project, select **OutsideManagerFinder** in the dropdown menu next to the **Run** project button. Running this configuration generates output, as shown in example below:

```
Emp Shelley Higgins(205) in dept 110 manager outside of department, Neena
Kochhar(101) in dept 90
Emp Hermann Baer(204) in dept 70 manager outside of department, Neena Kochhar(101)
in dept 90
Emp Susan Mavris(203) in dept 40 manager outside of department, Neena Kochhar(101)
in dept 90
Emp Michael Hartstein(201) in dept 20 manager outside of department, Steven
King(100) in dept 90
Emp Jennifer Whalen(200) in dept 10 manager outside of department, Neena
Kochhar(101) in dept 90
Emp Kimberely Grant(178) in dept null manager outside of department, Eleni
Zlotkey(149) in dept 80
Emp Eleni Zlotkey(149) in dept 80 manager outside of department, Steven King(100) in
dept 90
Emp Gerald Cambrault(148) in dept 80 manager outside of department, Steven King(100)
in dept 90
Emp Alberto Errazuriz(147) in dept 80 manager outside of department, Steven
King(100) in dept 90
Emp Karen Partners(146) in dept 80 manager outside of department, Steven King(100)
in dept 90
Emp John Russell(145) in dept 80 manager outside of department, Steven King(100) in
dept 90
Emp Kevin Mourgos(124) in dept 50 manager outside of department, Steven King(100) in
```

```
dept 90
Emp Shanta Vollman(123) in dept 50 manager outside of department, Steven King(100)
in dept 90
Emp Payam Kaufling(122) in dept 50 manager outside of department, Steven King(100)
in dept 90
Emp Adam Fripp(121) in dept 50 manager outside of department, Steven King(100) in
dept 90
Emp Matthew Weiss(120) in dept 50 manager outside of department, Steven King(100) in
dept 90
Emp Den Raphaely(114) in dept 30 manager outside of department, Steven King(100) in
dept 90
Emp Nancy Greenberg(108) in dept 100 manager outside of department, Neena
Kochhar(101) in dept 90
Emp Alexander Hunold(103) in dept 60 manager outside of department, Lex De Haan(102)
in dept 90
```

How to Add and Run the Department Manager Ruleset

The sample code that runs the department manager ruleset invokes the Decision Point with the view object set using the `setViewObject` method. This invokes the decision function once for each row in the view object. All decision function calls occur in the same RuleSession. Between decision function calls, the RuleSession preserves all state from the previous decision function call. Thus, any objects asserted during the previous call remain in working memory for the next call unless they are explicitly retracted by rulesets that you supply. When the state is maintained, you can retract all facts or selectively retract facts between calls by running a ruleset with rules that use the retract action. This ruleset is run as part of the same decision function that you use with the Decision Point. The retract all employees ruleset demonstrates retracting these facts. For more information, see How to Call a Decision Point with ADF Business Components Facts.

How to Add the Department Manager Finder Ruleset

You now add the department manager finder ruleset.

To add the department manager finder ruleset:

1. In Rules Designer, click **Create Ruleset...**.
2. In the Create Ruleset dialog, in the **Name** field enter `Department Manager Finder Ruleset`.
3. Click **OK**.

How to Add the Find Rule in the Department Manager Finder Ruleset

Next you add the **Find** rule to find department managers. This rule demonstrates the use of **Tree Mode** rules with Oracle ADF Business Components fact types.

Add department manager finder rule:

1. In Rules Designer select the Department Manager Finder Ruleset.
2. In the dropdown menu next to the **Add** button, click **Create Rule**.
3. Change the rule name by selecting the name **Rule_1**, and entering `Find`.
4. Click **Show Advanced Settings**. For more information, see How to Show and Hide Advanced Settings in a Rule or Decision Table.

5. In the rule, select **Tree Mode**.
6. Enter the **Find** rule tests and actions. The following shows the complete text of this rule:

```
ROOT: Employee
IF
Employee/DepartmentsView.ManagerID == Employee.EmployeeID
THEN
retract Employee
assert new MessageAction ( <add property> message: Employee.FirstName + " " +
Employee.LastName + " is the manager of dept "+ Employee/
DepartmentsView.DepartmentName )
```

How to Add Retract Employees Ruleset

You add a ruleset to retract the employee fact type instances. This ensures that the Employee fact type is removed between invocations of the decision function.

To add the retract employee ruleset:

1. Add the Retract Employees Ruleset.
2. In the Retract Employees Ruleset, add a rule and name it **Retract all employees**.

How to Add the Find Department Managers Decision Function

Now you create the decision function for the department manager finder ruleset. You use this decision function to execute the ruleset from a Decision Point.

To add a decision function for department manager finder ruleset:

1. Click the **Decision Functions** navigation tab.
2. In the Decision Functions area, click **Create...**. This displays the Edit Decision Function dialog.
3. Update the decision function fields as follows:
   - Enter **Name** value `FindDepartmentManagers`.
   - In the **Inputs** area, click the **Add Input** and edit the input information as follows:
     - Click the **Fact Type** field and select **Employee** from the list.
     - Select the **Tree** check box.

     In this decision function you do not define any outputs, because you use the `ActionType` API for taking action rather than producing output.
   - In the **Rulesets & Decision Functions** area, move the following items from the **Available** area to the **Selected** area, in the specified order:
     - **DecisionPointDictionary.Preprocessing**
     - **Department Manager Finder Ruleset**
     - **Retract Employees**

- **DecisionPointDictionary.Postprocessing**

4. Ensure that the items in the **Selected** area are in this order: DecisionPointDictionary.Preprocessing, Department Manager Finder Ruleset, Retract Employees, and DecisionPointDicitonary.Postprocessing.

   If they are not, select an item and use the **Move Up** and **Move Down** buttons to correct the order.

5. Click **OK**.

How to Add the Department Manager Finder Java Class

Add the department manager finder class. This class include the code with the Decision Point that executes the decision function.

Add the department manager finder class:

1. In the Application Navigator, select the **Chapter10** project.
2. Right-click and select **New...**.
3. In the New Gallery, in the **Categories** area select **General**.
4. In the New Gallery, in the **Items** area, select **Java Class**. Click **OK**.
5. In the **Name** field, enter `DeptManagerFinder`. Click **OK**.
6. Replace the contents of this class with the code example shown below:

```
package com.example;

import oracle.jbo.ApplicationModule;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.server.DBTransactionImpl2;

import oracle.rules.rl.exceptions.RLException;
import oracle.rules.sdk2.decisionpoint.DecisionPoint;
import oracle.rules.sdk2.decisionpoint.DecisionPointBuilder;
import oracle.rules.sdk2.decisionpoint.DecisionPointInstance;
import oracle.rules.sdk2.exception.SDKException;
import oracle.rules.sdk2.repository.DictionaryFQN;

public class DeptManagerFinder {
    private static final String AM_DEF = "com.example.AppModule";
    private static final String CONFIG = "AppModuleLocal";
    private static final String VO_NAME = "EmployeesView1";

    private static final String DF_NAME = "FindDepartmentManagers";

    private static final DictionaryFQN DICT_FQN =
                    new DictionaryFQN("com.example", "Chapter10Rules");

    private DecisionPoint dp = null;

    public DeptManagerFinder() {

        try {
            dp = new DecisionPointBuilder()
                            .with(DICT_FQN)
```

```
                            .with(DF_NAME)
                            .build();
        } catch (SDKException e) {
            System.err.println(e);
        }
    }

    public void run() {
        final ApplicationModule am =
                Configuration.createRootApplicationModule(AM_DEF, CONFIG);
        final ViewObject vo = am.findViewObject(VO_NAME);
        final DecisionPointInstance point = dp.getInstance();

        point.setTransaction((DBTransactionImpl2)am.getTransaction());
        point.setAutoCommit(true);
        point.setViewObject(vo);
        try {
            point.invoke();
        } catch (RLException e) {
            System.err.println(e);
        } catch (SDKException e) {
            System.err.println(e);
        }
    }

    public static void main(String[] args) {
        new DeptManagerFinder().run();
    }
}
```

How to Copy the Dictionary to an MDS Accessible Location

Copy the updated dictionary to an MDS accessible location.

Copy dictionary to MDS accessible location:

1. In a file system navigator, outside of Oracle JDeveloper, navigate to the **Chapter10** application, and project and copy the **oracle** directory that contains the dictionary.
2. In the application directory for **Chapter10**, above the **Chapter10** project, navigate to the `.adf` directory.
3. Copy the **oracle** folder to this directory.

How to Build and Run the Project to Check the Find Managers Rule

You can build and test the project to execute the department manager finder ruleset.

Build the project:

1. From the dropdown menu next to **Run** button, select **Manage Run Configurations...**.
2. In the Project Properties dialog, click **New...**.
3. In the Create Run Configuration dialog, enter the name. For example, enter `DeptManagerFinder`.
4. In the **Copy Settings From** field, enter **Default**.

5. Click **OK**.
6. With **DeptManagerFinder** selected, click **Edit...**.
7. In the **Default Run Target** field, click **Browse...**.
8. Select **DeptManagerFinder.java** from the `src\com\example` directory and click **Open**.
9. In the Edit Run Configuration dialog, click **OK**.
10. In the Project Properties dialog, click **OK**.

To run the project, select **DeptManager Finder** in the menu, next to the **Run** project button. Running the decision point generates output, as shown in code example below:

```
Michael Hartstein is the manager of dept Marketing
John Russell is the manager of dept Sales
Adam Fripp is the manager of dept Shipping
Den Raphaely is the manager of dept Purchasing
Alexander Hunold is the manager of dept IT
Shelley Higgins is the manager of dept Accounting
Hermann Baer is the manager of dept Public Relations
Susan Mavris is the manager of dept Human Resources
Jennifer Whalen is the manager of dept Administration
Nancy Greenberg is the manager of dept Finance
Steven King is the manager of dept Executive
Shelley Higgins is the manager of dept Accounting
Hermann Baer is the manager of dept Public Relations
Susan Mavris is the manager of dept Human Resources
Jennifer Whalen is the manager of dept Administration
Nancy Greenberg is the manager of dept Finance
Alexander Hunold is the manager of dept IT
Alexander Hunold is the manager of dept IT
Nancy Greenberg is the manager of dept Finance
Den Raphaely is the manager of dept Purchasing
Adam Fripp is the manager of dept Shipping
John Russell is the manager of dept Sales
Jennifer Whalen is the manager of dept Administration
Michael Hartstein is the manager of dept Marketing
Susan Mavris is the manager of dept Human Resources
Hermann Baer is the manager of dept Public Relations
Shelley Higgins is the manager of dept Accounting
```

When you see duplicate entries in the output, when working with tree mode rules in this example, the duplicate entries are due to multiple rule firings on the same data in a different part of the view object graph.

How to Add and Run the Raises and Retract Employees Rulesets

The sample code that runs the raises ruleset invokes the Decision Point by specifying the view object using the `setViewObject` method. This invokes the decision function once for each row in the view object. The retract employees ruleset retracts all instances of `Employee` asserted for each call, so that they do not remain in working memory between calls to the decision function. The action type shown in How to Create the Raise ActionType Java Implementation Class shows how to change the `ViewRowImpl` attribute values with a `ActionType`.

For more information, see How to Call a Decision Point with ADF Business Components Facts.

How to Add the Raises Ruleset

You now add the raises ruleset.

To add the raises ruleset:

1. In Rules Designer, click **Create Ruleset...**.
2. In the Create Ruleset dialog, in the **Name** field enter `Raises Ruleset`.
3. Click **OK**.

How to Create the Raise ActionType Java Implementation Class

To create this part of the sample application and to modify the view object in the raises rule, you need to create a Java implementation class for the abstract class `oracle.rules.sdk2.decisionpoint.ActionType`. All subclasses of `ActionType` must implement the abstract `exec` method.

To create the raise ActionType Java implementation class:

1. In Oracle JDeveloper, select the project named **Chapter10**.
2. In the Application Navigator, select the **Application Sources** folder.
3. Right-click and from the list select **New...**.
4. In the New Gallery, in the **Categories** area select **General**.
5. In the New Gallery, in the **Items** area select **Java Class**. Click **OK**.
6. In the Create Java Class dialog, configure the following properties:
   - Enter the **Name** value `RaiseAction`.
   - Enter the **Package** value `com.example`.
   - Enter the **Extends** value `oracle.rules.sdk2.decisionpoint.ActionType`.
7. Click **OK**.

   Oracle JDeveloper displays the Java Class.
8. Replace this code with the code example showing `actiontype` java implementation as shown below:

```
package com.example;

import oracle.jbo.domain.Number;

import oracle.rules.sdk2.decisionpoint.ActionType;
import oracle.rules.sdk2.decisionpoint.DecisionPointInstance;

public class RaiseAction extends ActionType {
    private double raisePercent;

    public void exec(DecisionPointInstance dpi) {
        Number salary = (Number)getViewRowImpl().getAttribute("Salary");
```

```
        salary = (Number)salary.multiply(1.0d + getRaisePercent()).scale(100,2, new
boolean[]{false});
        dpi.addResult("raise for " + this.getViewRowImpl().getAttribute("EmployeeId"),
                      getRaisePercent() + "=>" + salary );
        getViewRowImpl().setAttribute("Salary", salary);
    }

    public void setRaisePercent(double raisePercent) {
        this.raisePercent = raisePercent;
    }

    public double getRaisePercent() {
        return raisePercent;
    }
}
```

9. In the Application Navigator, right click the `RaiseAction.java` and from the list select **Make**.

How to Import the Raise Action Java Fact

You just created a new Java class. You import this class as a Java fact type in Rules Designer to use later when you create rules.

To create the Java fact type:

1. In Rules Designer, select the `ManagerRules.rules` dictionary.
2. Click the **Facts** navigation tab and select the **Java Facts** tab.
3. Click **Create...**.
4. In the Create Java Fact dialog, in the Classes area navigate in the tree and expand `com` and `example` to display the **RaiseAction** check box.
5. Select the **RaiseAction** check box.
6. Click **OK**.

   This adds the Raise Action fact type to the **Java Facts** table.

How to Add the 12 Year Raise Rule

This rule shows how to use action types to update database entries.

To add 12 year raise rule:

1. In Rules Designer in the Raises Ruleset, click **Create Rule**.
2. Change the rule name by selecting **Rule_1** and entering the value: `Longer than 12 years`.
3. Click **Show Advanced Settings**. For more information, see How to Show and Hide Advanced Settings in a Rule or Decision Table.
4. Select **Advanced Mode**.
5. Enter the 12 year raise rules, as shown in this example:

```
IF
Employee is Employee
and
```

```
CurrentDate is a CurrentDate and
Duration.years between(Employee.HireDate, CurrentDate.date) >=12
THEN
assert new RaiseAction( <add property> raisePercent: .03,
viewRowImple:Employee.ViewRowImpl)
retract Employee
```

How to Add the Employee Raises Decision Function

Now create the decision function for the employee raises and the retract all employees rulesets.

To add a decision function:

1. Click the **Decision Functions** navigation tab.
2. In the Decision Functions area, click **Create...**. This displays the Edit Decision Function dialog.
3. Update the decision function fields as follows:
   - Enter **Name** value `EmployeeRaises`.
   - In the **Inputs** area, click the **Add Input** and edit the input information as follows:
     – Click the **Fact Type** field and select **Employee** from the list.

     In this decision function you do not define any outputs, because you use the `ActionType` API for taking action rather than producing output.
   - In the **Rulesets & Decision Functions** area, move the following items from the **Available** area to the **Selected** area, in the specified order.
     – **DecisionPointDictionary.Preprocessing**
     – **Raises Ruleset**
     – **Retract Employees Ruleset**
     – **DecisionPointDictionary.Postprocessing**
4. Ensure that the items in the **Selected** area are in this order: DecisionPointDicitonary.Preprocessing, Raises Ruleset, Retract Employees Ruleset, DecisionPointDictionary.PostProcessing.

   If they are not, select an item and use the **Move Up** and **Move Down** buttons to correct the order.
5. Click **OK**.

How to Add the Employee Raises Java Class

Add the employee raises class. This executes the decision function.

To add the employee raises class:

1. Select the **Chapter10** project.

2. Right-click and select **New...**.
3. In the New Gallery, in the **Categories** area select **General**.
4. In the New Gallery, in the **Items** area, select **Java Class**. Click **OK**.
5. In the **Name** field, enter `EmployeeRaises`. Click **OK**.
6. Replace the contents of this class with the `deptmanagerfinder` class code shown below:

```
package com.example;

import oracle.jbo.ApplicationModule;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.server.DBTransactionImpl2;

import oracle.rules.rl.exceptions.RLException;
import oracle.rules.sdk2.decisionpoint.DecisionPoint;
import oracle.rules.sdk2.decisionpoint.DecisionPointBuilder;
import oracle.rules.sdk2.decisionpoint.DecisionPointInstance;
import oracle.rules.sdk2.exception.SDKException;
import oracle.rules.sdk2.repository.DictionaryFQN;


public class EmployeeRaises {
    private static final String AM_DEF = "com.example.AppModule";
    private static final String CONFIG = "AppModuleLocal";
    private static final String VO_NAME = "EmployeesView1";
    private static final String DF_NAME = "EmployeeRaises";

    private static final DictionaryFQN DICT_FQN =
        new DictionaryFQN("com.example", "Chapter10Rules");

    private DecisionPoint dp = null;

    public EmployeeRaises() {

        try {
            dp = new DecisionPointBuilder()
                    .with(DICT_FQN)
                    .with(DF_NAME)
                    .build();
        } catch (SDKException e) {
            System.err.println(e);
        }
    }

    public void run() {
        final ApplicationModule am =
            Configuration.createRootApplicationModule(AM_DEF, CONFIG);
        final ViewObject vo = am.findViewObject(VO_NAME);
        final DecisionPointInstance point = dp.getInstance();

        point.setTransaction((DBTransactionImpl2)am.getTransaction());
        point.setAutoCommit(true);
        point.setViewObject(vo);
        try {
            point.invoke();
        } catch (RLException e) {
```

```
            System.err.println(e);
        } catch (SDKException e) {
            System.err.println(e);
        }

        for (DecisionPoint.NamedValue result : point.getResults()){
            System.out.println(result.getName() +  " " + result.getValue());
        }

    }

    public static void main(String[] args) {
        new EmployeeRaises().run();
    }
}
```

How to Copy Dictionary to MDS Accessible Location

Copy the updated dictionary to the MDS accessible location.

To copy dictionary to MDS accessible location:

1. In a file system navigator, outside of Oracle JDeveloper, navigate to the **Chapter10** folder and the **Chapter10** project and copy the **oracle** directory that contains the dictionary.
2. In the application directory for **Chapter10**, above the **Chapter10** project, navigate to the `.adf` directory.
3. Copy the **oracle** folder to this directory.

How to Build and Run the Project to Check the Raises Rule

You can build and test the project by running employee raises ruleset.

To build the project:

1. From the dropdown menu next to **Run** button, select **Manage Run Configurations...**.
2. In the Project Properties dialog, click **New...**.
3. In the Create Run Configuration dialog, enter the name. For example, enter `EmployeeRaises`.
4. In the Copy Settings From field, enter **Default**. Click **OK**.
5. With **EmployeeRaises** selected, click **Edit...**.
6. In the **Default Run Target** field, click **Browse...**.
7. Select **EmployeeRaises.java** from the `src\com\example` folder. Click **Open**.
8. In the Edit Run Configuration dialog, click **OK**.
9. In the Project Properties dialog, click **OK**.

To run the project, select **EmployeeRaises** in the menu, next to the **Run** project button. Oracle JDeveloper displays the output as shown in example below:

```
raise for 100 0.03=>81.7
raise for 101 0.03=>1872.46
raise for 102 0.03=>60596.78
raise for 103 0.03=>31146.26
raise for 104 0.03=>20159.43
raise for 108 0.03=>35822.68
raise for 109 0.03=>26084.5
raise for 114 0.03=>27500.92
raise for 115 0.03=>7524.5
raise for 120 0.03=>16262.34
raise for 121 0.03=>16183.41
raise for 122 0.03=>15591.35
raise for 131 0.03=>3671.33
raise for 133 0.03=>4567.98
raise for 137 0.03=>4838.1
raise for 141 0.03=>4703.71
raise for 142 0.03=>4044.79
raise for 145 0.03=>17734.79
raise for 146 0.03=>17101.39
raise for 147 0.03=>15201.23
raise for 150 0.03=>12667.7
raise for 151 0.03=>12034.32
raise for 156 0.03=>13047.73
raise for 157 0.03=>12395.35
raise for 158 0.03=>11400.93
raise for 159 0.03=>10134.16
raise for 168 0.03=>14567.86
raise for 174 0.03=>13934.48
raise for 175 0.03=>11147.58
raise for 184 0.03=>5480.03
raise for 185 0.03=>5193.76
raise for 192 0.03=>5219.1
raise for 193 0.03=>4940.41
raise for 200 0.03=>5740.99
raise for 201 0.03=>16962.05
raise for 203 0.03=>8481.03
raise for 204 0.03=>13047.73
raise for 205 0.03=>15657.27
raise for 206 0.03=>10829.62
```

11

Working with Decision Components in SOA Applications

This chapter discusses the Decision Components that support Oracle Business Rules. It also covers how to use a Decision Component as a mechanism for publishing rules and rulesets as a reusable service that can be invoked from multiple business processes.

A Decision Component is an SCA component that can be used within a composite and wired to a BPEL component. Apart from that, Decision Components are used for the dynamic routing capability of Mediator and Advanced Routing Rules in Human Workflow.

This chapter includes the following sections:

Introduction to Decision Components

A Decision Component is a web service that wraps a rule session to the underlying decision function.

A Decision Component consists of the following:

- Rules or Decision Tables that are evaluated using the Rules Engine. These are defined using Rules Designer and stored in a business rules dictionary.
- Metadata that describes facts required for specific rules to be evaluated. Rulesets that contain rules or Decision Tables are exposed as a service with facts that are input and output. These facts must be exposed through XSD definitions.

  For example, a credit rating ruleset may expect a customer ID and previous loan history as facts, but a pension payment ruleset may expect a value with the years of employee service, salary, and age as facts.

  For more information, see Working with Decision Component Metadata.
- A web service wraps the input, output, and the call to the underlying rule engine.

  This service lets business processes assert and retract facts as part of the process. In some cases, all facts can be asserted from the business process as one unit. In other cases, the business process can incrementally assert facts and eventually consult the rule engine for inferences. Therefore, the service has to support both stateless and stateful interactions.

  You can create a variety of such business rules service components.

For more information, see *Developing SOA Applications with Oracle SOA Suite*.

Working with a Decision Component

Using Oracle JDeveloper with Rules Designer these tools automatically generate all required metadata and WSDL operations.

The Decision Component can be integrated into a SOA composite application in the following ways:

- Create a Decision Component as a standalone component in the SOA Composite Editor. In this scenario, the Decision Service is exposed on the composite level and thus can be invoked from any web service client.

  For more information, see Getting Started with Oracle Business Rules in *Developing SOA Applications with Oracle SOA Suite*.

- Create a Decision Component in the SOA Composite Editor that you later associate with a BPEL process. In this scenario the Decision Service is not exposed on the composite level. However it can be wired to any other component within the composite, such as BPEL, Oracle Mediator, and Oracle Human Workflow.

  For more information, see Getting Started with Oracle Business Rules in *Developing SOA Applications with Oracle SOA Suite*.

- Create a Decision Component within the Human Task editor of a human task component.

This integration provides the following benefits:

- Dynamic processing: provides for intelligent routing, validation of policies within a process, and constraint checks.

- Integration with ad hoc human tasks: provides policy-based task assignment, various escalation policies, and load balancing of tasks.

Working with Decision Component Metadata

A Decision Component is defined by the following files:

- Decision Service Metadata (.decs) File

- SCA Component Type (.componentType) File

- Decision Component Entry in composite.xml

Typically, Oracle JDeveloper generates and maintains these files.

- Decision Service Metadata (.decs) File

  Every Decision Component within a composite comprises one business rule metadata file. The business rule metadata file provides information about the location of the component business rule dictionary and the Decision Services exposed by the Decision Component.

  One Decision Component might expose one or more Decision Services. For example, a CreditRating Decision Component might expose two services, CheckEligibility and CalculateCreditRating.Oracle Fusion Middleware 11g Release 1 (11.1.1) onwards, the Decision Service metadata comprises of the decision function name that is exposed as a web service. For projects that are migrated from

older releases of Oracle SOA Suite, the Decision Service metadata typically has more information depending on the interaction pattern used in 10.1.3.x.

The business rule metadata file (*business_rule_name*.decs) defines the contract between the components involved in the interaction of the business rule with the design time and back-end Oracle Rules Engine.

This file is in the **SOA Content** area of the Application Navigator in Oracle JDeveloper for your SOA composite application. Table 11-1 describes the top-level elements in the Decision service .decs file.

Table 11-1 Decision Metadata File (.decs) Top-level Elements

Element	Description

ruleEngineProvider	The *business_rule_name*.decs file ruleEngineProvider element includes details about the rule dictionary to use:
	
	`<ruleEngineProvider name="OracleRulesSDK" provider="Oracle_11.0.0.0.0">`
	`  <repository type="SCA-Archive">`
	`  <path>`**`AutoLoanComposite/oracle/rules/AutoLoanRules.rules`**`</path>`
	`  </repository>`
	`</ruleEngineProvider>`
	
	The repository type is set to SCA-Archive for Decision Components. This indicates that the rule dictionary is located in the service component architecture archive. The path is relative and interpreted differently by the following:
	– Design time — The path is prefixed with Oramds:/. Metadata service (MDS) APIs open the rule dictionary. Therefore, the full path to the dictionary is as follows:
	
	`Oramds:/AutoLoanComposite/oracle/rules/AutoLoanRules.rules`
	– Runtime (business rule service engine) — The business rule service engine uses the Oracle Business Rules SDK RuleRepository API to open the rule dictionary located in MDS. The composite name prefix, for example (AutoLoanComposite) is removed from the path and the metadata manager assumes the existence of oracle/rules/AutoLoanRules.rules relative to the composite home directory.

Table 11-1 (Cont.) Decision Metadata File (.decs) Top-level Elements

Element	Description
`decisionService`	A Decision service is a web service (or SOA) enabler of business rules. It is a service in the sense of a web service, thus opening the world of business rules to service-oriented architectures (SOA). In 12*c* (12.2.1), a Decision service consists of metadata and a WSDL contract for the service. The *`business_rule_name`*`.decs` file `decisionService` element defines the metadata that describes business rules exposed as a web service. In general, a Decision service includes the following elements: – Target namespace – Reference to the back-end Oracle Rules Engine (this is the link to the rule dictionary). Note that `OracleRulesSDK` is the reference name that matches the name of the Oracle Rules Engine provider in `ruleEngineProvider` element. – Name (`CreditRatingService` in the following example) – Additional information about the dictionary name and ruleset to use – List of supported operations (patterns) Apart from the operations (patterns), the parameter types (or fact types) of operations are specified (and validated later at runtime). Therefore, every Decision service exposes a strongly-typed contract.

- SCA Component Type (.componentType) File

  An SCA `business_rule_name.componentType` file is included with each Decision Component. This file lists the services exposed by the business rules service component. In the following sample, two services are exposed: `CreditRatingService` and `LoanAdvisorService`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Generated by Oracle SOA Modeler version 1.0 at [5/24/07 9:27 AM]. -->
<componentType xmlns="http://xmlns.oracle.com/sca/1.0">
  <service name="CreditRatingService">
    <interface.wsdl
interface="http://xmlns.oracle.com/creditrating/Rating#wsdl.interface(IDecisionSer
vice)"/>
  </service>
  <service name="LoanAdvisorService">
    <interface.wsdl
interface="http://xmlns.oracle.com/loanoffer/Advisor#wsdl.interface(IDecisionServi
ce)"/>
  </service>
</componentType>
```

- Decision Component Entry in composite.xml

  An entry in `composite.xml` is created for a Decision Component. For example,

```
<component name="OracleRules1">
     <implementation.decision src="OracleRules1.decs"/>
</component>
```

  The business rules service engine uses the information from this implementation type to process requests for the Service Engine. From an SCA perspective, a Decision Component is a new "implementation type".

Working with Decision Components that Expose a Decision Function

You can use a decision service to expose an Oracle Business Rules Decision Function as a service. A Decision Function is a function you use to call rules from a Java EE application or from another component.

The code example below shows a *`business_rule_name`*`.decs` file `decisionServices` element that defines the metadata for an Oracle Business Rules Decision Function exposed as a service.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<decisionServices xmlns="http://xmlns.oracle.com/bpel/rules" name="PurchaseItems">
    <ruleEngineProvider name="OracleRulesSDK" provider="Oracle_11.0.0.0.0">
        <repository type="SCA-Archive">
            <path>PurchasingSampleProject/oracle/rules/com/example/PurchaseItems.rules</path>
        </repository>
    </ruleEngineProvider>
    <decisionService targetNamespace="http://xmlns.oracle.com/PurchaseItems/
PurchaseItems_DecisionService_ValidatePurchasesDF"
ruleEngineProviderReference="OracleRulesSDK"
name="PurchaseItems_DecisionService_ValidatePurchasesDF">
        <catalog>PurchaseItems</catalog>
        <pattern name="CallFunctionStateless">
            <arguments>
                <call>com.example.PurchaseItems.ValidatePurchasesDF</call>
            </arguments>
        </pattern>
        <pattern name="CallFunctionStateful">
            <arguments>
                <call>com.example.PurchaseItems.ValidatePurchasesDF</call>
            </arguments>
        </pattern>
    </decisionService>
</decisionServices>
```

In this case, the decision function `ValidatePurchasesDF` itself is specified entirely in the `PurchaseItems.rules` file.

For more information, see Working with Decision Functions.

Using Stateful Interactions with a Decision Component

To provide a stateful decision service you create a decision function and specify that the decision function is not stateless. To do so, clear the **Stateless** check box in a decision function.

Note the following details about stateful interactions with a Decision Component (also see Figure 11-2):

- Rule sessions from the cache and those from the pool are mutually exclusive:
  - The rule session pool is for simple, stateless interactions only.
  - The rule session cache keeps the state of a rule session across Decision service requests.

What You Need to Know About Stateful Interactions with Decision Components

A Decision Component running in a Business Rules service engine supports either stateful or stateless operation. The **Reset Session** (stateless) check box in the Create Business Rules dialog provides support for these two modes of operation.

When the **Reset Session** (stateless) check box selected, this indicates stateless operation.

When **Reset Session** (stateless) check box is cleared, the underlying Oracle Business Rules object is kept in memory of the Business Rules service engine at a separate location (so that it is not given back to the Rule Session Pool when the operation is finished). Only use stateful operation if you know you need this option (some errors can occur at runtime when using stateful operation and these errors could use a significant amount of service engine memory).

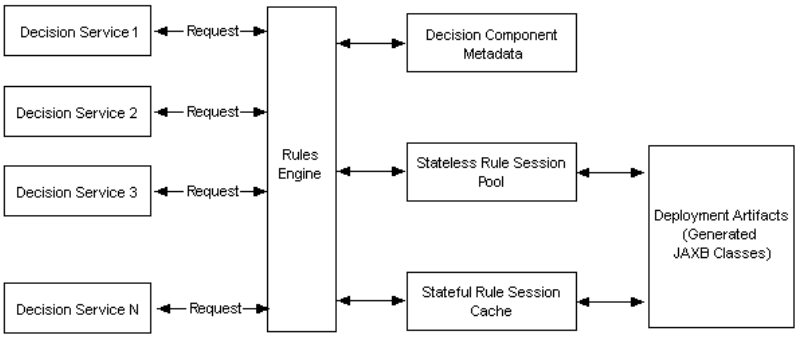
When **Reset Session** (stateless) check box is cleared, a subsequent use of the Decision Component reuses the cached RuleSession object, with all its state information from the `callFunctionStateful` invocation, and then releases it back to the Rule Session pool after the `callFunctionStateless` operation is finished.

Decision Service Architecture

A Decision service consists only of the service description. All other artifacts are shared within a Decision Component.

This is shown in Figure 11-1:

Figure 11-1 Decision Service Architecture



The heart of runtime is the decision service cache, which is organized in a tree structure. Every Decision Component owns a subtree of that cache (depending on the composite distinguished name (DN), component, and service name). In this regard, decision services of a Decision Component share the following data:

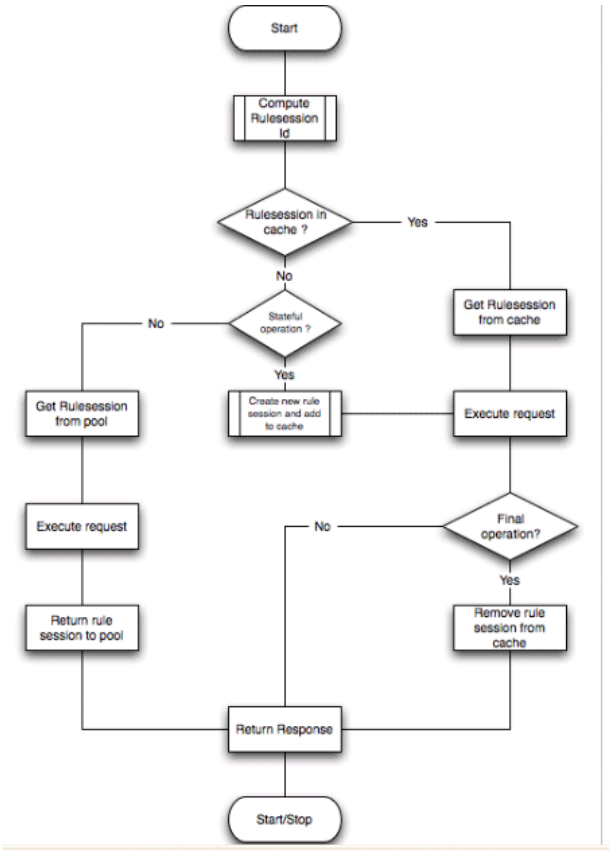
- Metadata of the Decision Component
  - Fact type metadata.
  - Function metadata.
  - Ruleset metadata.
- Rule session pool
  - One rule session pool is created per Decision Component.
  - The rule sessions in the pool are pre-initialized with the data model Oracle RL and the ruleset Oracle RL already executed.
  - New rule sessions are created on demand.
  - Rule sessions can be reused for a configurable number of times.
  - The initial size of the rule session pool is configurable.

- Stateful rule session cache
  - A special cache is maintained for stateful rule sessions.

    For more information, see Using Stateful Interactions with a Decision Component.
- Deployment artifacts
  - Decision Component deployment can end up in class generation for JAXB fact types. The classes can be shared across the composite.

Figure 11-2 shows how both stateless and stateful rule sessions interact with the rule session pool and how stateful rule sessions interact with the stateful rule session cache during a decision service request.

Figure 11-2 Stateless and Stateful Rule Session Usage for a Decision Service Request



12

Using Oracle SOA Composer with Oracle Business Rules at Runtime

This chapter describes how to use the Oracle SOA Composer application to work with a deployed dictionary and tasks that are part of a SOA composite application.

The chapter includes the following sections:

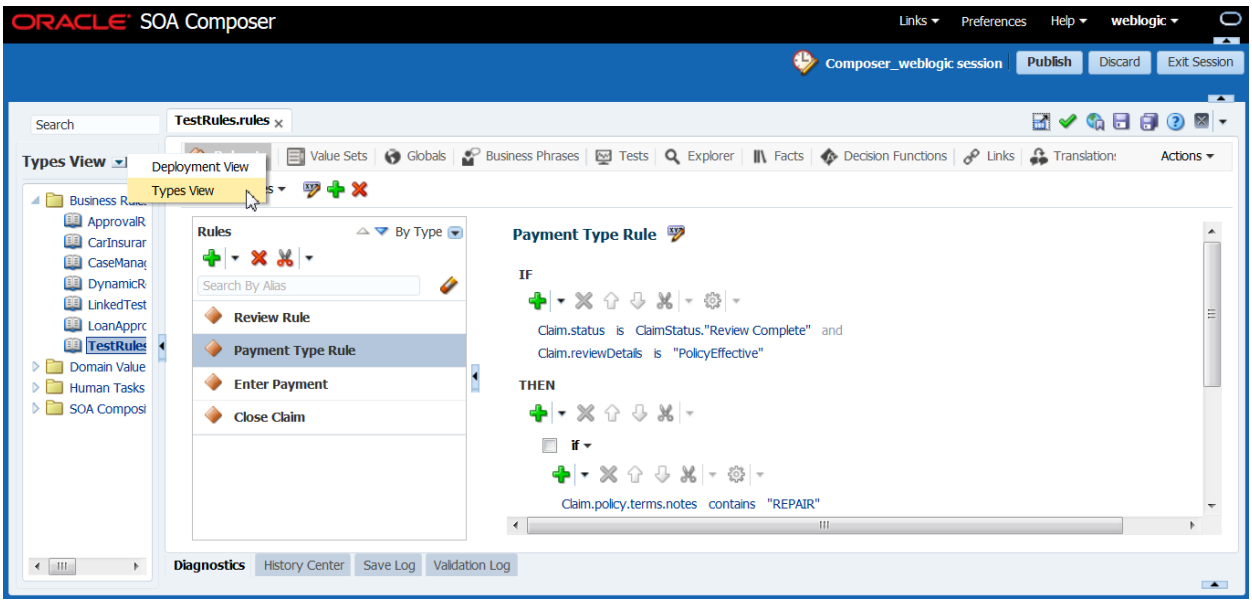
Introduction to Oracle SOA Composer

Oracle SOA Composer is a web-based application that enables you to work with Oracle Business Rules dictionaries and tasks for deployed applications. Oracle SOA Composer accesses a dictionary or a task in an MDS repository.

Oracle SOA Composer supports viewing and editing different types of metadata artifacts, such as DVM documents, SOA composites, and Oracle Business Rules dictionaries. You can view the different types of metadata by Types View or Deployment View, as shown in Figure 12-1.

The **Deployment View** is the default. Choose the **Types View** to see artifacts listed by type: **Business Rules, Domain Value Maps, Human Tasks,** or **SOA Composites.**

Figure 12-1 Oracle SOA Composer Types View



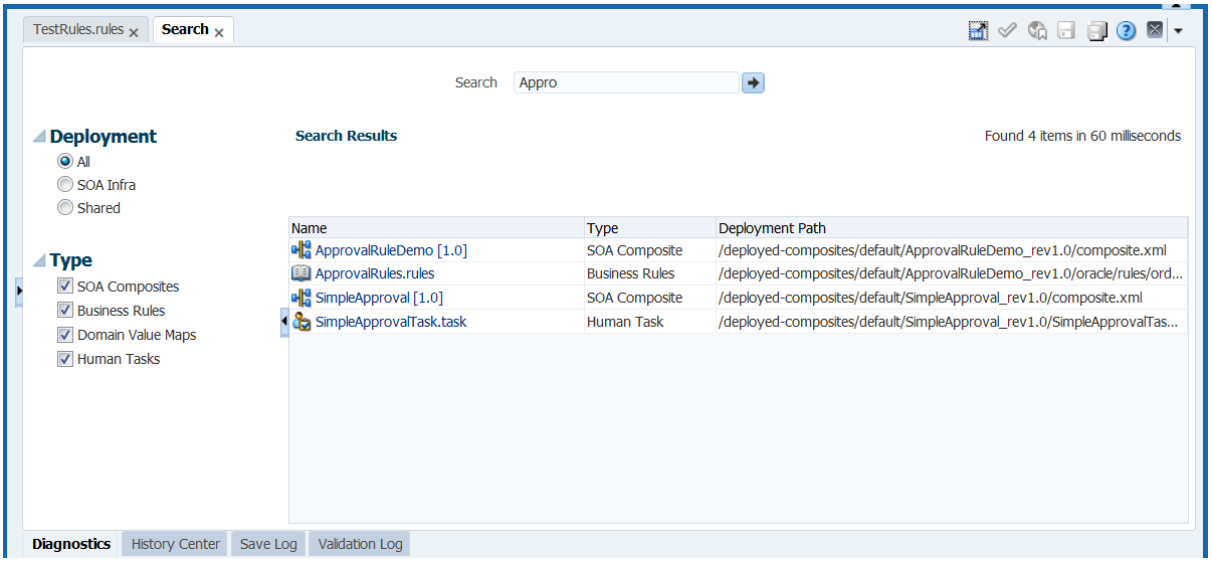
The **Deployment View** has two nodes: **SOA-Infra** and **Shared.** The **default** node is a SOA partition created and managed in Enterprise Manager. The **Shared** folder displays rules or DVMs created in JDeveloper and deployed as shared artifacts.

From either view, click artifacts in the navigation tree to open them in separate tabs.

In SOA Composer, the Verbal Rules and Business Phrases features do not appear if you have not installed BPM.

Use the search field just above the **View** drop down to quickly find and filter types by name, as shown in Figure 12-2. Click to open artifacts from this page.

Figure 12-2 Search and Filter



Creating and Publishing Sessions

Click an artifact to open it in read-only mode. If you plan to make changes to an artifact, click the **Edit Session** button. Most action buttons only become active if you

are in a session. When you are done making changes, click **Publish.** All changes in a session are committed to the main repository, can be seen by others, and the server will begin executing. If you have validation errors, you cannot publish, though you can save rules with validation errors and work on them again in another session.

The **Discard** button enables you to cancel out of changes that you do not want to save. If, after making some changes in a session, you wanted to go back to the original state that you started from, click **Exit Session.** Click **Edit Session** again to see the last saved session information.

These icons and buttons provide more information:

Table 12-1 SOA Composer Buttons

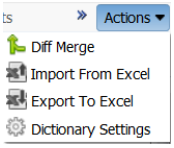
Button	Description
 Session Info	Hover over this icon to see Session Details.
 Maximize	Click to maximize the tabs section. This increases screen space when writing or editing rules. Click again to restore the view.
 Validate	Click to validate your changes. The system validates when you save and you can save rules with validation errors, though you cannot publish. For more information about validating, see Validating and Diagnosing an Oracle Business Rules Dictionary.
 Bookmark	Click the bookmark if you want to avoid the search/ deployment view in a future session. For more information, see Creating a Bookmark.
Save Changes in Current Tab or Save Changes in All Tabs	Click to save as appropriate.
Get Context Sensitive Help	Click to view the online help file.
Close All, Close Others	Click the drop down and select to close tabs.
	The **Actions** drop down enables you to use the compare and merge dictionaries, work with Decision Tables in Microsoft Excel, and review dictionary settings. Diff Merge enables you to compare the currently selected dictionary with the File System, the Published Version, or the Saved Version. If there are differences, you can choose to merge the dictionaries. For more information about comparing or merging dictionaries, see Comparing and Merging Oracle Business Rules Dictionaries. Import From Excel enables you to import decision tables from Excel. Export To Excel enables you to export decision tables and edit them in Excel. For more information about working with Excel, see Editing Decision Tables in Microsoft Excel. Dictionary Settings enables you to set dictionary preferences. For more information about reviewing dictionary settings, see Reviewing Dictionary Settings.

Table 12-1 (Cont.) SOA Composer Buttons

Button	Description
Diagnostics, History Center, Save Log, Validation Log	At the bottom of the SOA Composer page are four tabs: **Diagnostics, History Center, Save Log**, and **Validation Log.** Use these tabs to validate changes to rules and perform and resolve changes to artifacts. For more information about these tabs, see Validating and Diagnosing an Oracle Business Rules Dictionary.

Publishing Changes for an Oracle Business Rules Dictionary

After you verify dictionary modifications, click Publish to commit those changes to the MDS repository.

To publish changes to an Oracle Business Rules dictionary:

1. Click the **Publish** menu item.
2. In the Confirm dialog, click **Yes** if you want to make the changes in the MDS repository. Click **No** if you do not want to make the changes in the MDS repository.

   Remember to update the runtime changes into Rule Editor ADF following the tasks described in Importing Runtime Rules Changes From Repository Into JDeveloper
3. When you open the dictionary after saving the edit session and deploying the composites, SOA composer opens the last saved edit session.

   When multiple users are editing the same dictionary, Oracle SOA Composer shows a message that the dictionary is being edited by another user and asks for a confirmation. When multiple users work on a single dictionary, only the last publish is persisted.

   To open the new dictionary click **Discard**, **Clear all session edits and save changes** button in the top menu.

Note:

A dictionary with validation errors can be saved, but it can be committed only after correcting the validation issues.

Creating a Bookmark

Create a bookmark to avoid the search or deployment view.

To open a dictionary using a known URL:

1. In an open dictionary, click **Bookmark** in the toolbar of Oracle SOA Composer.
2. Copy the URL and paste in a browser to launch SOA composer with the bookmarked artifact opened in a tab.

Figure 12-3 Obtain the URL for an Open Dictionary

3. In a browser, use the saved URL to directly access the dictionary.

   For example,

   ```
   http://SERVER_NAME_OR_IP_ADDRESS/soa/composer?docPath=/deployed-
   composites/ default/BusinessRulesTest_rev1.0/oracle/rules/
   businessrulestest/OrderBooking.rules
   ```

   According to the preceding example, composites are stored as per the following structure: deployed-composites/composite name_rev *`composite revision`*/ oracle/rules/*`dictionary package path`*/*`dictionary name`*.rules

Reviewing Dictionary Settings

Click the Dictionary Setting button to set preferences.

Figure 12-4 Dictionary Settings Dialog

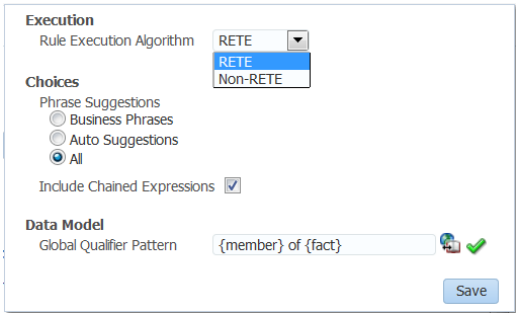


Table 12-2 Dictionary Settings

Dialog Sections	Settings
Execution section	Rule Execution Algorithm: choose RETE or Non-RETE. For more information, see 1.3 Oracle Business Rules Engine Architecture. The Rete Algorithm and The Non-Rete Algorithm
Choices section	Phrase Suggestions: choose Business Phrases, Auto Suggestions, or All.
Data Model section	Global Qualifier Pattern: confirm {member} of {fact}. Translations: click to translate {member} of {fact}. Validate: click to validate {member} of {fact}.

Using Oracle SOA Composer User Authentication

Figure 12-5 shows the Oracle SOA Composer login page. This page allows Oracle SOA Composer to authenticate the specified user.

Figure 12-5 Oracle Oracle SOA Composer Login Page

To login to Oracle SOA Composer:

1. Access Oracle SOA Composer using the following URL in your browser address bar:

   `http://SERVER_NAME_OR_IP_ADDRESS/soa/composer`

2. In the Oracle SOA Composer login page, in the **Username** field, enter a user name.
3. In the **Password** field, enter a password.
4. Click **Sign In**.

For information about creating and managing users and groups, see the integrated SOA Composer Console online help.

What You Need to Know About SOA Composer Access Control and User Authentication

Oracle SOA Composer supports user and password access control and only authenticated users can use Oracle SOA Composer. However, Oracle SOA Composer does not provide finer grained access control. For example, Oracle SOA Composer does not support access control for individual rulesets or rules within a dictionary.

Oracle SOA Composer does support access control to metadata. Using Oracle SOA Composer, only users with the SOADesigner application role can access the metadata from Oracle SOA Composer. By default, all users with WLS Administrator privileges have this role.

If a user without the SOADesigner role logs into Oracle SOA Composer, a message appears, stating that the user is not authorized to modify the SOA metadata.

For more information about assigning the SOADesigner role to a nonadmin user who requires access to Oracle SOA Composer, see Managing Application Roles in Oracle Enterprise Manager Fusion Middleware Control Console in *Oracle Fusion Middleware Administrator's Guide for Oracle SOA Suite and Oracle Business Process Management Suite*.

Setting Accessibility Options

Accessibility settings help you read all components of the application.

You can set accessibility options in SOA Composer for the current instance, or for all instances.

How to Set Accessibility Features Before Logging In

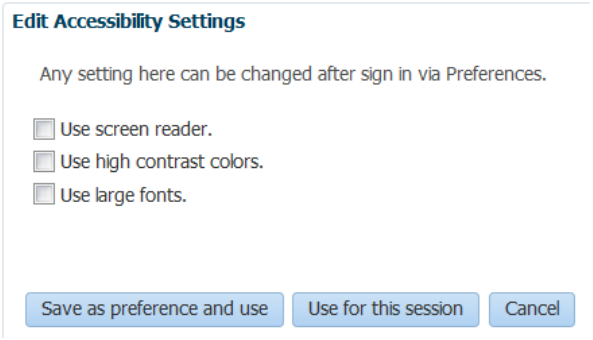
SOA Composer presents the Accessibility menu on the login page, so you can configure accessibility before you log in. These settings can be persisted for only the current session or for all sessions.

To set accessibility options for the current session only:

1. Launch SOA Composer.
2. On the login page, click **Accessibility** in the top right corner.

   The Edit Accessibility Settings page appears, as shown in Figure 12-6.

Figure 12-6 Edit Accessibility Settings Page



3. Select any of the following options:
   - Use screen reader.
   - Use high contrast colors.
   - Use large fonts.
4. To save the new settings only for this session, click **Use for this session**. To save the settings for all sessions, click **Save as preference and use**.

How to Set Accessibility Options After Logging In

Once you log in to SOA Composer, you can configure accessibility options from any page. This changes the user preferences, which are retained through all sessions until you change them again.

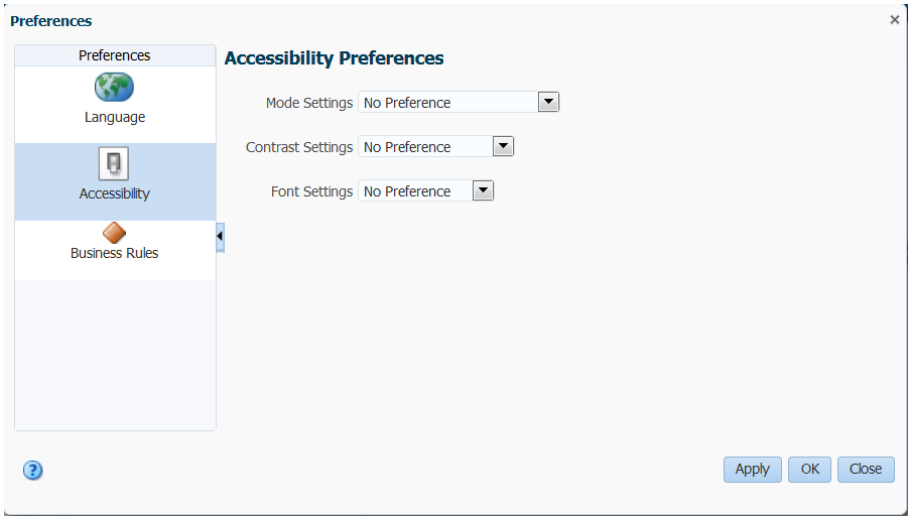
To set accessibility options after logging in:

1. Launch SOA Composer and log in.
2. From any page, select **Preferences** in the top right corner.

   The Preferences dialog appears.
3. In the Preferences column, click **Accessibility**.

   The Accessibility Preferences appear, as shown in Figure 12-7.

Figure 12-7 Preferences Dialog



4. In the **Mode Settings** field, select **Enable screen reader mode** if you use a screen reader. Select **Disable screen reader mode** if you do not use a screen reader.
5. In the **Contrast Settings** field, select **Use high contrast** to increase the contrast between objects on the console; otherwise, select **Use normal contrast**.
6. In the **Font Settings** field, select **Use large fonts** to increase the font size; otherwise, select **Use normal fonts**.
7. Click **OK**.

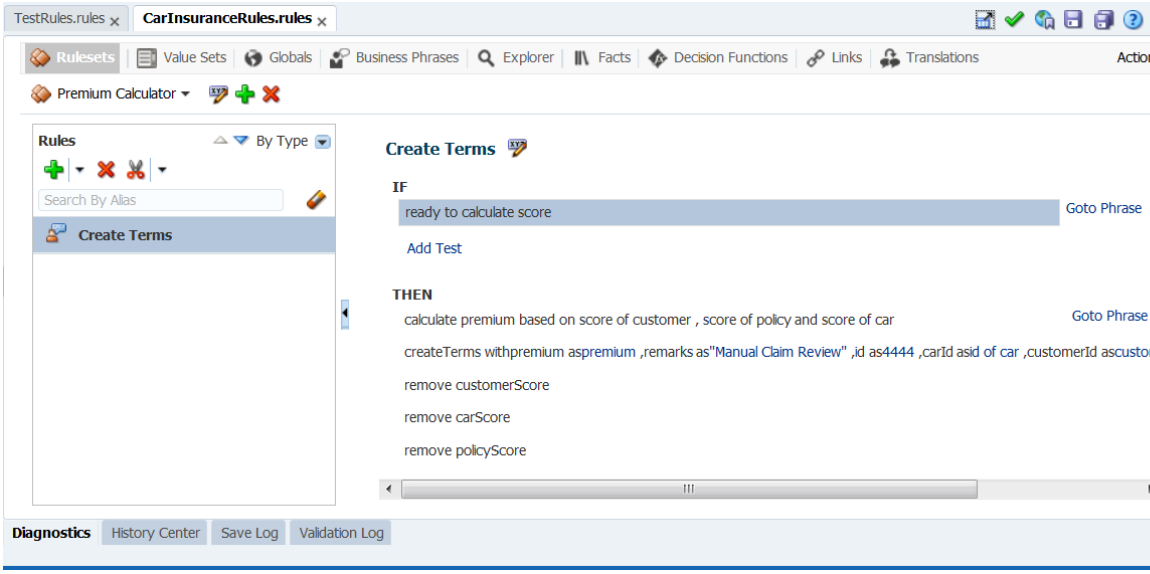
Opening and Viewing an Oracle Business Rules Dictionary

When you open Oracle SOA Composer, it connects to MDS and displays the available composite applications that contain dictionaries. In addition, it lists the shared dictionaries, and these shared dictionaries can also be viewed and edited.

As shown in Figure 12-8, Oracle SOA Composer shows a navigation tree that displays a left-side panel with a list of metadata artifacts. Details for the selected item are shown on the right-hand side. Oracle SOA Composer includes the following tabs:

- Rulesets
- Value Sets
- Globals
- Business Phrases
- Tests
- Explorer
- Facts
- Decision Functions
- Links
- Translations

Figure 12-8 Rules Tabs



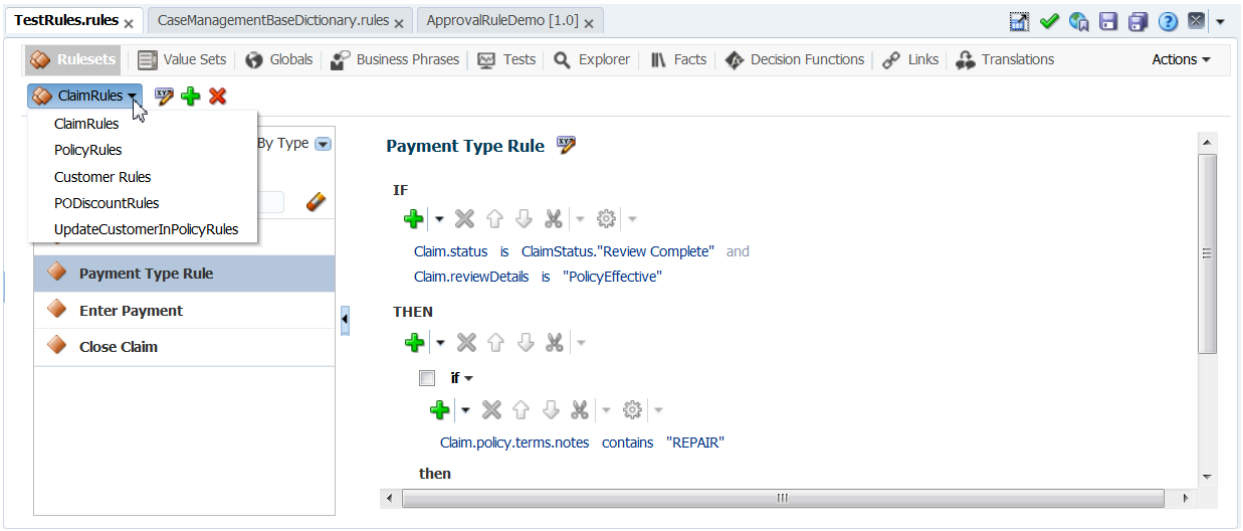
Note:

In SOA Composer, the Verbal Rules and Business Phrases features do not appear if you have not installed BPM.

How to View and Edit Rulesets

Oracle SOA Composer displays the rulesets in the dictionary, as shown in Figure 12-9. You can select a ruleset to display a detailed view of the ruleset. You can add and delete rulesets and rules.

Figure 12-9 Using the Oracle SOA Composer Rules Dictionary Rulesets Tab to View Rules



To use the ruleset tab:

For detailed documentation of fields and other UI controls, click **Help, Help for This Page** from within SOA Composer.

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Rulesets** tab, and click the **Create Session** button. The action buttons are enabled.
3. Click the down arrow next to ClaimRules, as shown in Figure 12-9, and choose a Ruleset. The ruleset is displayed and is editable. You can also add or delete rulesets from the toolbar.
4. Click the Rules panel to add Decision Tables, Verbal Rules, or General Rules. You can also delete, cut, copy, or paste rules.

   For information on adding verbal rules in SOA Composer, see How to Add Verbal Rules in SOA Composer.
5. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

For more information about Decision Tables, see Editing Decision Tables in an Oracle Business Rules Dictionary.

For more information about Verbal and General Rules, see Editing Rules in an Oracle Business Rules Dictionary

For more information about Rulesets, see Working with Rulesets and Rules .

How to Add Verbal Rules in SOA Composer

Verbal rules provide a flexible way to author rules using natural language statements to express rule logic in domain specific sentences.

To add verbal rules in SOA Composer:

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Rulesets** tab, and click the **Create Session** button. The action buttons are enabled.
3. Select the Ruleset from the drop down list.
4. Click the Rules panel to add Decision Tables, Verbal Rules, or General Rules.

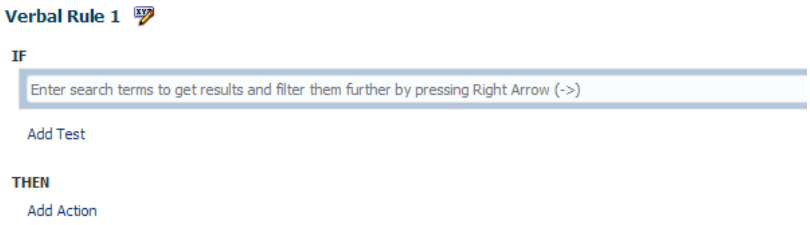
   To add Decision Tables, see Adding a Decision Table.

   To add Verbal Rules:

   a. Click the **Add** icon and select **Verbal Rules**.

      The Verbal Rules window appears.

Figure 12-10 Verbal Rule window



b. In the **If** field, add a test. Once done, add an action in the **Then** field.

Note that when you add a test or an action, the test or the action becomes editable. Type in a filter in the If field, for example, 'customer number', all related options are displayed in the drop down list.

c. Use the up/down arrow keys to select and use the right arrow key on the selected option to get similar choices.

To get more choices in the list, scroll to the end of the drop down list and select the **More** option by using either keyboard or mouse.

d. From the list, select existing business phrase or you can instantiate a new business phrase based on what you typed.

Once a choice is set, the text field is no longer editable and the existing parameters become links. The links when clicked becomes editable. To set the value, double-click the links.

Note:

Some of the important keyboard-based interface for Verbal Rules are:

- Copy: ctrl + c
- Paste: ctrl + v
- Cut: ctrl + x
- Move row up: ctrl + up
- Move selection up: up arrow
- Move row down: ctrl + down
- Move selection down: down arrow
- Delete row: ctrl + delete
- Add new row: ctrl + enter
- Edit row: enter

Important keyboard gestures for setting parameter values:

- Avoid having a selected row while specifying parameter values. Since 'enter' makes the selected row editable and you may be trying to set a parameter value.
- Using the Esc button within the parameter text field converts it back to a link without setting the value.
- Using tab or entering key values does not make the link editable when its in focus. You must use the mouse or use the enter key to activate the link.
- To set a parameter value, you can tab out of the parameter text field or press enter.

5. When done, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

To add General Rules, see How to Add General Rules.

How to View and Edit Value Sets

When you open a dictionary and select the **Value Sets** tab, if the dictionary contains value sets, the table shows all available value sets. Value sets from linked dictionaries are also displayed. You can select a linked value set and click the **Edit** button to view the values. However, a linked value set is not editable even in the edit mode.

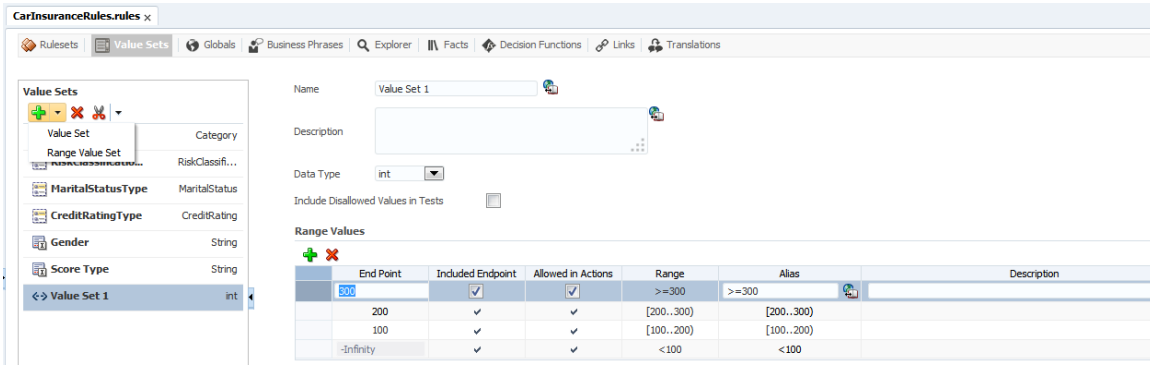
For information on the Oracle SOA Composer edit mode, see Getting Started with Editing a Dictionary.

To view value sets in Oracle SOA Composer:

For detailed documentation of fields and other UI controls, click **Help, Help for This Page** from within SOA Composer.

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Value Sets** tab, and click the **Create Session** button. The action buttons are enabled, as shown in Figure 12-11.
3. Click + and select **Value Set** to add a value set to the dictionary.
4. Click + and select **Range Value Set** to create a range value set.

Figure 12-11 Using the Oracle SOA Composer Rules Dictionary Value Sets Tab



5. Click **+** to add values in the table. You can click in a row to make it editable. Selected rows can also be deleted or moved up or down.
6. The **Name, Description,** and **Data Type** cannot be changed only for 'enum' type value sets. They are editable otherwise.
7. The **Include Disallowed Values in Tests** controls whether cleared values from the Values list are included in tests.
8. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

How to View and Edit Globals

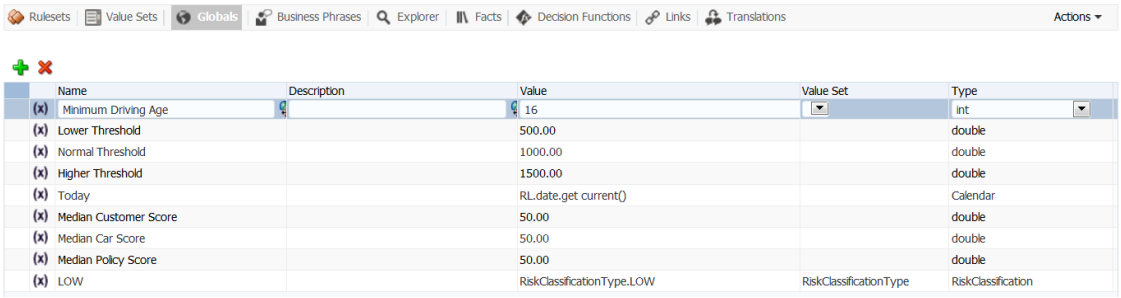
When you open a dictionary Oracle SOA Composer displays the **Globals** tab. Globals can be final or not and can be edited in SOA Composer.

For the **Value** field, you can use the expression builder to set the value. To check for validity, you can click the **Validate** button.

To view globals in Oracle SOA Composer:

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Globals** tab, and click the **Create Session** button, as shown in Figure 12-12.

Figure 12-12 Using the Oracle SOA Composer Rules Dictionary Globals Tab



3. Click **+** to add a Global. Enter a **Name, Description**, and **Value**.
4. Choose a **Value Set** and **Type** from the drop down.
5. Check the **Final** check box to indicate whether the global can be changed at runtime.
6. Check the **Constant** check box to indicate if the global is a constant or can be modified.
7. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

How to View and Edit Business Phrases

Use the Business Phrases tab to view and manage business phrases in your rules project.

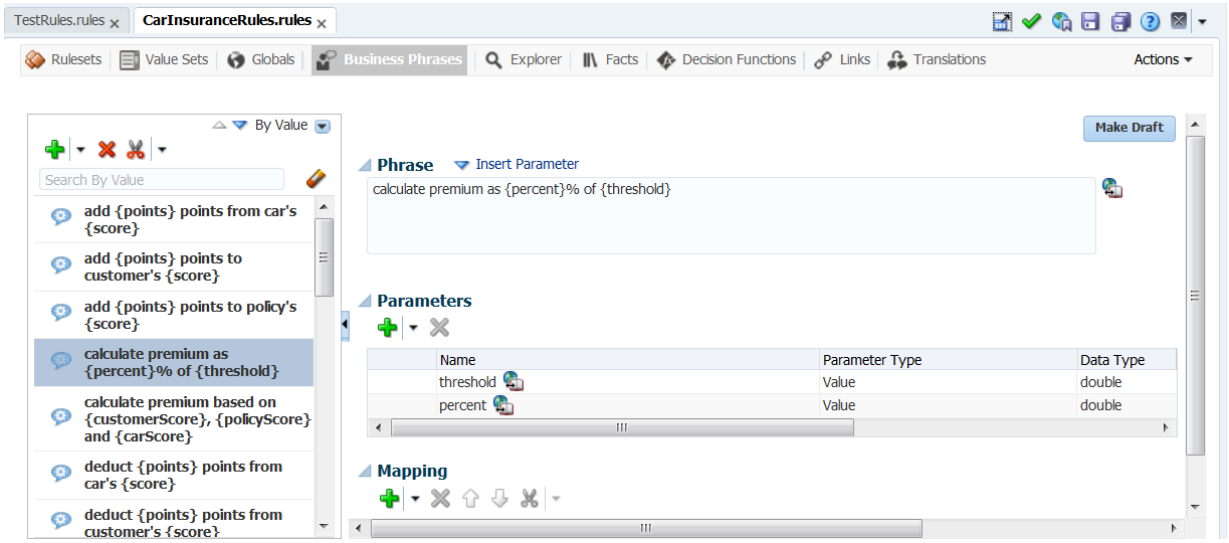
Note:

In SOA Composer, Verbal Rules and Business Phrases features do not appear if you have not installed BPM.

To edit Business Phrase:

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Business Phrases** tab, and click the **Create Session** button. The action buttons are enabled, as shown in Figure 12-13.

Figure 12-13 Using the Oracle SOA Composer Rules Dictionary Business Phrases Tab



3. Click the action buttons to add, cut, copy or paste.

   To add a Business Phrase, enter the following details:

   a. Phrase - A phrase can be a test or an action. It can be an English phrase, for example, "calculate premium as {threshold} of {percentage}". You can mark the phrase as a draft to edit later by selecting the **Make Draft** button.

   b. Parameters - You can edit, add, or delete the parameters in the parameters table. You can drag and drop parameters into the phrase field. You can also use the **Insert parameter** link to drop parameters into the phrase.

   c. Mapping - The mapping section is used to map the business phrase to the internal test/action.

4. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

For more information about business phrases, see Introduction to Verbal Rules and Business Phrases.

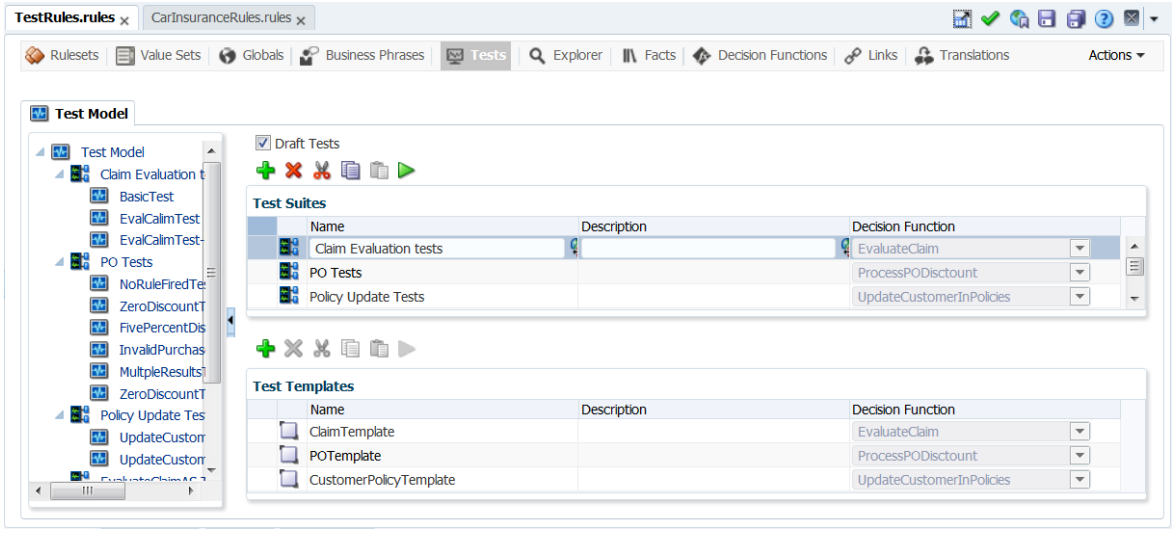
How to View and Edit Tests

At runtime, you can use SOA Composer to regression test rules. This enables business users to quickly check if a modified rule changes the existing functionality. The Tests tab only appears if you have a deployed composite and are in a SOA Composer session.

To view and edit tests:

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Tests** tab, and click the **Create Session** button. The action buttons are enabled, as shown in Figure 12-14.

Figure 12-14 Using the Oracle SOA Composer Rules Dictionary Tests Tab



3. You can create and run Test Suites and Test Templates.

   For more information about testing and validating rules at runtime, see Testing Rules in SOA Composer.

4. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

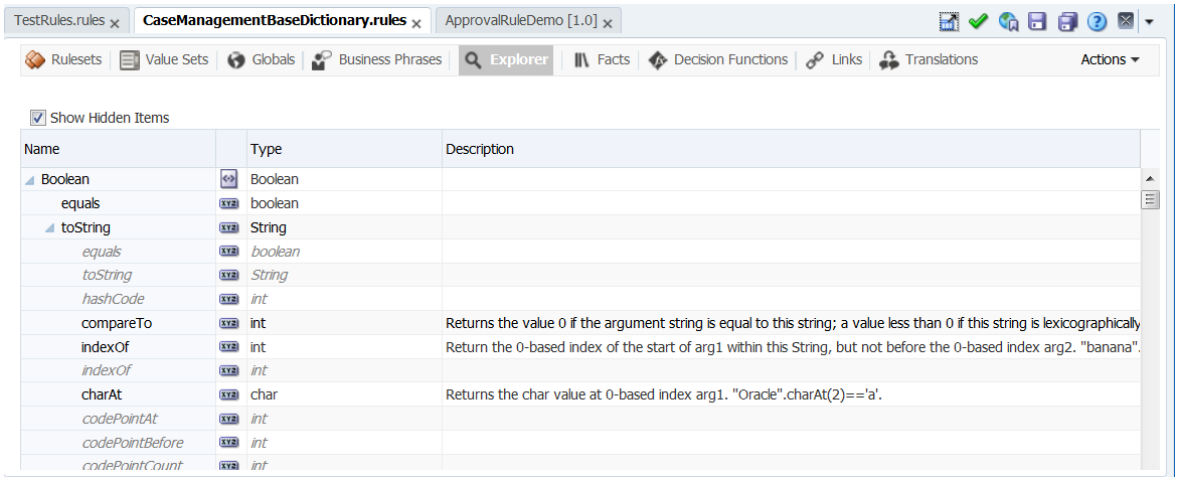
How to View Explorer

To view the Explorer tab:

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Explorer** tab, as shown in Figure 12-15.

   The Explorer tab is used to view the data, its type and description. You cannot make any changes in the Explorer table.

Figure 12-15 Using the Oracle SOA Composer Rules Dictionary Explorer Tab

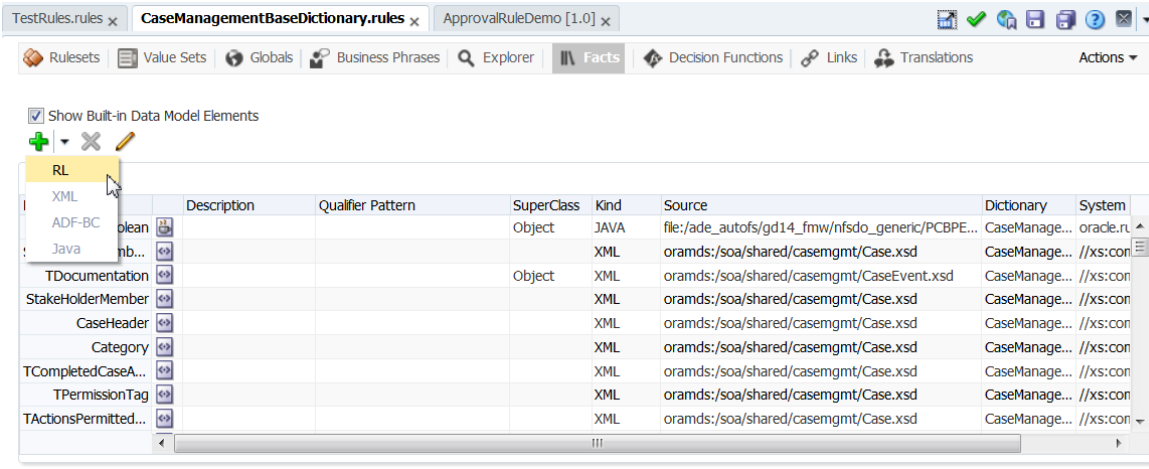


How to View and Edit Facts

To view and edit facts:

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Facts** tab, and click the **Create Session** button. The action buttons are enabled. Only RL facts can be created in SOA Composer, as shown in Figure 12-16.

Figure 12-16 Using the Oracle SOA Composer Rules Dictionary Facts Tab



3. Select a fact and click the **Edit Facts** button to open the **Edit Facts** dialog. You can edit RL and XML facts here, but Java and ADFBC facts are read-only.
4. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

How to View Decision Functions

In Oracle SOA Composer, you can view the decision functions that are available to the current dictionary by using the Decision Functions tab. Currently, even in a session, you can only modify the following fields and options:

- Description
- Rule Firing Limit
- Check rule flow
- Make stateless
- Available Rulesets to fire

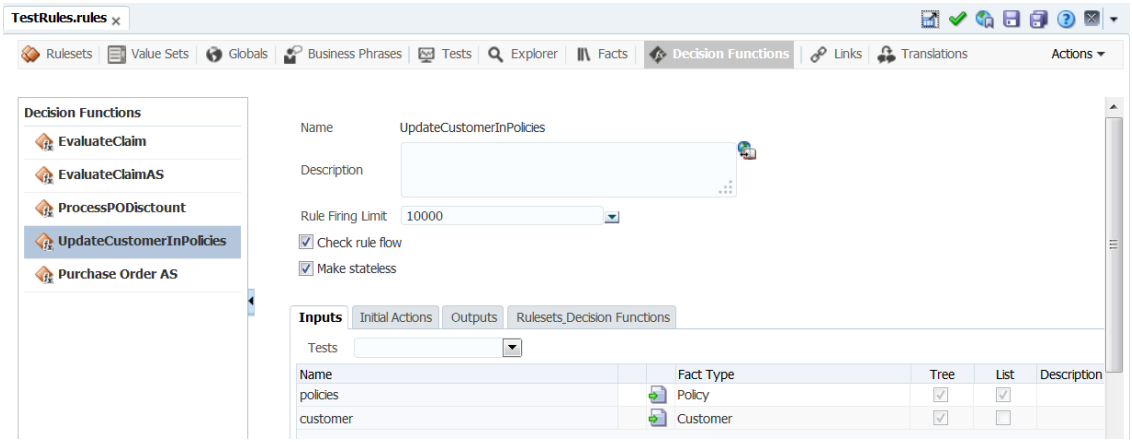
You cannot create any decision function, rename an existing decision function, or add or delete any input or output.

To view decision function names in Oracle SOA Composer:

1. In Oracle SOA Composer, open a Rules file.
2. Click the Decision Functions tab.

3. You can view information on the following tabs: **Inputs, Initial Actions, Outputs,** and **Rulesets Decision Functions** as shown in Figure 12-17.

Figure 12-17 Viewing Decision Functions



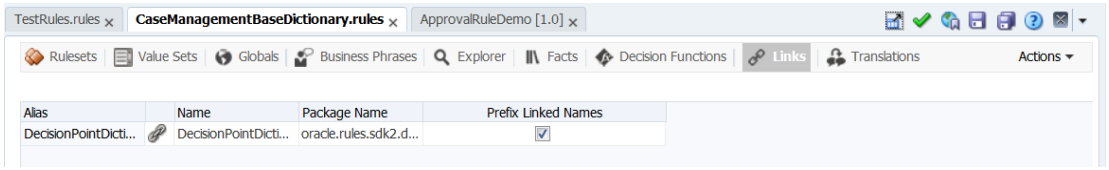
How to View Linked Dictionary Names

In Oracle SOA Composer, you can view the names of the dictionaries to which the current dictionary is linked by using the **Links** tab as shown in Figure 12-18. Currently, even in a session, you can view the linked dictionary names, but you cannot link to a dictionary or delete an existing link to any dictionary.

To view linked dictionary names in Oracle SOA Composer:

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Links** tab, as shown in Figure 12-18.

Figure 12-18 Viewing the Linked Dictionary Name



3. Select to **Prefix Linked Names.**
4. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

How to Work With Dictionary Links in an Oracle Business Rules Dictionary

An Oracle Business Rules dictionary can be linked to other dictionaries. The complete data model defined by a dictionary and its linked dictionaries is called a combined dictionary. You can create multiple links to the same dictionary. However, in this case, all but the first link is ignored.

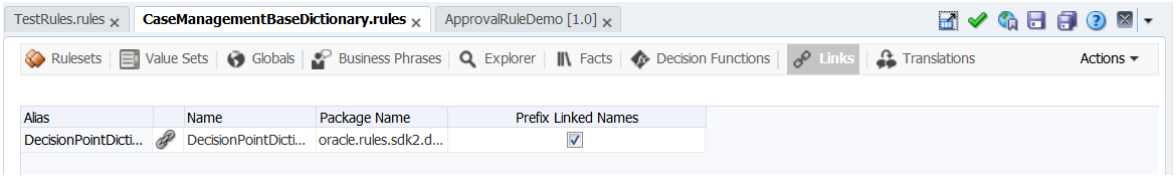
You cannot use Oracle SOA Composer to link dictionaries. However, if a deployed composite already has linked dictionaries, using Oracle SOA Composer, you can view the linked dictionary names and make use of the Globals, Value Sets, and Rulesets of the linked dictionaries across applications. For example, you have an application called `App1` that contains a dictionary called `Dict1`. `Dict1` is linked to another

dictionary called `Dict2`. Because `Dict1` is linked to `Dict2`, the objects of `Dict2` will be available for use in `App1`.

For more information on viewing linked dictionary names, see How to View Linked Dictionary Names.

In Oracle SOA Composer, you can use the **Prefix Linked Names** check box in the Links table to either display or hide the linked dictionary name that is prefixed to the all the items in the dictionary such as Globals, Value Sets, and Rulesets. Selecting the check box prefixes facts from the linked dictionary with its dictionary name, and deselecting hides the linked dictionary facts prefix. By default, the **Prefix Linked Names** check box is in selected state as shown in Figure 12-19.

Figure 12-19 Using the Links Tab



For more information about linked dictionaries, see What You Need to Know About Dictionary Linking.

How to View and Edit Translations

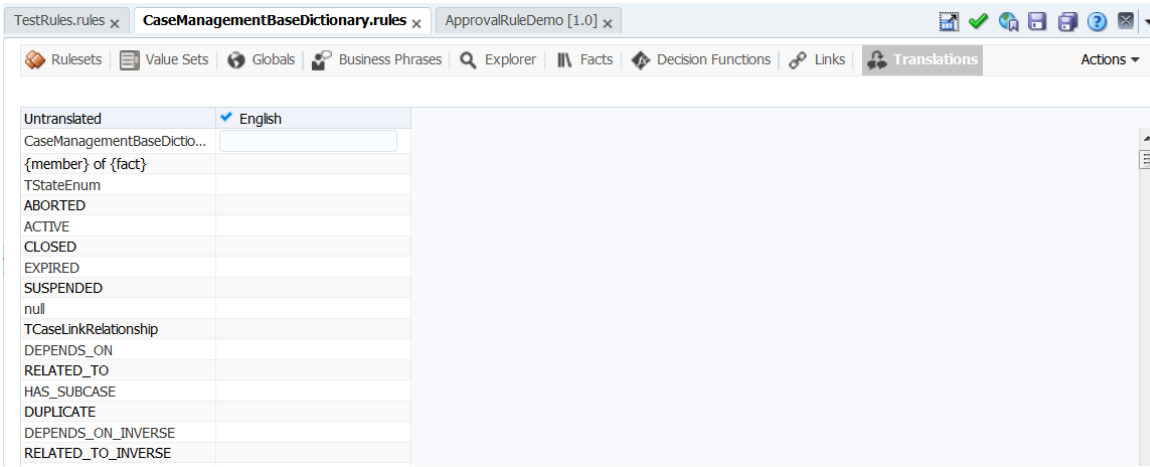
Use the Translations tab to view the phrases included in the selected dictionary and their translated strings.

The translation table contains all translated strings in the current locale as well as earlier locales. For example, the current locale is Japanese and you edit some translations and save them. If you log out, and then log back in with a different locale, for example, English, then the translation table will contain columns for both English and Japanese.

To translate phrases:

1. In Oracle SOA Composer, open a Rules file.
2. Click the **Translations** tab, as shown in Figure 12-20.

Figure 12-20 Using the Oracle SOA Composer Translations Tab



3. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

Getting Started with Editing a Dictionary

When you select and open a dictionary, Oracle SOA Composer shows the dictionary in read-only mode. On each tab in read-only mode, use the Session buttons to make changes and then Save them to a work area. To apply the changes to the runtime version of the dictionary, click Publish.

For more information about how to use SOA Composer features, see Creating and Publishing Sessions.

What You May Need to Know About Localized Number Formatting Support in Oracle SOA Composer

In Oracle SOA Composer, number formatting changes based on the browser locale. For example, you are using Oracle SOA Composer with U.S. English as the browser language. You enter a floating-point data, such as 34533223.2345, as a value. If you wish to view the data in any other language, such as French, you need to:

1. Modify the browser locale for the instance to French.
2. Click the **Refresh** button of the browser to view the number formatting changes

   In French, the value should display as 34533223,2345.

Note:

The grouping and decimal separators specified in Oracle SOA Composer overrides the locale-specific ones.

What You May Need to Know About Cutting/Copying and Pasting Rule Elements

You can cut/copy a value set or rule from one dictionary and open another dictionary in composer and paste it. However, cut/copy/paste works between different dictionaries within the same session.

Cutting/copying and pasting feature enables you to quickly create a new rule element based on an existing one, without having to create the new element from scratch.

The buttons in the Figure 12-21 help you with cut, copy and paste options.

Figure 12-21 Cut, Copy and Paste Buttons

Oracle SOA Composer enables you to cut/copy and paste the following elements of a rule:

- Rules
- Patterns
- Conditions
- Actions

- Value sets

Cut/copy/paste is **not** supported for the following:

- Globals
- Links
- Values
- Decision Functions

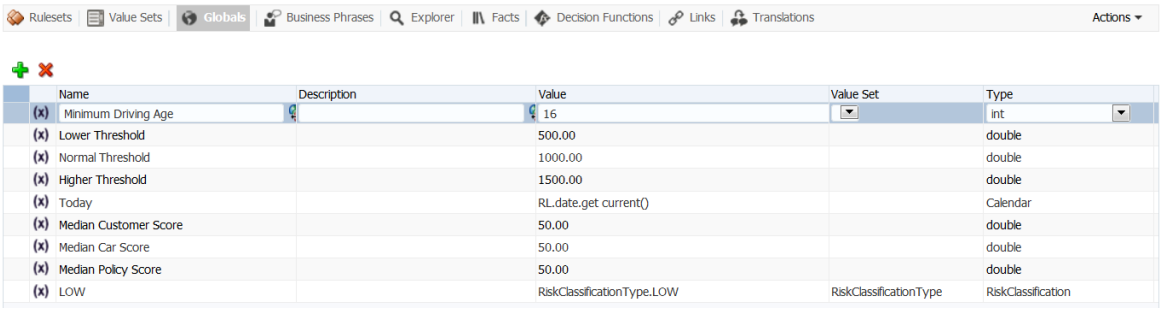
Note:

The **Paste** button is disabled if multiple conditions or actions are selected. The button is enabled only on single selected condition/action. When pasting, the copied/cut items are added at the end of the list.

How to Edit Globals in an Oracle Business Rules Dictionary

In Oracle SOA Composer, selecting the **Globals** tab shows you a table listing the globals in the dictionary, as shown in Figure 12-22. To edit a global, select the appropriate row, and the entire row becomes editable. Make necessary changes as required.

Figure 12-22 List of Globals in the DIctionary



To add a global, click the **Add Global** button on the top. A new empty row is added. Make necessary changes to Name, Description, Value, Value Set, Type, Final, Consent. For more information on adding globals, see Working with Oracle Business Rules Globals.

To delete a global, select a row and click the **Delete** button.

How to Edit Value Sets in an Oracle Business Rules Dictionary

In Oracle SOA Composer, selecting the **Value Sets** tab displays a master list on the left which displays the value sets in the dictionary, and a detail section with a table that display the values. To edit a value set, click the appropriate Value Set in the master list and then click the value in the detail section that you want to change.

You can create a Range Value Set by clicking the **Add** button and selecting a type. This adds a new value set in the master list. Adding a range value automatically adds an end point for a range and a value for an LOV based on the datatype. You can modify the newly added value end point or value. Note that the alias is modified when an end point or value is changed.

For more information on adding value sets, see Working with Value Sets and Associating a Value Set with Business Terms.

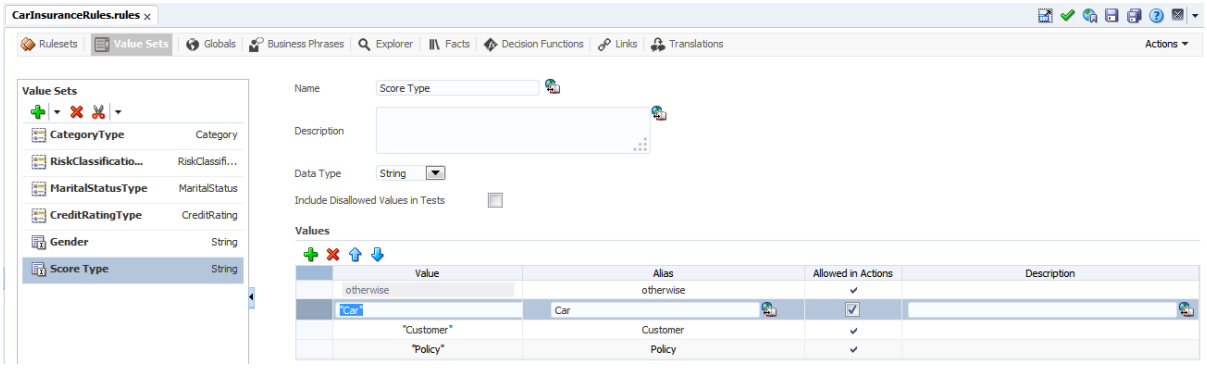
To cut or copy a value set, select a row and click **Cut** or **Copy**. To paste a copied value set, click **Paste**.

To delete a value set, select a row and click **Delete**.

To edit Value Sets:

1. To edit either a Value Set or a Range Value Set, in Oracle SOA Composer select the **Value Sets** tab. This displays both master and detail sections for the value sets in the dictionary.
2. Select the appropriate Value Set from the master list. This displays the detail table, as shown in Figure 12-23.

Figure 12-23 Editing Value Sets



3. Edit the appropriate fields in the table. You can click **Add Value** to add a value, and also select a row and click **Delete Value** to delete a value.
4. To change the order of values in the value set, select a value and then click the up or down arrow to move the selected value.

   You can change the relative position of values in an LOV value set only; you cannot reorder values in a Range value set.

   Only when a value has the **Allowed in Actions** field selected does the value set display in the condition cell drop down in a Decision Table.
5. Click **Save Changes in Current Tab** to confirm the changes.

   Click **Validate** in the menu bar to validate the dictionary while making changes to a Value Set.

How to Edit Decision Functions in an Oracle Business Rules Dictionary

In Oracle SOA Composer, the **Decision Functions** tab displays a table listing the decision functions that are available to the dictionary, both parent and linked.

You can only modify the following fields and options:

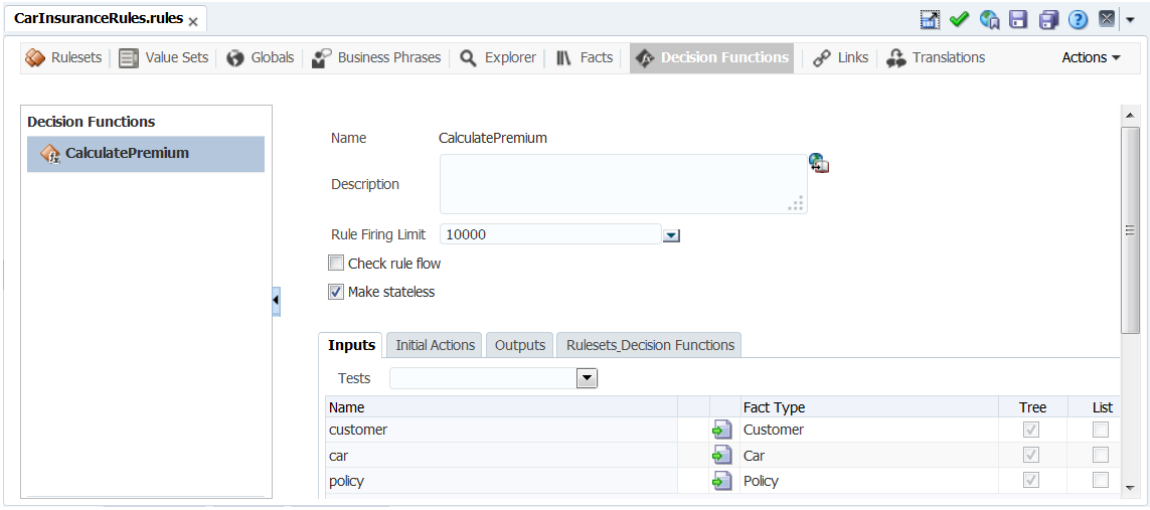
- Description
- Rule Firing Limit

- Check rule flow
- Make stateless
- Initial Actions
- Rulesets and Decision Functions

To edit a decision function:

1. To edit a decision function, in Oracle SOA Composer, select the **Decision Functions** tab. This displays a master list of decision functions on the left, and the detail panel on the right.
2. Select the appropriate decision function on the left. This displays the Decision Function Editor dialog box as shown in Figure 12-24.

Figure 12-24 Editing a Decision Function



3. In the **Description** field, optionally enter a description.
4. Enter the required number value from the **Rule Firing Limit** list. By default, the selected value is `unlimited`. However, you can enter an integer value for the rule firing limit and press the Tab key. The newly specified value gets added to the **Rule Firing Limit** list.
5. Select the appropriate decision function options:
   - **Check rule flow:** When selected, this option specifies that the rule flow is checked
   - **Make stateless:** When selected specifies the decision function is stateless.

   You cannot edit the following:
   - Name field
   - Inputs tab
   - Outputs tab

6. In the **Initial Actions** tab, you can add actions that could be used to change input facts before they are asserted, change the ruleset stack, set the effective date, or even assert output facts. These actions could be used instead of rules, or to "set up" the environment for running rules. Initial Actions always run just before the inputs are asserted and the rules are run. The RL for the actions will be executed just before the inputs are asserted.

   Consider a situation where a decision function (DF1) calls another decision function (DF2) using the **Initial Actions** tab. DF1 is configured to push Ruleset1 to the ruleset stack. DF2 is configured to push Ruleset2. In DF1, before the initial actions are executed, Ruleset1 is pushed to the ruleset stack. Then, when DF2 is called, Ruleset2 is also pushed. So when rules start running, rules from both rulesets fire because of the ruleset stack. If you want to push Ruleset2 (because in the initial actions, you are calling DF2), you can use initial actions in DF1 to clear the ruleset stack before calling DF2, and push Ruleset1 on the stack after calling DF2.

   You can add any required action ranging from `assert`, `call`, `modify` to even conditional actions such as `if`, `else`, `elseif`, `while`, `for`, `if (advanced)`, and `while (advanced)`.

   The `if (advanced)` and `while (advanced)` structs accepts only `boolean` values. For each of the action conditions, you can add different test form types.

   Note:

   If decision function DF1 contains DF2 in the **Rulesets & Decision Functions** tab, then DF2 may not have any initial actions.

   ---

7. In the **Rulesets & Decision Functions** tab, use the left and right arrow buttons to move items from the **Available** box to the **Selected** box.

8. Select an item in the **Selected** box, and click up or down arrow buttons as appropriate to order the rulesets and the decision functions.

9. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

For more information on decision functions, see Working with Decision Functions.

What You May Need to Know About Oracle Business Rules Dictionary Editor Declarative Component

You can use the Oracle Business Rules Dictionary Editor declarative component to leverage the functionality of editing Rules Dictionaries in any ADF-based Web application. It enables you to edit business rules metadata artifacts, such as Globals, Value Sets, and Rulesets, by using the Rules SDK2 API.

For more information on Oracle Business Rules Dictionary Editor, see "Using the Oracle Business Rules Dictionary Editor Declarative Component" in *Developing SOA Applications with Oracle SOA Suite*.

What You May Need to Know About Oracle Business Rules Dictionary Editor Task Flow

Rules Dictionary Editor Task Flow, which is a wrapper around the Rules Dictionary Editor declarative component is used in ADF-based Web applications that require a task flow instead of a declarative component.

For more information on Oracle Business Rules Dictionary Editor, see Using the Oracle Business Rules Dictionary Task Flow in *Developing SOA Applications with Oracle SOA Suite*.

Editing Rules in an Oracle Business Rules Dictionary

SOA Composer provides an interface to the dictionary that enables you to edit most dictionary components, though you can only create and edit some dictionary components at design-time using the Rules Designer extension to Oracle JDeveloper.

In SOA Composer, Verbal Rules and Business Phrases features do not appear if you have not installed BPM.

Using the Rulesets Tab

Use the Rulesets tab to view and edit Rulesets, and the General Rules, Verbal Rules and Decision Tables they contain, in the currently selected Business Rules dictionary, as shown in Figure 12-25.

Figure 12-25 Using Oracle SOA Composer to Edit a Ruleset in a Dictionary

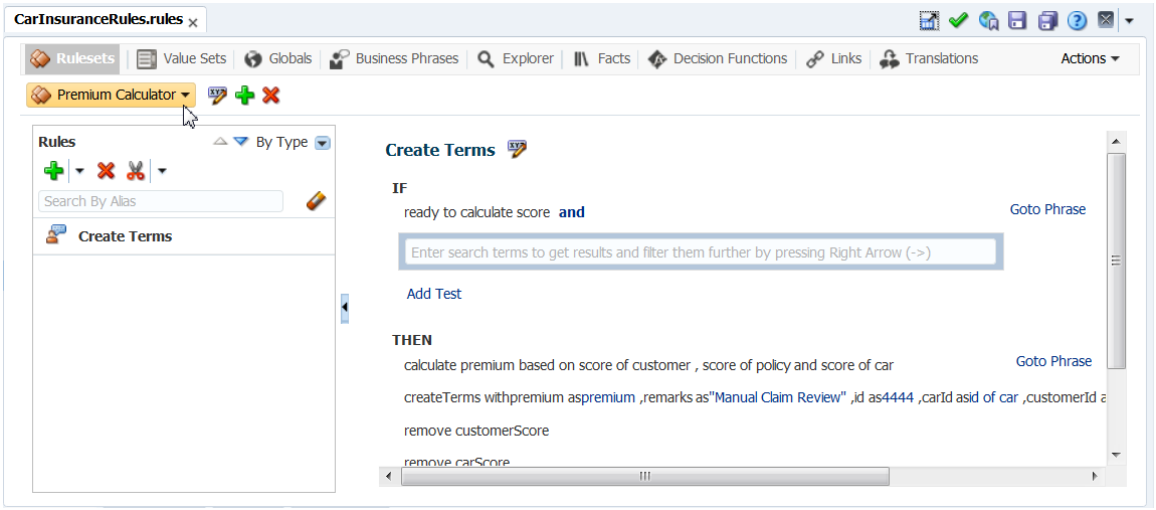


Table 12-3 Rulesets tab

Button	Description
Rulesets drop down	Click and select a ruleset from the list.
Advanced Property Editor	Click to edit properties in the pop-up Advanced Property Editor.
Add Ruleset	Click to add a Ruleset.
Delete Ruleset	Click to delete a Ruleset.

In the **Rules** master list, you can enter an alias and search for rules. Click **Clear** to clear the Search by Alias field. You can also sort rules--click the Sort Ascending or Sort Descending arrows to sort the IF/THEN detail panel.

Table 12-4 Rules master list buttons

Button	Description
Add	Click to add a new Decision Table, a Verbal Rule, or a General Rule.
Delete	Click to delete.
Cut, Copy, Paste	Click to Cut, Copy, or Paste.
	Click the Sort Ascending or Sort Descending arrows to sort the IF/THEN detail panel. Click By Type and choose an option to sort by any of these options: Type, Name, Active, Effective Start or End Date, Priority.
Search by Alias	Enter an alias name to search for rules by alias name.
 Clear	Click Clear to clear search results.

How to Edit Rules in an Oracle Business Rules Dictionary

Oracle SOA Composer enables you to edit the rules in a dictionary.

To edit a rule with Oracle SOA Composer:

1. In Oracle SOA Composer, with an Oracle Business Rules dictionary open, click the Rulesets tab.
2. Select the appropriate ruleset from the drop down and choose a rule from the list. The rule appears in the detail panel.

   Use the **Add, Delete, Cut, Copy**, and **Paste** buttons in the **Rules** toolbar to modify the rule.

   Note:

   The **Paste** button is disabled if the selection is multiple. The button is enabled only on single selected condition/action. When pasting, the copied/cut items are added at the end of the list.

3. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

How to Add a Rule

In Oracle SOA Composer you can add rules to a ruleset.

To add a rule in a ruleset:

1. In a session, the **Ruleset** tab, select a ruleset of interest.

2. In the rule area, click Add Rule and select to add either a Decision Table, Verbal Rule, or General Rule.
3. In the IF area, enter search terms to get results and filter them further by pressing the right arrow to create the condition.
4. In the THEN area for the rule, click Add Action to add the required action for this rule.
5. When done, click **Save Changes in Current Tab.**
6. If you are ready to apply the changes to the runtime version, click **Publish.**

How to Delete a Rule

In Oracle SOA Composer you can delete rules in a ruleset.

To delete a rule in a ruleset:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule detail area, locate the rule you want to delete and click **Delete**.
3. When done, click **Save Changes in Current Tab.**
4. If you are ready to apply the changes to the runtime version, click **Publish.**

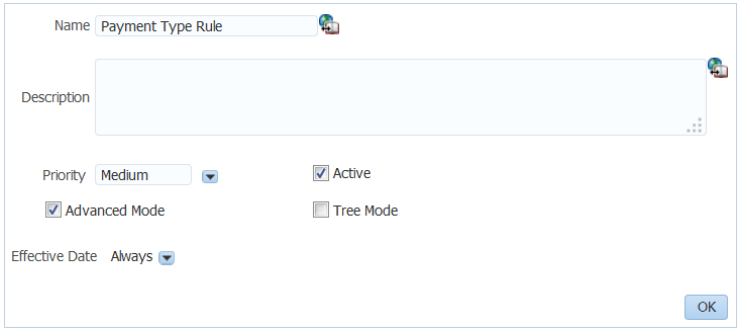
How to Show and Edit Advanced Settings for Rules

In Oracle SOA Composer you can edit advanced settings for rules in a ruleset. For more information on advanced settings, see Using Advanced Settings with Rules and Decision Tables.

To show and edit advanced settings in a rule:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area and locate the rule you want to show or change advanced settings. Expand the rule first, if necessary.
3. Click the **Advanced Property Editor** button next to the rule name. This displays the advanced settings dialog, as shown in Figure 12-26.

Figure 12-26 Advanced Properties Editor Dialog



How to Add Rule Conditions

In Oracle SOA Composer you can add conditions to a rule in a ruleset. Conditions within a rule use a tree representation. Use the toolbar at the top of the conditions tree to add, delete, cut, copy and paste. Within the condition tree, you can select a parent node and perform similar actions.

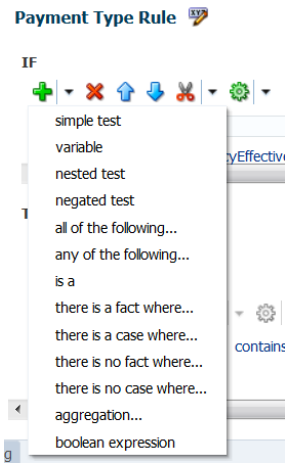
For more information on working with rule conditions, see Working with Rules.

To add rule conditions:

If no condition is selected, the condition is added at the end. If a condition is selected, a sibling to the selected condition is added.

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area, locate the rule where you want to add a condition.
3. Next to the existing rule condition, click the down arrow to display a list of options available for adding a condition as shown in Figure 12-27.

Figure 12-27 Adding a Condition



If the rule where you want to add a condition does not contain any existing condition, then you need to click the Add Test down arrow to display a list of available options for adding a condition as shown in Figure 12-27.

The following are some of the available options for adding a condition:

- simple test: Adds a simple test condition
- variable: Adds a variable definition. The variable and its value can be represented as an inline business term definition.
- (...): Adds a new simple test within a nested parenthesis
- not(...): Adds a new simple test within a NOT nested parenthesis

Each nesting level provides a list with the preceding options to operate on a nested block.

For more information on tests, see How to Work with Extended Tests.

How to Delete Rule Conditions

In Oracle SOA Composer you can delete conditions for a rule in a ruleset. For more information on working with rule conditions, see Working with Rules.

To delete rule conditions:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area, locate the rule where you want to delete a condition.
3. Next to the rule condition that you want to delete, click the down arrow, and then click **Delete Test** from the list.

   A separate list is available for each nesting level. So the delete operation can be performed on a single condition or a nested block.

How to Modify Rule Conditions

Using Oracle SOA Composer, you can edit conditions in a rule. You can select a rule condition for nesting or modify expression values within the condition. For more information on working with rule conditions, see Working with Rules.

To modify a condition in a rule:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area locate the rule where you want to modify conditions.
3. In the **IF** area, use the controls, buttons, and selection boxes, including the **Left Value** expression button, list for an operator, and **Right Value** expression button to modify the condition.

   Filtering is supported for expressions. For example, when you type `Employee`, values are filtered and the values with `Employee` are displayed in the drop-down. Use mouse or arrow keys to select a value.

You can use the Expression Builder, Condition Browser, Date Browser, and Right Operand Browser to edit the left and right-side expressions.

In addition to modifying the values, you can also change the form type of a condition. For example, a simple test can be changed to variable definition and so on. To change the form type of a condition, you need to select the condition by using the adjacent check box and select the required form type from the **Selected Tests** list.

How to Add Rule Actions

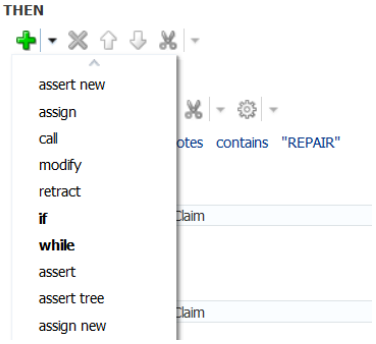
In Oracle SOA Composer you can add actions to a rule. For more information on working with rule actions, see Working with Rules.

To add rule actions:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area locate the rule where you want to add an action.

3. In the **THEN** area for the rule, next to the rule action click **Add Action**, as shown in Figure 12-28.

Figure 12-28 Rule Actions in a Ruleset



If the rule to which you want to add an action does not contain any existing action, then you need to click the **Add Action** button in the **THEN** area.

How to Delete Rule Actions

In Oracle SOA Composer you can delete actions in a rule. For more information on working with rule actions, see Working with Rules.

To delete rule actions:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area, locate the rule where you want to delete an action.
3. In the **THEN** area for the rule, select the action.

   Click **Delete Action**.

How to Modify Rule Actions

In Oracle SOA Composer you can modify actions in a rule. For more information on working with rule actions, see Working with Rules.

To modify rule actions:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area, locate the rule where you want to modify an action.
3. In the **THEN** area for the rule you can do the following:

   - Add and delete actions using Add and Delete buttons on the top.
   - Select the action and move it up and down using the respective arrow buttons.
   - Cut, copy and paste using the Cut, Copy and Paste buttons on the top.
   - Click the `More` link in the drop-down area to launch **Select a Target** popup and select a value.
   - Click the **Edit Properties** button next to the rule action and modify properties.

   The Properties dialog box is displayed where you can modify the property details.

For more information on number formatting in rules, see What You May Need to Know About Localized Number Formatting Support in Oracle SOA Composer.

How to Work with Advanced Mode Rules

In Oracle SOA Composer, you can work with advanced mode rules in a ruleset.

Note:

Advanced Mode capability has been maintained for backward compatibility only. We recommend that you use extended tests in simple mode to create any kind of condition that you need.

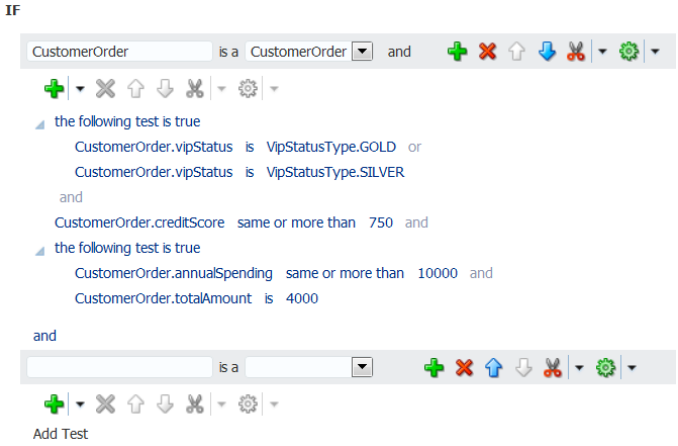
Everything that can be done in Advanced Mode can be done in simple mode. Advanced mode rules can be converted to equivalent simple mode rules simply by clearing the Advanced Mode check box.

For more information, see How to Work with Extended Tests.

To show and modify advanced mode rules:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area, locate the rule where you want to show or modify advanced mode rules.
3. Click **Advanced Property Editor** button to show advanced settings. For more information on showing advanced settings, see How to Show and Edit Advanced Settings for Rules.
4. If the **Advanced Mode** check box is not selected, then select the **Advanced Mode** check box. This shows the advanced mode rule options, as shown in Figure 12-29.

Figure 12-29 Showing Advanced Mode Rule Options



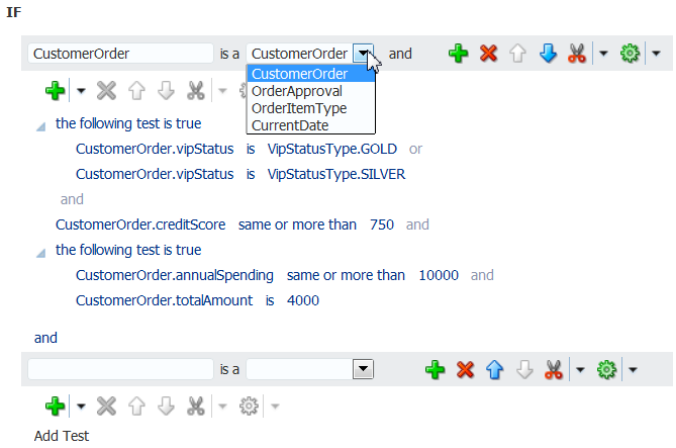
Working with Advanced Mode Options

The Advanced Mode rules options enables you to create, modify, and delete patterns, as well as add, modify, and delete conditions and actions within a pattern.

Using the Advanced Mode rule options, you can:

- Specify a pattern variable and select a fact type for the variable: You can directly enter the name of the pattern variable in the variable field. You can specify the fact type for the variable by using the fact type list as shown in Figure 12-30.

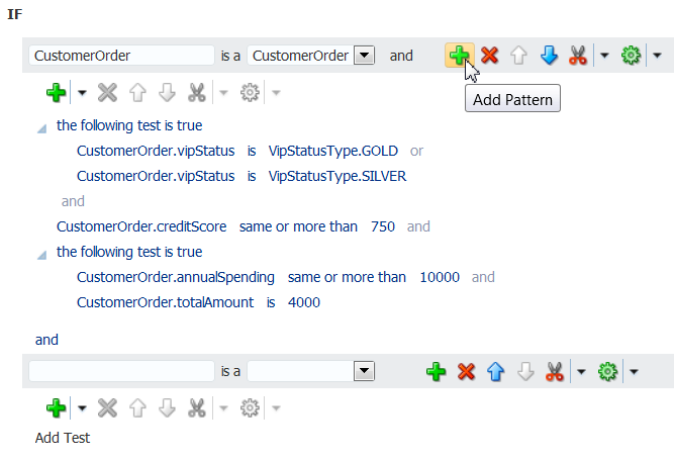
Figure 12-30 Specifying Pattern Variable and Fact Type



In the graphic example, `CustomerOrder` is a pattern variable of `CustomerOrder` fact type.

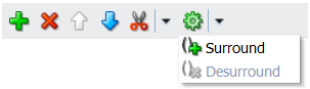
- Add a pattern: Click the **Add Pattern** button to create a pattern to the existing rule. Figure 12-31 displays an added pattern. The newly created pattern is blank.

Figure 12-31 Adding a Pattern



- Delete a pattern: Click the **Delete Pattern** button to delete a pattern from a rule.
- Specify connectives: Two or more patterns are joined by a connective, `and` or `or`. You can use the connective link to toggle between the connectives.
- Work with nested patterns: A nested pattern has patterns inside it. These are enclosed within curly braces ({}). The pattern operator list is followed by the open curly brace. You can create a nested pattern by clicking **Surround pattern with parentheses** button and you can remove the pattern nesting by clicking the **Remove parentheses from pattern** button as shown in Figure 12-32.

Figure 12-32 Adding and Removing Pattern Nesting



Inside the open curly brace, you can specify a pattern and then click the **Add Test** down arrow to add conditions to the nested pattern as well as add another pattern to the same pattern block.

A nested pattern block ends with a closing curly brace. You can have multiple levels of nested patterns, which means that inside a nested pattern, you can have another nested pattern. You can click the **Delete Nested Pattern Block** button to remove the entire nested pattern block.

When you nest a pattern, an operator list is displayed with **(for each case where)** selected as the default operator in the operator list. The other items are **there is a case where**, **there is no case where**, and **aggregate** and so on.

The user interface remains the same as **(for each case where)** when you select **there is a case where** or **there is no case where** as the operator. However, when you select **aggregate**, the user interface changes. For an aggregate operator, you *must* enter a variable in the available field and select a function from the function list. The function list displays the following:

- `count`
- `average`
- `maximum`
- `minimum`
- `sum`
- `collection`

Except for the `count` function, all the other functions require an expression. You can specify an expression in the available field or launch the Condition Browser window.

In the Advanced Mode of rules, in the THEN part, you can add any required action ranging from `assert`, `call`, `modify` to even conditional actions such as `if`, `else`, `elseif`, `while`, `for`, `if (advanced)`, and `while (advanced`.

How to Work with Extended Tests

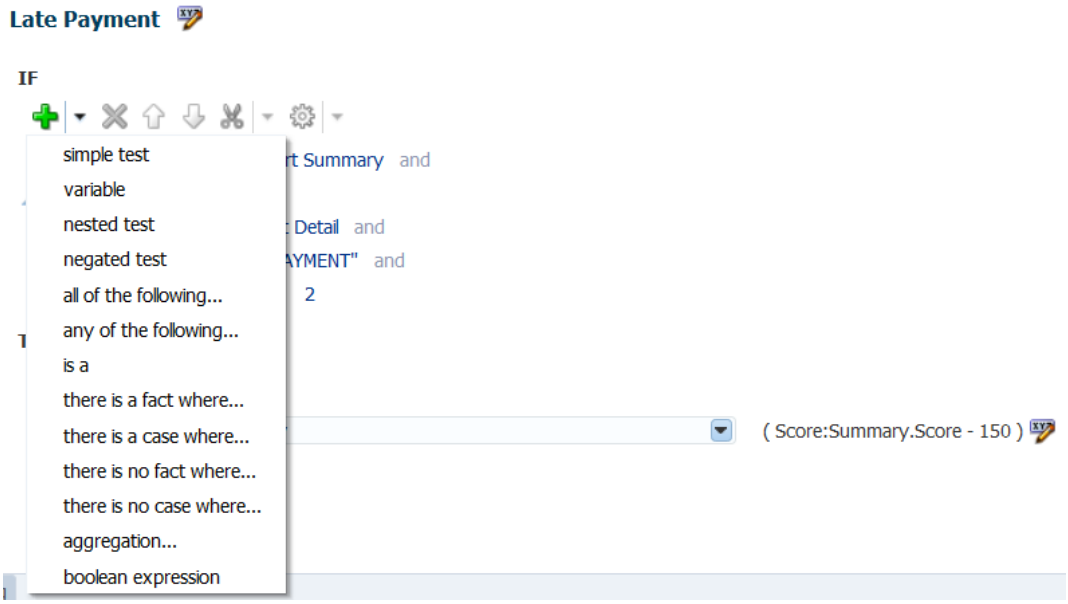
Extended tests should be used when building complex rules. Extended tests, or Simple Mode, replaces Advanced Mode rules.

Note:

Advanced Mode capability has been maintained for backward compatibility only.

Everything that can be done in Advanced Mode can now be done in Simple Mode. The UI has been streamlined and improved to enable you to more easily create complex rules and tests, as shown Figure 12-33

Figure 12-33 List of Extended Tests



Advanced mode rules can be converted to the equivalent simple mode rules by clearing the Advanced Mode check box in the Advanced Property Editor.

Extended tests are only applicable to general rules, decision tables, and while defining business phrases. They are not visible in verbal rules.

In addition to the original four tests (shown first in Table 12-5) there are new forms:

Table 12-5 Extended Tests

Forms	Description
simple test	This is the building block for conditions. Compares a value against another value, range or set. For example: Emp.salary > 1000
variable	Initializes variables. For example: age = Duration.years between(Emp.birthdate,RL.date.get current())
nested test	Encapsulates tests in a containing block. For example: (age > 50 or Emp.salary > 50000)
negated test	Negates a test. For example: not(age > 50 and Emp.salary > 50000)
all of the following	all of the following are true. For example: (age > 50 and Emp.salary > 50000)

any of the following	some of the following are true.For example:
	`IF`
	`e is a Emp and there is no Emp where Emp.salary < e.salary <insert test> <insert test>THEN assign e.isLowestPaid = true`

Table 12-5 (Cont.) Extended Tests

Forms	Description
is a	Defines a fact. For example: e **is a** Emp

there is a case where	This test has 1 or more child tests that are ANDed.
	The child tests are all true for at least 1 case. A case is a binding of facts to contained **is a** tests.
	Must have **is a** descendant.
	Example:
	`There is a case where`
	`e is a Emp and`
	`d is a Dept and`
	`e.salary > 1000000 and`
	`d.name == "Marketing" and`
	`d.employees contains e`

there is a <factType1>,...<factTypeN> where#*
This test has N or more child tests that are ANDed
	Hidden <factType> **is a** <factType> tests as first N children.
	The child tests are all true for at least 1 case.
	It is legal to have no visible child tests, in which case the where keyword should be suppressed.
	Example:
	`IF`
	`  there is a Emp, Dept where`
	`  Emp.salary > 1000000 and`
	`  Dept.name == "Marketing" and`
	`  Dept.employees contains Emp`
	`THEN`
	` call print "there is a highly paid marketer!"`
	`IF`
	`  there is a Emp`
	`THEN`
	`  call print "somebody works here!"`

there is no case where	This test has 1 or more child tests that are ANDed. The child tests are true for no case (no binding of facts to contained is a tests satisfy all the other tests). Must have **is a** descendant.
there is no <factType1>,...,<factTypeN> where	Hidden <factType> **is a** <factType> as first N children The child tests are true for no case

Table 12-5 (Cont.) Extended Tests

Forms	Description

aggregation	This test has 0 or more child tests that are ANDed.
	Must have **is a** child (may be hidden).
	v is the sum | average | minimum | maximum | count | collection of <expression> where
	Where clause omitted when there are no visible child tests.
	`IF`
	`  number of employees is the count of Emp`
	`THEN`
	`  call print "number of employees: " + number of employees`
	
	`IF`
	`  number of male employees is the count of Emp where Emp.gender == "M"`
	`THEN`
	`  call print "number of male employees: " + number of male employees`
	
	Note that in both rules above, the SDK will create a hidden nested **is a** test for Emp.
	You can also use an explicit **is a**
	`IF`
	`  number of male employees is the count of e where e is Emp and e.gender == "M"`
	`THEN`
	`  call print "number of male employees: " + number of male employees`

boolean expression	Captures a boolean expression. For example: isEligible(Emp)

Figure Figure 12-34 shows an example of "there is a case where" form:

Figure 12-34 Extended Test Example 1

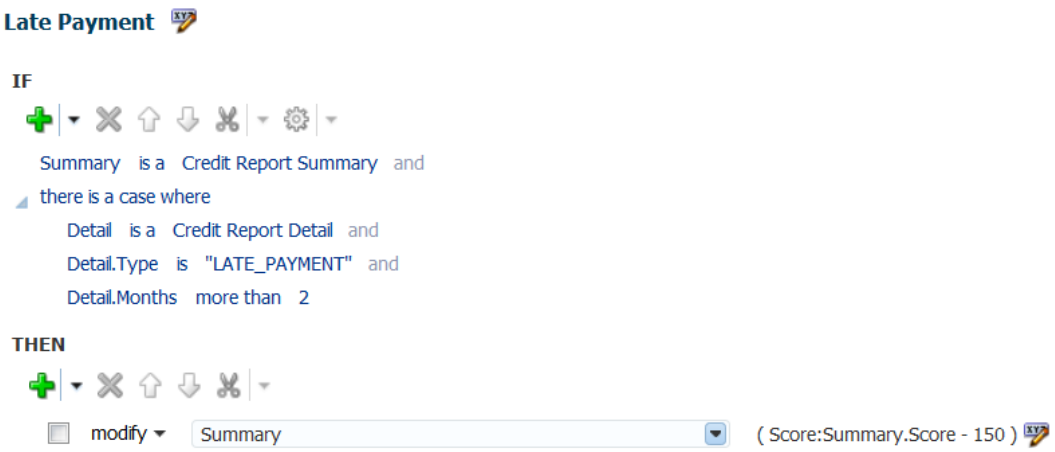
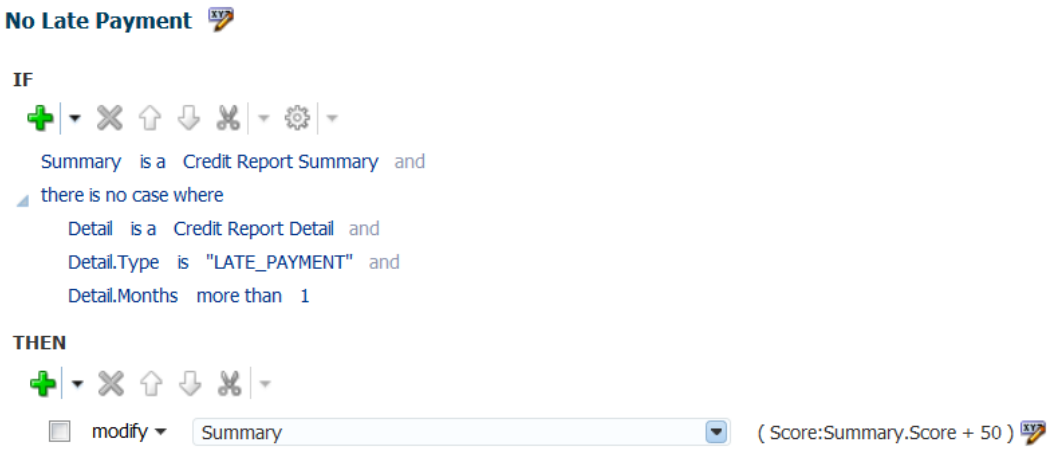


Figure Figure 12-35 shows an example of "there is no case where" form:

Figure 12-35 Extended Test Example 2



For information about how to build complex rules, see How to Add Rule Conditions.

For more information about Advanced Mode, see How to Work with Advanced Mode Rules.

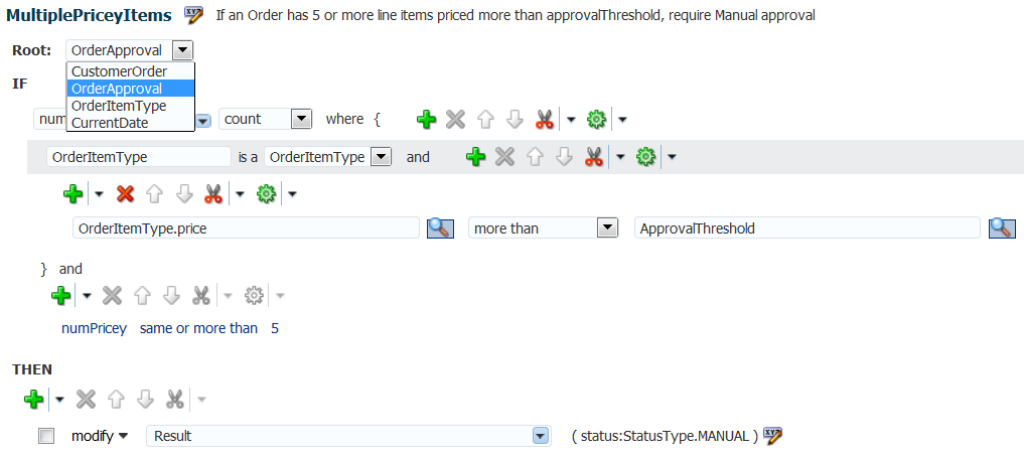
How to Work with Tree Mode Rules

In Oracle SOA Composer you can work with tree mode rules in a ruleset. For more information on working with tree mode rules, see Working with Tree Mode Rules.

To show and modify tree mode rules:

1. In a session, the **Ruleset** tab, select a ruleset of interest.
2. In the rule area locate the rule where you want to show or modify tree mode rules.
3. Select **Advanced Property Editor** button to show advanced settings.
4. If the **Tree Mode** check box is not selected, then select the **Tree Mode** check box. This shows the tree mode rule options, as shown in the ROOT area in Figure 12-36.

Figure 12-36 Showing the Tree Mode Rule Area in a Rule



What You May Need to Know About Oracle Business Rules Editor Declarative Component

You can use the Oracle Business Rules Editor composite declarative component to leverage the functionality of editing business rules in any ADF-based Web application. It enables you to edit business rules available in rulesets by using the Rules SDK2 API.

For more information on Oracle Business Rules Editor, see Using the Oracle Business Rules Editor Declarative Component in *Developing SOA Applications with Oracle SOA Suite*.

What You May Need to Know About Oracle Business Rules Dictionary Editor Declarative Component

The Oracle Business Rules Dictionary Editor is a composite declarative component that can be embedded in any ADF-based web application. It enables you to edit business rules metadata artifacts, such as globals, value sets, and rulesets, by using the Rules SDK2 API.

For more information on Oracle Business Rules Dictionary Editor, see Using the Oracle Business Rules Dictionary Editor Declarative Component in *Developing SOA Applications with Oracle SOA Suite*.

What You May Need to Know About Oracle Business Rules Dictionary Editor Task Flow

The Oracle Rules Dictionary Editor Task Flow is basically a wrapper around the Rules Dictionary Editor declarative component. The task flow is used in ADF-based web applications that require a task flow instead of a declarative component.

For more information on Oracle Business Rules Dictionary Editor Task Flow, see Using the Oracle Business Rules Dictionary Editor Task Flow in *Developing SOA Applications with Oracle SOA Suite*.

Using the Oracle SOA Composer Browser Windows

Oracle SOA Composer provides browser windows that helps you to work with different types of expressions such as rule expressions, XPATH expressions, date expressions, and so on.

The different types of browsers provided by Oracle SOA Composer are:

- Expression Builder
- Condition Browser
- Date Browser
- Right Operand Browser

Expression Builder

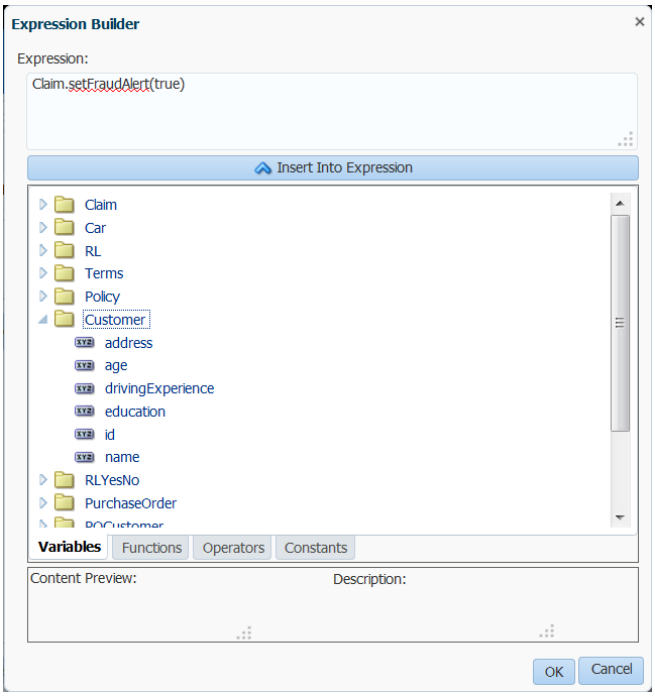
Expression Builder is used to build different types of expressions such as XPATH expressions, rule expressions, and so on.

Expression Builder has a field where you can enter the expression directly. It has four tabs: Variables, Functions, Operators, and Constants. Each of these tabs display data in a tree structure. The Variables tab displays all the variables in the rules meta-data. The Functions tab displays all the functions in the rules meta-data. The Operators tab

displays operators such as +, -, *, and so on. The Constants tab displays all the constants that exist in the rules meta-data.You can switch between the tabs, select an item in the tree, and click the **Insert Into Expression** button to insert the selected item at the cursor position in the expression field. When an item is selected in the tree, the **Content Preview** and the **Description** areas display more information about the selected item. Once you create the expression and click **OK**, the newly created expression appears in the field that is available to the left of the expression builder button.

Figure 12-37 displays the Expression Builder browser.

Figure 12-37 The Expression Builder Browser



Condition Browser

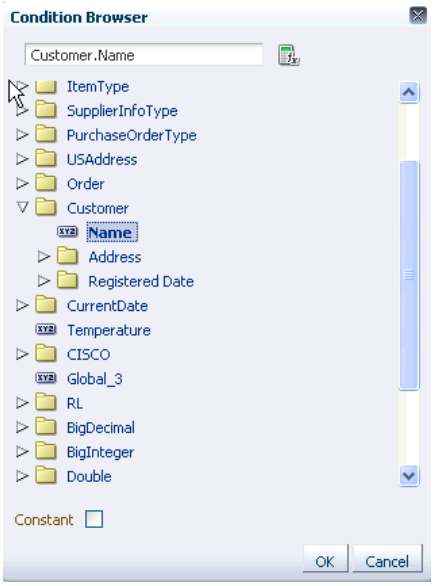
The Condition Browser has a field, a hierarchical tree, and an Expression Builder embedded inside it. You can enter the expression directly in the field, or select an item from the tree. Condition Browser supports filtering. For example, when you start entering `customer` the tree is narrowed down to items with `customer`.

When an item is selected in the tree, the new selection appears in the field immediately. You can also use the embedded Expression Builder to create an expression.

Once the Expression Builder is launched and an expression is created, the new expression appears in the Condition Browser field. Once you create an expression and click the **OK** button in the Condition Browser, the newly created expression appears in the field that is to the left of the Condition Browser button.

Figure 12-38 displays the Condition Browser.

Figure 12-38 The Condition Browser



Date Browser

The Date Browser is used to select a Literal Date or a Date Expression. The Date Browser has two options to switch between a Literal Date and a Date Expression. When one option is selected, the other one is disabled.

Select:

- Literal Date option to enter a date using a Calendar pop-up.
- Date Expression option to enter the expression directly in the Date Expression field or to launch the Condition Browser to select a date expression.

Figure 12-39 displays the Date Browser.

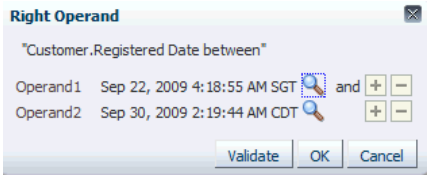
Figure 12-39 The Date Browser

Right Operand Browser

The Right Operand browser is used to select multiple right expressions. The browser displays operands in each row. You can enter an expression directly in the operand field or launch the Condition Browser to select an expression. The **+** button adds a row after the current one. The **-** button deletes the current row. These buttons are enabled and disabled based on the selected operator. For instance the in operator allows multiple right expressions. So in this case, the buttons are enabled.

Figure 12-40 displays a Right Operand browser.

Figure 12-40 The Right Operand Browser



Note:

Using Right Operand browsers, you can enter multiple values for the right-side expression. However, you can place a Date browser outside a Right Operand browser, and in which case, only one expression can be entered. For both these browsers, you cannot enter values directly in the right-side expression field. Once you have entered values using the browser and clicked OK, the values get added as comma-separated values on the Rules UI.

Editing Decision Tables in an Oracle Business Rules Dictionary

When Oracle SOA Composer is in a session, you can edit, add, and delete a Decision Table in a ruleset.

For more information on how to use sessions, see Creating and Publishing Sessions.

You can edit the description of a rule/condition or action within a decision table. If you hover over a condition, a right arrow is used to select the condition. If you click on the condition value, a pop-up appears where you can edit the description.

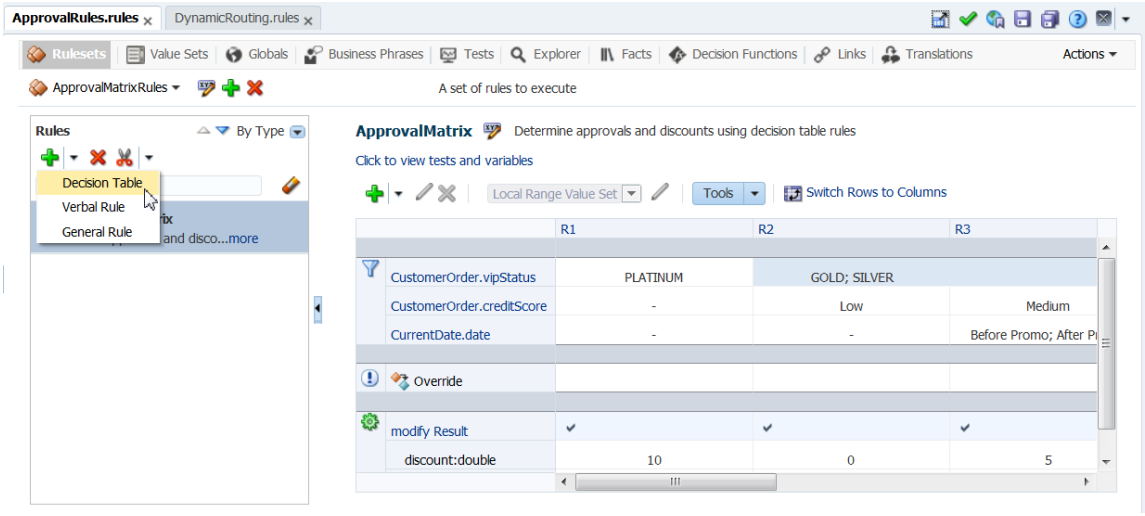
Adding a Decision Table

In Oracle SOA Composer, you can add a Decision Table to a ruleset. For more information on working with Decision Tables, see Introduction to Working with Decision Tables.

To add a Decision Table in a ruleset:

1. In a session, select a ruleset of interest.
2. In the ruleset area, click **Add** and then **Add Decision Table**, as shown in Figure 12-41. An empty Decision Table appears.

Figure 12-41 Adding a Decision Table in a Ruleset



Adding Condition Rows to a Decision Table

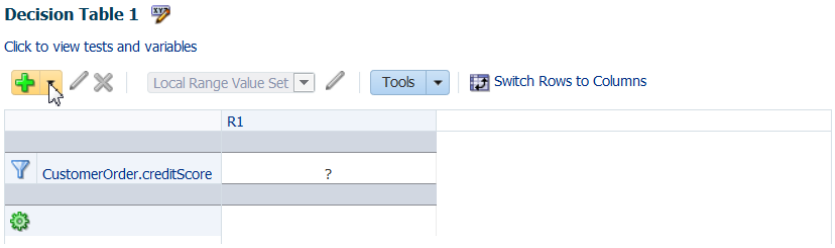
Using Oracle SOA Composer, you can add condition rows to a Decision Table.

To add condition rows to a Decision Table:

1. In the Decision Table toolbar, from the list next to the **Add** button, select **Add Condition** that displays the Condition Browser window where you can specify or select conditions.

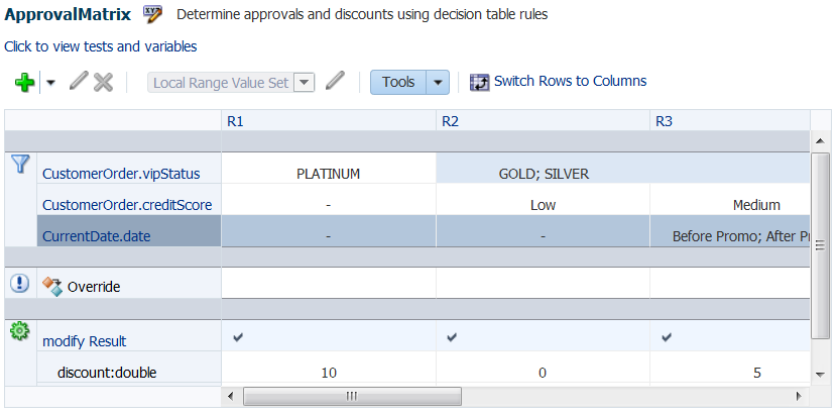
   The selected or specified condition row and a Rules column with the header R1 is added to the table; the cell below R1 has a "?" symbol (Figure 12-42). The "?" symbol indicates that the cell does not have a value yet.

Figure 12-42 New Condition Row Added in a New Decision Table



   If you are adding a condition to a table that has existing condition rows, similar to adding a condition to a blank Decision Table, Oracle SOA Composer prompts for specifying the condition details. Once the details are provided, the specified condition is added as the last condition row; the condition cells under each rule column in the new row also have "?" symbols.

Figure 12-43 New Condition Row Added As Last Row in a Decision Table



For information about all symbols that might be used in a decision table, see Editing Decision Table Cells.

2. If you want to edit a specified condition, in the **Conditions** area, click the condition row, and then click the **Edit Condition** button on the toolbar. This displays the Condition Browser.

3. Enter an expression by clicking in the Conditions Browser to select a variable, or click the **Expression Builder** button to display the Expression Builder.

   Expression Builder lets you build expressions.

4. Each condition row requires a value set from which to draw the values for each cell. When the value you select has an associated global value set, then by default the value set is associated with the condition row.

   If there is no global value set associated with the value, then after you add a condition row to a Decision Table, you need to either specify an existing global value set or create a Local List of Values or a Local List of Ranges value set.

   To associate a value set for the condition, perform either of the following:

   - In the **Conditions** area, select the condition, and select an existing value set from the **Select Value Set** list.
   - In the **Conditions** area, select the condition, and select either **Local List of Values** or **Local List of Ranges** (as relevant) from the **Select Value Set** list.

   You can edit the value set for the selected condition by clicking the **Edit Value Set** button.

   This displays the Value Set Editor where you can add, edit or delete values. If editing a Local List of Values value set, you can also reorder values in the value set.

   For more information on number formatting in value sets, see What You May Need to Know About Localized Number Formatting Support in Oracle SOA Composer.

5. Repeat **Step 2** through **Step 5**, as required to add additional condition rows in the Decision Table.

For more information on adding condition rows, see How to Add Condition Rows to a Decision Table.

Adding Actions to a Decision Table

In Oracle SOA Composer, you can add actions to a Decision Table.

To add actions to Decision Table:

1. From Oracle SOA Composer, select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to add actions.
2. From the list next to the **Add** button, select **Add Action** and select an available action from the list. For example, click **Modify** as shown in Figure 12-44.

Figure 12-44 Adding Actions to a Decision Table

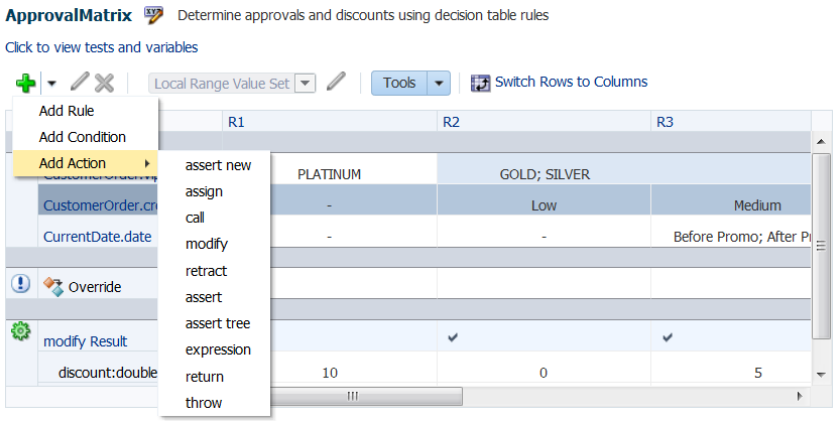


Table 5-1 in Working with Decision Tables, lists the available actions.

3. In the Action Editor window, select the action target and then specify values for an action cell.

   For more information on number formatting in value sets, see What You May Need to Know About Localized Number Formatting Support in Oracle SOA Composer.

For more information on adding actions to Decision Tables, see How to Add Actions to a Decision Table.

Adding Rules to a Decision Table

Using Oracle SOA Composer, you can add a rule to a Decision Table.

To add a rule to a Decision Table:

1. In a session, select a ruleset of interest, select the Decision Table where you want to add the rule.
2. In the Rules master list, select the Decision Table where you want to add the rule. Next to the **Add** button in the detail section, select **Add Rule**.

   A new column for the added rule is displayed.

   Notice that the new rule is added as the first rule of the Decision Table and the other rules have moved as required to keep the values in their defined order. This is because **Order Rules By Bucket** is enabled by default, which means rule ordering in a Decision Table is set according to the relative position of values

associated with a condition expression. If **Order Rules By Bucket** is not enabled when you add a rule, the new rule is added as the last rule of the Decision Table. In either case, the cells in the new rule column have "?" symbols, indicating the cells do not have values yet.

For information about all symbols used in a table, see Editing Decision Table Cells. For additional information about rules ordering, see Controlling the Order of Rules in a Decision Table.

3. Enter values for the condition cells by clicking the cells.
4. Click an Action row to enter values for the action cells.

Note:

If because of the inadequate column width, you cannot view the complete contents of a cell in a Decision Table, you can roll your mouse pointer over the cell to view the contents. Also, click the **Maximize tabs section** button in the toolbar to increase the view.

Editing Decision Table Cells

Each rule in a Decision Table contains cells pertaining to three sections: Conditions, Conflicts, and Actions.

Working with Condition Cells

In view mode, a condition cell with a "?" symbol indicates that the cell does not have a condition value. If a cell has two or more values specified, a semicolon-separated list of values is displayed in the cell.

In the editable mode the condition cells display specified condition values in multichoice lists. When editing a new rule or when a condition value is unspecified, the condition cell is blank.

If you select **All**:

- When the particular condition cell is clicked, the cell displays "All"
- When the particular condition cell is not selected, the cell displays the "-" symbol

You can select any value that is available in the condition value list.

Note:

When you edit the condition cells, if **Order Rules By Bucket** is selected, the Decision Table is refreshed and the edited rule column may shift to the left or right depending on the selected condition cell value. Click the **Tools** drop down to select **Order Rules By Bucket.**

Note:

You can modify the value set associated with a a condition, by clicking the condition. This enables the value set list and the **Edit Value Set** button so that you can edit the associated value set.

Working with Action Cells

When you add an action, an action row is created with the specified action type. There are two types of action cells:

- The Action form cells contain check boxes. When a rule fires, only selected actions are executed. In Figure 12-45, R1 and R3 action check boxes are selected whereas the other action check boxes are cleared. In this case, if R1 fires, the action will be executed, but if R2 fires, then the action will not be executed.

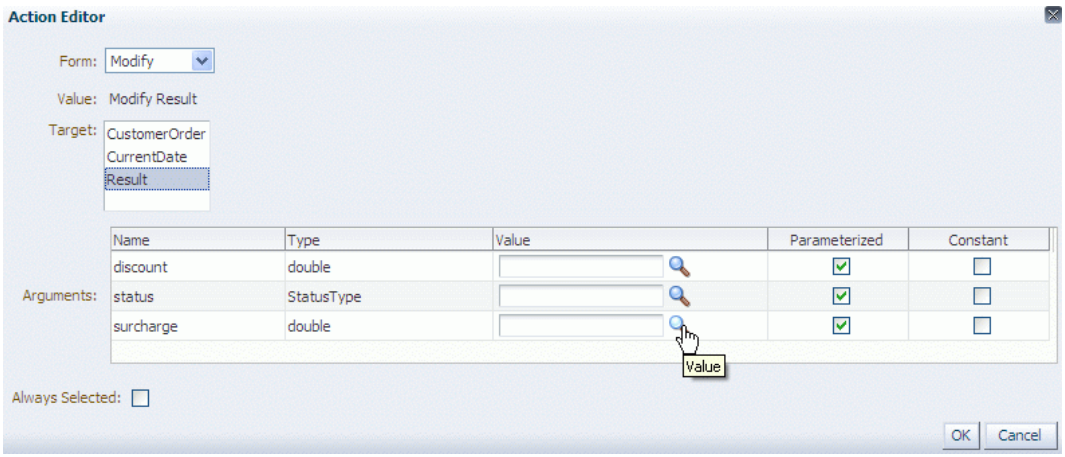
Note:

The **Edit Action** button is enabled only if the action form cell row is selected. The **Edit Action** button invokes the Action Editor window.

- The Action parameter cells contain the parameters of the action form. You can directly enter the action parameter values in the respective field or you can invoke the Condition Browser window to select a value.

Figure 12-45 displays the Action Editor window where you can select the values for an action parameter cell. If you select the **Always Selected** check box, all the check boxes for the particular action form get selected. All the check boxes pertaining to the action form are also disabled, because the specified action "is always selected".

Figure 12-45 The Action Editor Window



Note:

You can delete all the condition cells and all the action cells of a Decision Table at one go. Clicking the Conditions or the Actions box selects all the conditions or actions in the Decision Table respectively.

You can then click the **Delete** button on the Decision Table toolbar to delete the conditions or actions.

Controlling the Order of Rules in a Decision Table

By default the **Order Rules by Bucket** check box is enabled in a Decision Table. This means the order of the values in the value set associated with a condition row

determines the order of the condition cells, and thus the order of the rules. Click the **Tools** drop down to select **Order Rules By Bucket.**

To change the order of rules in a Decision Table, you need to change the order of values in the value set. For example, you can control rule ordering in a Decision Table by changing the relative position of the values in an LOV value set associated with a condition row. Note, however, that you cannot reorder range value sets.

When the **Order Rules by Bucket** check box is selected in a Decision Table and you add a rule, by default the new rule is added as the first rule column; the other rule columns move as required to keep the value set values in their defined order. When the **Order Rules by Bucket** check box is not enabled and you add a rule, the new rule is added as the last rule column. If you now select the **Order Rules by Bucket** check box, the newly added rule shifts to the first column.

Deleting Rules in a Decision Table

You can delete one or multiple rules in a Decision Table.

To delete rules in a Decision Table:

1. Select the rules column that you want to delete.

   If you want to delete more than one rule, press the Ctrl key, and by keeping the key pressed, select the other rule columns

2. Click the **Delete** button.

Defining Tests in a Decision Table

In Oracle SOA Composer, you can define tests in a Decision Table by adding conditions to facts. For more information about defining tests and working with rule conditions, see Working with Rules.

Note:

To add more complex conditions to facts, see How to Work with Extended Tests.

To add tests to a Decision Table:

1. From Oracle SOA Composer, select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to add a test.
2. Click the **Advanced Properties Editor** button next to the Decision Table name. If **Advanced Mode** is selected, clear the check box.
3. Click the **Click to view tests and variables** link under the Decision Table name.
4. Click the down arrow next to the **Add** button and select any of the options according to your requirements.
5. Use the field controls or **Left Value** and **Right Value** buttons, and the operator list to create the condition expression.

Note:

If a Decision Table already contains test conditions, you can add new test conditions by clicking Add at the end of an existing condition and selecting the required test form type.

Splitting and Compacting a Decision Table

You can modify the contents of a Decision Table to create a table that includes a complete set of rules for all cases, or a table that provides the least number of rules for the cases. The split and compact operations enables you to manipulate the contents in a Decision Table.

The split table operation creates a rule for every combination of values across the conditions. For example, in a Decision Table with 2 boolean conditions, 2 x 2 = 4 rules are created. In a Decision Table with 20 boolean conditions, 2**20 ~ 1 million rules are created. So, you only use split table when the number of rules created is small enough that filling in the action cells is feasible.

Using Oracle SOA Composer, split can be applied to an entire Decision Table. However, you cannot perform split operation on an individual condition row or cell.

To split or compact a Decision Table:

1. From Oracle SOA Composer, select a ruleset from the **Rulesets** navigation tab and select the Decision Table that you want to split or compact.
2. Click the **Split Table** button or **Compact Table** button on the **Tools** drop down.

Using Oracle SOA Composer, you can compact a Decision Table by merging conditions of rules with identical actions. So, compacting a table enables you to remove conditions from a Decision Table. However, using Oracle SOA Composer, you cannot merge two or more condition cells.

For more information on splitting and compacting Decision Tables, see Introduction to Decision Table Operations.

Checking for Missing Rules in a Decision Table

In a Decision Table, a missing rule is also called a "gap." A gap in a Decision Table occurs when a rule does not cover some combinations of values, one from each condition.

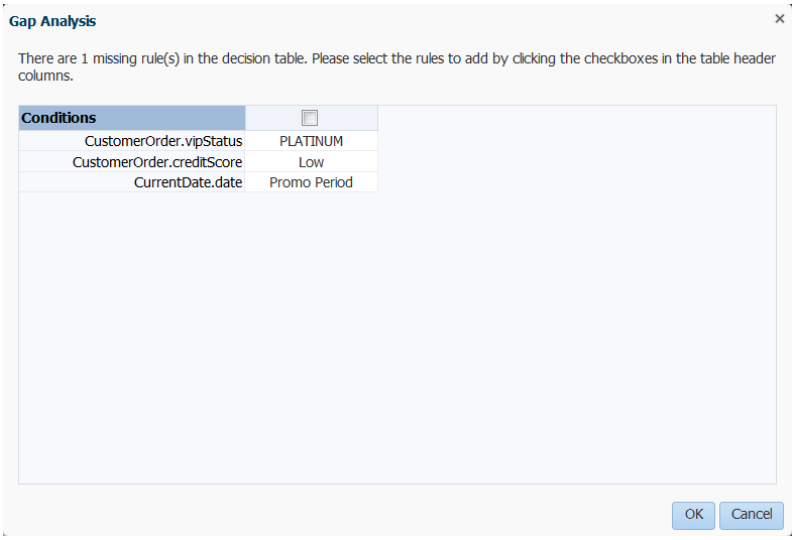
Using Oracle SOA Composer, you can check for missing rules in Decision Tables.

To check for missing rules:

1. From Oracle SOA Composer, select a ruleset from the **Rulesets** navigation tab and select the Decision Table in which you want to check for missing rules.
2. Select **Gap Analysis** from the Tools drop down.

   The Gap Analysis window is displayed as shown in Figure 12-46. You can select the rules that need to be added to the Decision Table.

Figure 12-46 The Gap Analysis Window



For more information about checking for missing rules, see How to Perform Decision Table Gap Checking.

Performing Conflict Resolution in Decision Tables

Rules in a Decision Table can conflict when they overlap and have different actions. Two rules overlap when at least one of their condition cells has a value in common. However, overlap without conflict is common and harmless. For more information about conflicts in Decision Tables, see Understanding Decision Table Conflict Analysis.

Using Oracle SOA Composer, you can find and resolve conflicts in a Decision Table.

To perform conflict resolution in a Decision Table:

1. From Oracle SOA Composer, select a ruleset from the **Rulesets** navigation tab and select the Decision Table on which you want to perform the Conflict Resolution.
2. Ensure that the **Show Conflicts** button is selected on the Tools drop down.
3. Click the **Advanced Property Editor** button next to the Decision Table name.
4. Ensure that **Conflict Policy** is set to **Manual** in the Advanced Settings area. This is the default conflict policy.

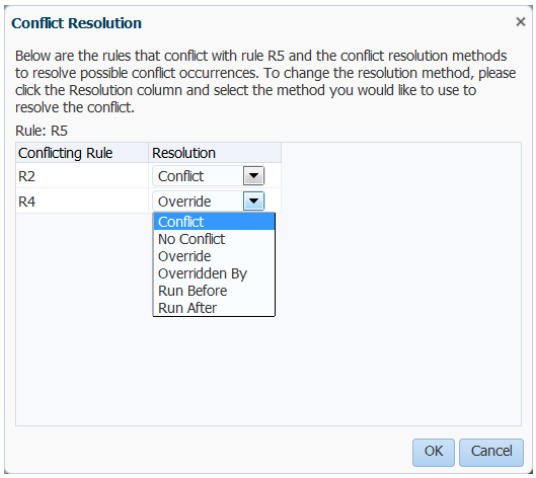
Note:

For more information on conflict policies, see Understanding Decision Table Conflict Analysis.

---

5. Select the **Conflict** row and then click the rule that has a conflict to display the **Conflict Resolution** window.
6. In the Conflict Resolution window, for each conflicting rule, in the **Resolution** field select a resolution from the list and click **OK** as shown in Figure 12-47.

Figure 12-47 Conflict Resolution Dialog



For more information about the conflict resolution options in Decision Tables, see Understanding Decision Table Conflict Analysis.

Switching From Rows to Columns

In Oracle SOA Composer, you can turn the rows in a Decision Table to columns by clicking the **Switch Rows to Columns** link in the detail area. This enables the rules to be displayed as rows, and the conditions, actions, and conflicts to be displayed as the columns.

Switching rows to columns provides ease of navigation when a Decision Table has many rules because you can see all the rules together and you do not need to "page the columns" for viewing the rules.

Figure 12-48 displays a Decision Table before the switch operation.

Figure 12-48 A Sample Decision Table

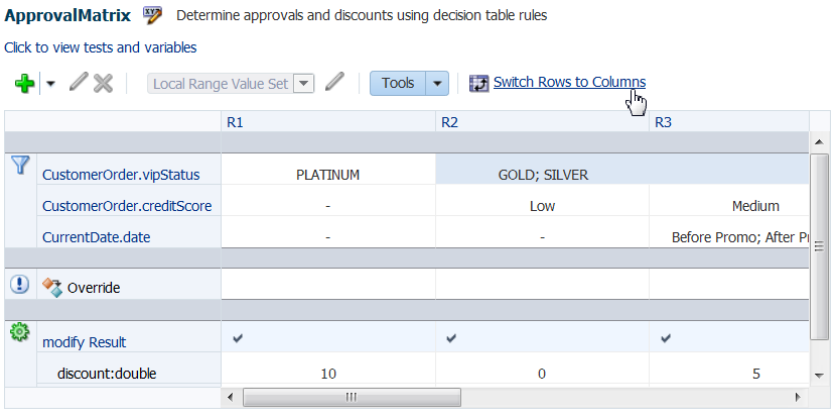
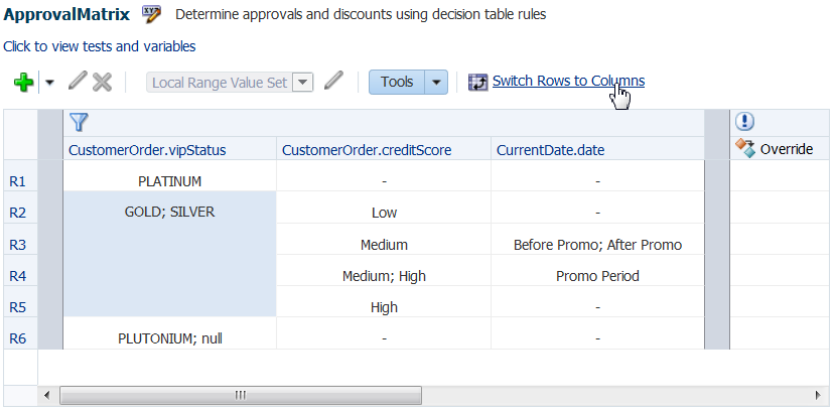


Figure 12-49 displays the sample Decision Table after switching the rows to columns.

Figure 12-49 Switching Rows to Columns



Working with Advanced Mode Options in a Decision Table

In Oracle SOA Composer, you can use advanced mode rules in a Decision Table just like you can work with advanced mode rules in a ruleset. The Advanced Mode rules options enable you to create, modify, and delete patterns, as well as add, modify, and delete conditions and actions within a pattern.

Note:

Advanced Mode capability has been maintained for backward compatibility only. We recommend that you use extended tests in simple mode to create any kind of condition that you need.

Everything that can be done in Advanced Mode can be done in simple mode. Advanced mode rules can be converted to equivalent simple mode rules simply by clearing the Advanced Mode check box.

For more information, see How to Work with Extended Tests.

To show and use advanced mode options:

1. From Oracle SOA Composer, select a ruleset from the **Rulesets** navigation tab and select the Decision Table on which you want to add more complex rules.
2. Click the **Advanced Property Editor** button next to the Decision Table name.
3. Select **Advanced Mode**.

   The advanced mode options in a Decision Table are similar to the advanced mode options in a ruleset. For more information, see Working with Advanced Mode Options.

Deleting a Decision Table

In Oracle SOA Composer, you can delete Decision Tables in a ruleset. For more information on working with Decision Tables, see Introduction to Working with Decision Tables.

To delete a decision table in a ruleset:

1. In a session, select a ruleset of interest.
2. In the Rules master list, click the Decision Table you want to delete.
3. Click **Delete**.
4. When done with changes, click **Save Changes in Current Tab.** If you are ready to apply the changes to the runtime version, click **Publish.**

Editing Decision Tables in Microsoft Excel

Business users may find that editing Decision Tables is easier to do in Microsoft Excel. New functionality enables both developers and business users to export and edit Decision Tables in Excel and then import the Decision Tables back into the dictionary.

When exporting Decision Tables in Microsoft Excel, only basic Action types such as Assert New, Modify, Assign, Retract, and Call are supported.

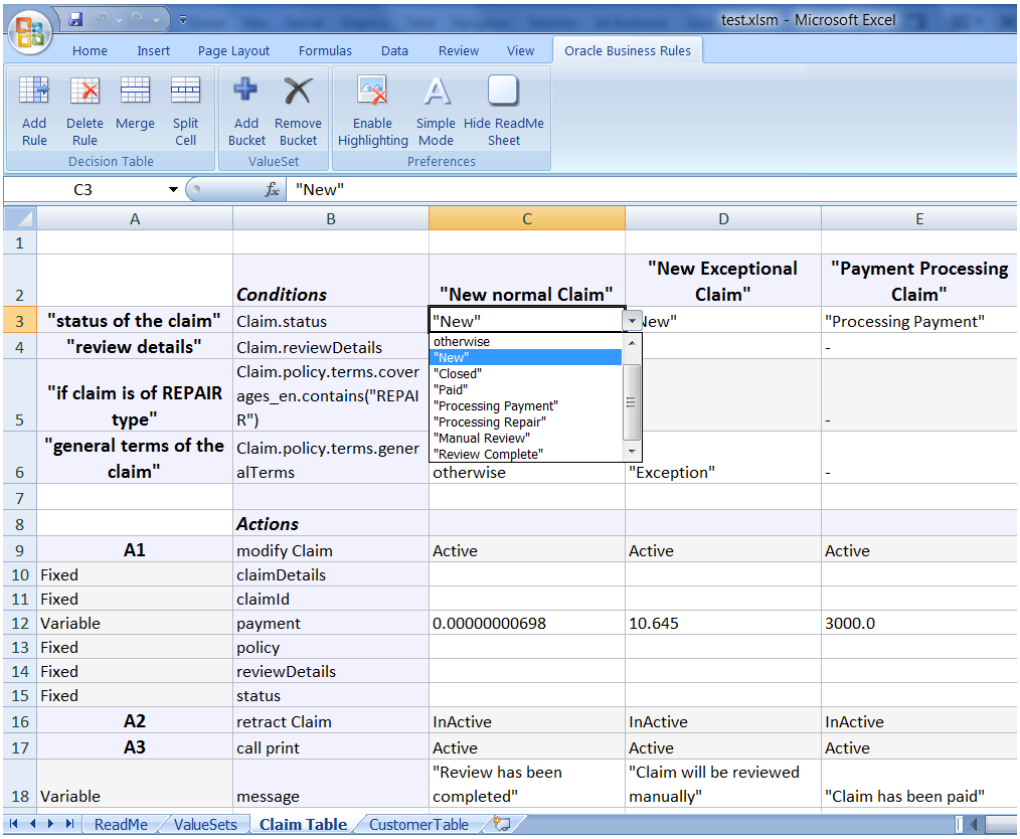
You can export and edit Decision Tables at design-time in Oracle JDeveloper or Business Process Composer. At runtime, you can export and edit in SOA Composer. You can export one or more Decision Tables from a Rule dictionary to the same Excel workbook.

When you import back into the dictionary, you can create a new dictionary, overwrite the existing dictionary, or perform a Diff-Merge. The Diff-Merge enables you to compare dictionaries.

For more information about comparing dictionaries, see Comparing and Merging Oracle Business Rules Dictionaries.

The Excel workbook structure consists of several worksheets: a Readme sheet, a Value Set sheet, and one sheet for each exported Decision Table, as shown in Figure 12-50. Only Rules and Value Sets can be edited in Excel. You can export to .xlsm (default) or .xls.

Figure 12-50 Excel Workbook



	A	B	C	D	E
1					
2		***Conditions***	**"New normal Claim"**	**"New Exceptional Claim"**	**"Payment Processing Claim"**
3	**"status of the claim"**	Claim.status	"New"	lew"	"Processing Payment"
4	**"review details"**	Claim.reviewDetails			-
5	**"if claim is of REPAIR type"**	Claim.policy.terms.coverages_en.contains("REPAIR")			-
6	**"general terms of the claim"**	Claim.policy.terms.generalTerms	otherwise	"Exception"	-
7					
8		***Actions***			
9	**A1**	modify Claim	Active	Active	Active
10	Fixed	claimDetails			
11	Fixed	claimId			
12	Variable	payment	0.00000000698	10.645	3000.0
13	Fixed	policy			
14	Fixed	reviewDetails			
15	Fixed	status			
16	**A2**	retract Claim	InActive	InActive	InActive
17	**A3**	call print	Active	Active	Active
18	Variable	message	"Review has been completed"	"Claim will be reviewed manually"	"Claim has been paid"

When you open the spreadsheet, the macros are disabled by default. If you enable the macros, a new tab called Oracle Business Rules, appears. This tab enables you to add or delete rules, merge or split cells, and add or remove values from value sets. You can also disable or enable highlighting, use a simple or advanced mode and hide or show the Readme sheet.

You can edit with the macros disabled, though you will not be able to:

- Choose values from drop lists for restricted cells.
- Edit free form cells.
- Copy and paste a range of cells to add a rule or Value Set.
- Delete a range of cells to delete a rule or Value Set.
- Split or merge cells.
- Create Value Sets automatically.
- Validate the structure of Decision Tables or Value Sets.

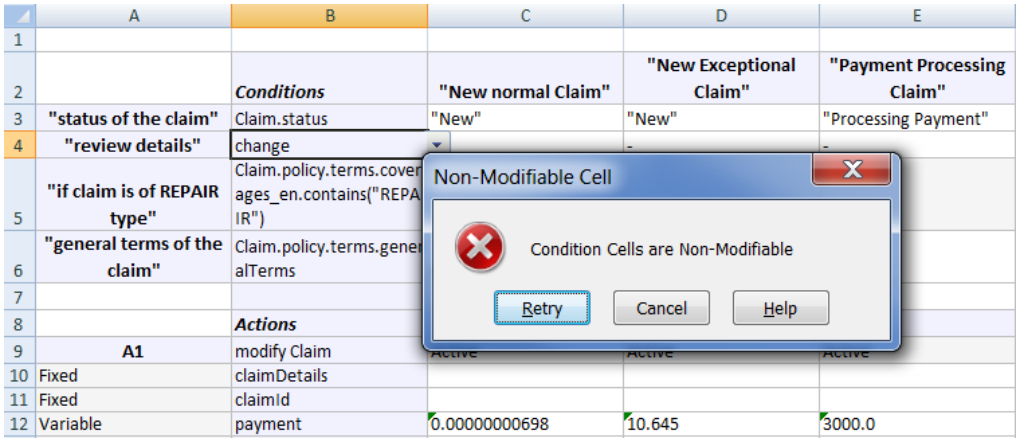
Using the predefined macros, you can:

- Add and delete rules.
- Split or merge cells.
- Add or delete Value Sets.

- Editable cells include:
  - Description for Rules, Conditions, Actions.
  - Condition and Action nodes.
  - Action state.
  - Parameterized options for Action parameters.
- Non-editable cells include:
  - Condition expressions.
  - Action expressions.
  - Action parameters.

  If you try to edit these cells, you will get an error message, as shown in Figure 12-51.

Figure 12-51 Non Modifiable Cell



	A	B	C	D	E
1					
2		***Conditions***	**"New normal Claim"**	**"New Exceptional Claim"**	**"Payment Processing Claim"**
3	**"status of the claim"**	Claim.status	"New"	"New"	"Processing Payment"
4	**"review details"**	change			
5	**"if claim is of REPAIR type"**	Claim.policy.terms.coverages_en.contains("REPAIR")			
6	**"general terms of the claim"**	Claim.policy.terms.generalTerms			
7					
8		***Actions***			
9	**A1**	modify Claim			
10	Fixed	claimDetails			
11	Fixed	claimId			
12	Variable	payment	0.00000000698	10.645	3000.0

Understanding What is Exported

In the SDK, there are shared Value Sets that can be associated with multiple conditions across Decision Tables. However, in Excel there are no shared Value Sets--each condition has its own Value Set--so you can only export a Value Set if it is modifiable in Excel. The Value Sets that are non-modifiable include:

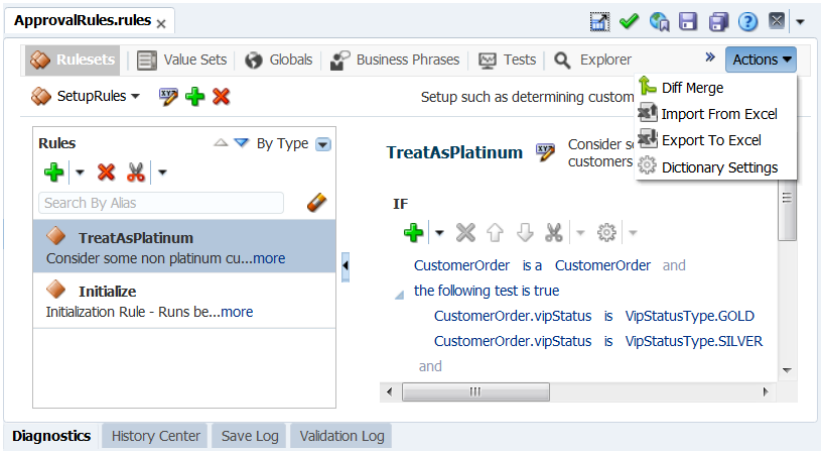
- Linked Dictionary Value Sets.
- Enums.
- Internal Value Sets, for example, boolean Value Sets.

In the worksheet, you can only select values from the drop down for the conditions associated with non-modifiable Value Sets. A highlighting mechanism informs you which conditions are associated with non-modifiable Value Sets.

How to Export Decision Tables

The export functionality is invoked by using the **Export to Excel** button in the toolbar options, as shown in Figure 12-52.

Figure 12-52 Actions Toolbar



To export to Excel:

1. In SOA Composer, in a session, click **Actions, Export to Excel**.
2. In the **Export Decision Tables** dialog box, select the **Format** and browse to the folder where you want to save the worksheet.
3. Select the Decision Table to export and click **OK.**
4. Check the **Read Only Value Set** check box to make all of the value sets read-only in Excel. There will not be any Value Sets sheet in the Excel workbook. All conditions will have drop down menus from which values can be selected but no values can be added or removed.
5. Click **Export.** You can now open the worksheet and edit the Decision Table.

How to Import Decision Tables to the Dictionary

You can only import Excel spreadsheets that have been previously exported.

To import edited Decision Tables back to the Dictionary:

1. In Rules Designer, click **Actions, Import from Excel**.
2. In the **Import Decision Tables** dialog box, click **Browse** to browse to the folder where you saved the worksheet.
3. The **Diff-Merge** check box is selected by default. For more information about using the Diff-Merge, see Comparing and Merging Oracle Business Rules Dictionaries. Select **Create New** or **Overwrite** depending on your requirements.
4. Click **Import.** The decision table is imported into Rules Designer, where you can accept or reject changes. Each changed artifact is flagged with a change icon.

Note:

Merges should be done with caution. See Comparing and Merging Oracle Business Rules Dictionaries

---

How to Edit Decision Tables in Excel

In Excel, enable the macros to view the Oracle Business Rules tab, which provides you with options to author rules, edit Value Sets, and set preferences.

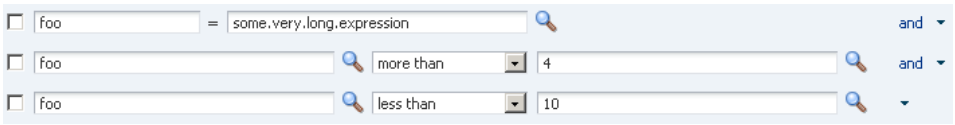
For more information, see Editing Decision Tables in Microsoft Excel.

What You Need to Know About Rule Test Variables

Oracle SOA Composer enables you to define test variables that provide a way to shorten lengthy expressions that occur in rule and decision table conditions and actions. The variable and its value can be represented as an inline business term definition. The test variables are also called inline aliases.

So, instead of writing:

You can write:



In subsequent test conditions, you can use `foo` as part of your expressions. The expression can be anything from a simple to a complex expression.

To define a variable, in the IF section of a rule, you need to click the down arrow adjacent to **Add Test**, and select **variable** from the list.

Apart from variables, you can also define other test form types, such as simple test, nested tests (`(...)`), and not nested tests (`not (...)`).

Comparing and Merging Oracle Business Rules Dictionaries

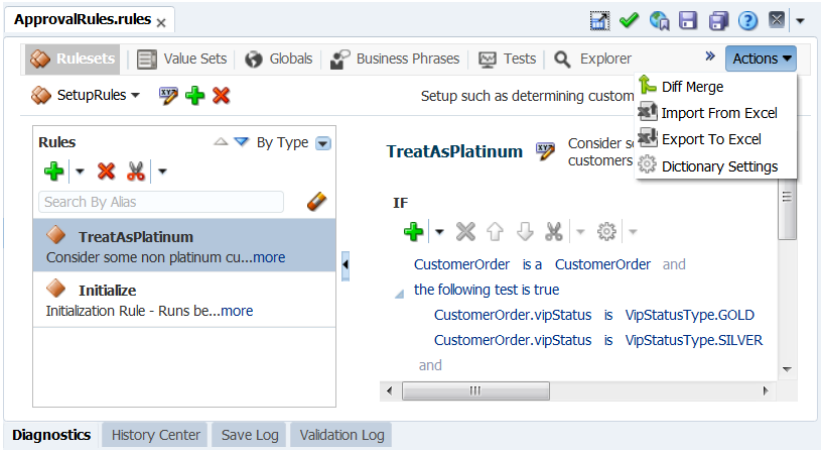
The Diff Merge feature enables you to review any differences in the latest revision of a dictionary against a previous revision and be able to save or roll back any changes since then. At runtime, in SOA Composer, you can use the Diff Merge feature to compare the File-System, Published Version, or the Saved Version to the dictionary that you have open.

SOA Composer only supports the compare of the edited version with one prior saved version and the ability to select items that have changed since the saved version and to revert them back to their saved values. The differences are viewed from the perspective of the latest revision.

The Merge feature enables you to review any differences between the two versions and be able to resolve or merge the differences among them. The differences are viewed from the perspective of the changed versions.

The **Diff Dictionary** option is available in the Rules Designer toolbar, as shown in Figure 12-53.

Figure 12-53 Actions Drop Down List



Warning:

Before you decide to run this feature, you must be ready to resolve all changes because the dictionary becomes read-only when in diff or merge mode.

Merging dictionaries should be done with care.

How to see Differences Between Dictionaries

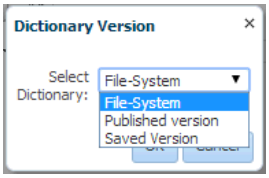
When you want to compare dictionaries, you have the newer dictionary opened and then use the Diff Merge to select the dictionary to compare with. Anything missing from the newer dictionary is flagged as a deletion from the newer version.

To see differences between dictionaries:

1. In SOA Composer, with the newer dictionary open, click **Actions, Diff Merge.**

   The **Dictionary version** dialog appears, as shown in Figure 12-54.

   Figure 12-54 Dictionary Version dialog

2. In the **Select Dictionary** field, select one of the Dictionary version to compare.

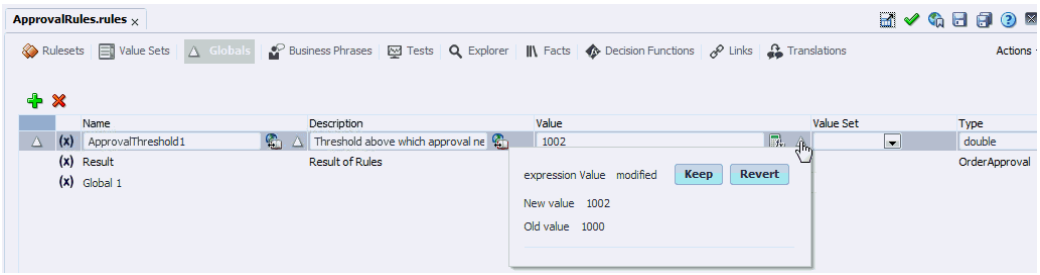
   - File-System - File-system version give users an option to compare the rules file available on the local file system against the rules file of the composite deployed on the server.
   - Published Version - Published version is the composite versions which changes when versions change.
   - Saved Version - Saved version is the composite version which changes when versions change.

3. Click **OK** to open the dictionary that you want to compare with.

   All differences between the two dictionaries are flagged with change icons.

   The change icons are shown for all tabs and for the specific artifacts within each tab. An example is shown in Figure 12-55.

Figure 12-55 Merging or Reverting Changes

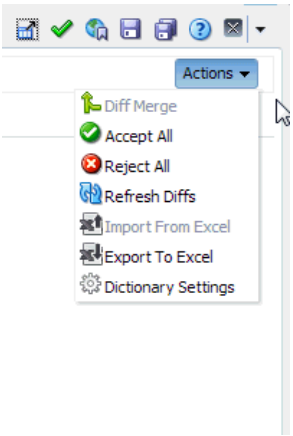


4. Click **Keep** to retain the changes or select **Revert** to discard the changes made.

   You may click each tab and decide to keep or revert the changes.

5. Alternatively, you can choose to **Accept All** or **Reject All** in the **Actions** drop down list to accept or reject all the changes on one click.

Figure 12-56 Accept All or Reject All from the Actions Drop Down



The Diff Merge feature is more fully functional in JDeveloper Rules Designer. For more information, see How to Compare or Merge Two or More Dictionaries.

Localizing Names of Resources in Oracle Business Rules

Oracle BPM allows you to localize the names of some rules components.

Providing a translated version of these aliases enables users to view these aliases based on the local setting of their browser when using the following applications:

- Oracle SOA Composer
- Oracle Business Process Composer
- Oracle Process Workspace

Note:

Locale dictionaries are stored as resource bundles. You must create the resource bundle using Oracle JDeveloper. They must be deployed as part of the SOA composite application.

Resource bundles cannot be created using Oracle SOA Composer. However, you can use Oracle SOA Composer to edit the localized strings within a resource bundle.

Oracle SOA Composer enables you to localize the aliases of the following rules components.

- Values
- Value Sets
- Decision Functions
- Decision Function Facts
- Globals
- Links
- Rulesets
- Rules
- Patterns

How to Localize the Alias of a Oracle Business Rules Component

Using Oracle SOA Composer, in a session, you can add translated versions of the aliases and their descriptions used to identify rules components.

To localize the alias of a rules component:

1. In Oracle SOA Composer, select the **Translations** tab. The Translations tab displays a table with multiple columns. By default, there are two columns one displaying the untranslated identifier of the rules component. The other displays the English locale.

   If you have defined other locales in your application, these also appear as columns in this table. See Localizing Oracle Business Rule Resources. for more information.

2. In the column of the locale you want to edit, double-click in cell corresponding to the alias you want to translate.

3. Enter the localized text for the alias.

4. Repeat steps 2 and 3 to localize all the aliases required for the locale.

Note:

Offline editing of locale files is not supported. When a locale is added, the xml file generated does not contain all the keys by default. They are added when a value is added.

You can also localize from the editor. To localize from the editor click the Translations button.

The Translations Editor enables you to appears enter the **Alias** for the rule components and click **OK**.

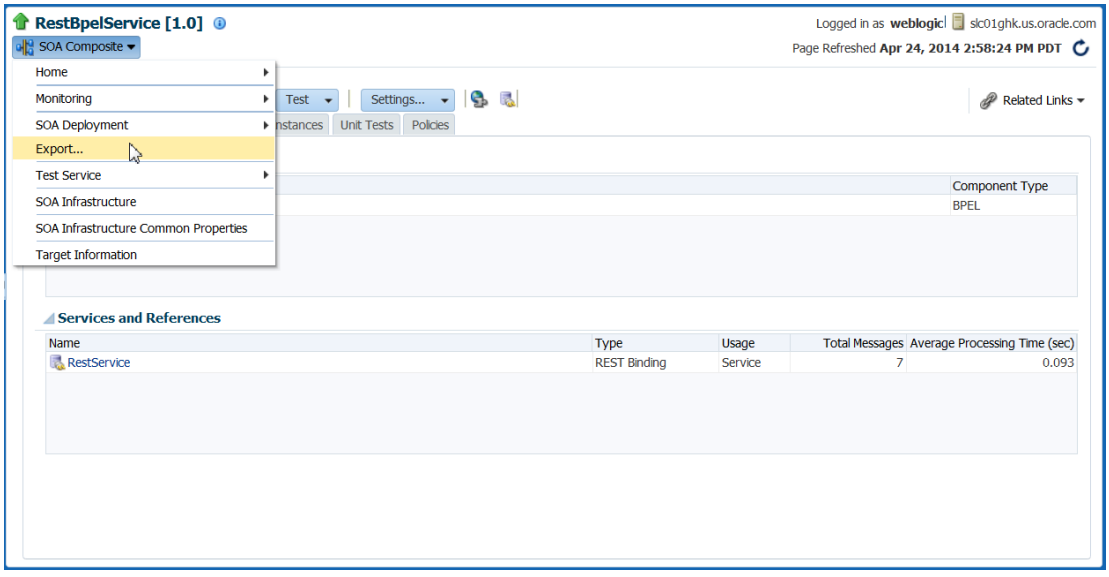
Synchronizing Rules Dictionary in Oracle JDeveloper With Runtime Dictionary Updates

Oracle SOA Composer enables you to update rules dictionaries at runtime. However, the modifications made to the dictionaries through Oracle SOA Composer are not automatically reflected in Oracle JDeveloper. To synchronize the dictionary updates made in Oracle SOA Composer with the dictionaries available in Oracle JDeveloper, you must select the **Export** option in Oracle Enterprise Manager Fusion Middleware Control Console. This utility allows you to export the SOA composite application along with the dictionary.

To select the Export option in Fusion Middleware Control Console:

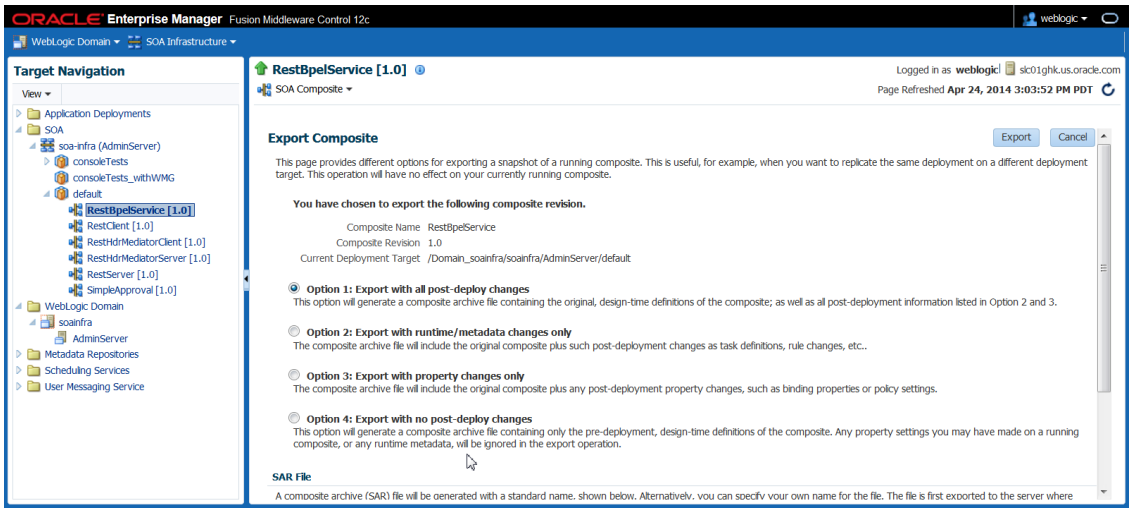
1. In Fusion Middleware Control Console, select the composite that contains the dictionary to be exported.
2. Click **SOA Composite** drop-down list on the right panel and select **Export** as shown in Figure 12-57.

Figure 12-57 Selecting the Export Utility



3. Select **Option 1: Export with all post-deploy changes** from the Export Composite page and click **Export** as shown in Figure 12-58.

Figure 12-58 Exporting All Postdeployment Changes



Validating and Diagnosing an Oracle Business Rules Dictionary

In Oracle SOA Composer, in a session, you use the bottom tabs to check diagnostics and validate a dictionary for errors. The diagnostics tab is populated after you publish a session.

During the publish of the session, if another user has made any changes to the same artifact (like a dictionary) it will be listed in this section. You then have three options to handle any conflicts.

Understanding the Validation Log Tab

The Validation Log tab lists all the dictionary-level validation errors.

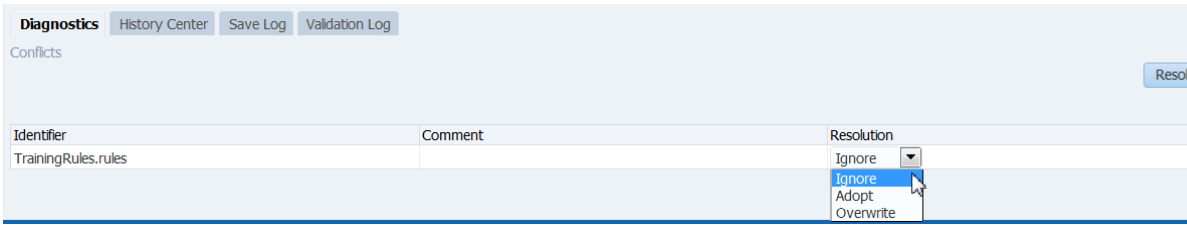
The Validation Log does not get updated automatically. The validation is only run if you click **Validate** and when you save any changes.

For example, when a new rule is added with errors, the Validation Log tab is not updated automatically. Click the **Validate** button on the toolbar to update the Validation Log with the new error entries.

Understanding the Diagnostics Tab

Use the Diagnostics tab see if other users are modifying the same artifacts and resolve those conflicts, as shown in Figure 12-59.

Figure 12-59 Diagnostics Tab



To resolve changes made by you or other users:

1. For each item in the table, use the Resolution drop down to **Ignore, Adopt,** or **Overwrite** changes.

2. **Adopt** will try to merge changes.
3. **Ignore** means your changes will be discarded.
4. **Overwrite** means that your changes will be made, other users changes will be discarded.
5. Click **Publish** when done.

Understanding the History Center Tab

The **History Center** tab displays any pending changes that you or other users have made in a currently active session. You can discard changes from here, for example, if you want to quickly undo all changes you have made all at once.

Understanding the Save Log Tab

The Save Log tab is updated whenever you save. It adds an entry to the save log if it succeeds. It it fails it will show the error message here.

Working with Tasks

Using Oracle SOA Composer, you can view and edit tasks that may be or may not be associated to Approval Management Extensions (AMX) rules. AMX enables you to define complex task routing slips for human workflow by taking into account business documents and associated rules to determine the approval hierarchy for a work item.

Additionally, AMX lets you define multi-stage approvals with associated list builders based on supervisor or position hierarchies. At design time, you can define the approval task in the Human Task Editor of Oracle JDeveloper, and associate the task with a BPEL process. For more information about approval management and tasks, see Using Approval Management in *Oracle Fusion Middleware Modeling and Implementation Guide for Oracle Business Process Management*.

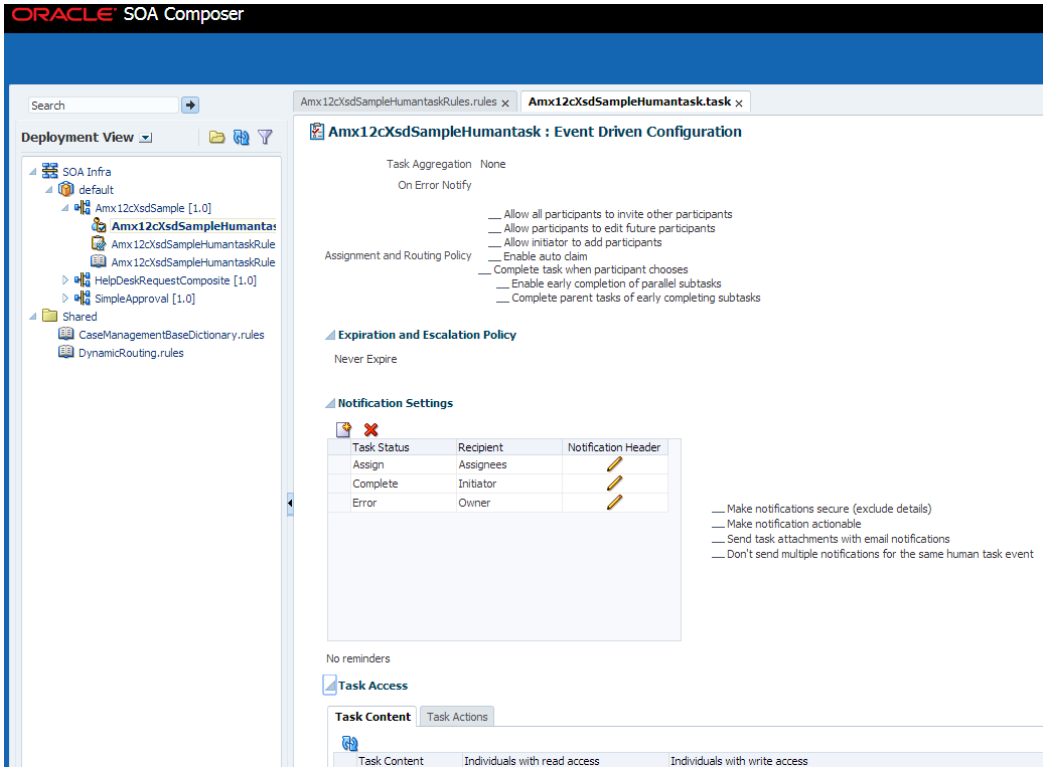
In Oracle SOA Composer, the Task Editor is embedded as a task flow so that you can view and perform all the task metadata lifecycle operations.

How to View Task Metadata

To view task metadata:

1. In Oracle SOA Composer, to open a task or an AMX rules metadata artifact, from the **Types View**, expand the **Human Tasks** folder, and click an artifact to open it in a new tab, as shown in Figure 12-60.

Figure 12-60 Opening a Task



2. If you want to make changes, click **Edit Session.** When you are ready to apply the changes to the runtime version, click **Publish.**

   For more information about how to use the session buttons, see Creating and Publishing Sessions

You can differentiate between traditional rules and AMX rules depending on the naming convention.

For example, a composite may have the following artifacts:

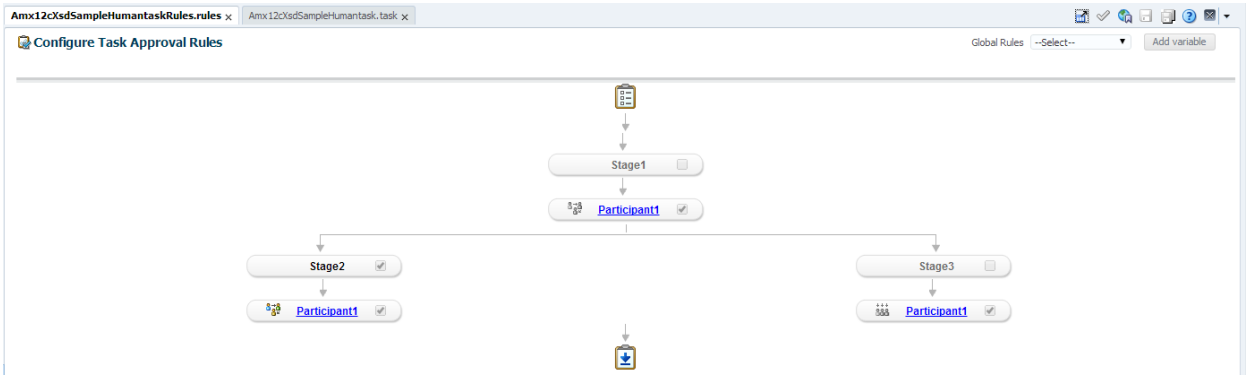
- <AMX task name>.tsk
- <AMX rule name>Rules.rules

How to Configure a Task or an AMX Rule Metadata

Task Configuration enables business users and administrators to review the rules that were configured automatically by the workflow designer. These predefined rules can be changed for a specific customer based on the customer's applicable corporate policies.

In Oracle SOA Composer, Task Configuration enables you to edit the event-driven (only tasks) and data-driven rules (tasks with an associated AMX rules) associated with an approval flow at runtime.

Figure 12-61 Configuring Tasks

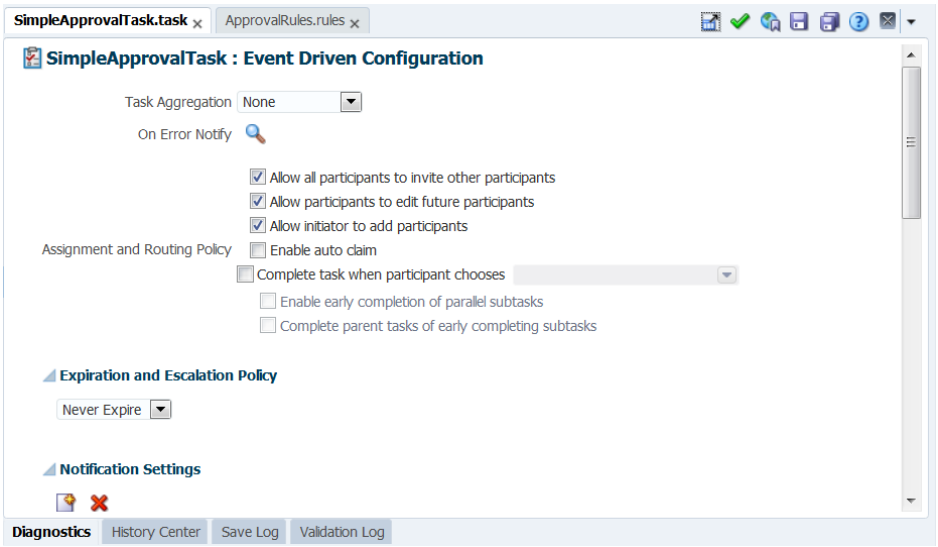


Configuring Event-Driven Settings

To configure event-driven settings:

1. Log on to Oracle SOA Composer and open the required task.
2. Click **Edit Session** on the Oracle SOA Composer menu bar to open the selected task for editing as shown in Figure 12-62.

Figure 12-62 Opening a Task for Editing



3. Make the relevant edits and click **Save Changes in Current Tab** When you are ready to apply the changes to the runtime version, click **Publish.**

You can configure the following options and settings:

- Task aggregation
- Error notification
- Assignment and routing policy
- Expiration and escalation policy
- Notification settings

- Task access settings

Setting Approval Aggregation Requirements

Task aggregation requirements can be any of the following:

- None
- Once per task
- Once per stage

Notifying Errors

You can specify the user and group names that need to be notified in case of an error in the task. You need to click the On Error Notify search button to display the Configure Error Assignees dialog box where you can specify the user or group names.

Setting Assignment and Routing Policy

You can set the assignment and routing policy by using the options available in Oracle SOA Composer. Click to select the available options for setting assignment and routing policy.

For more information about the assignment and routing options available in event-driven configuration, see Routing Policy Method in *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

Setting Expiration and Escalation Policy

You can set the expiration and escalation policy for the task by using the available items in the **Expiration and Escalation Policy** list. The available list items are:

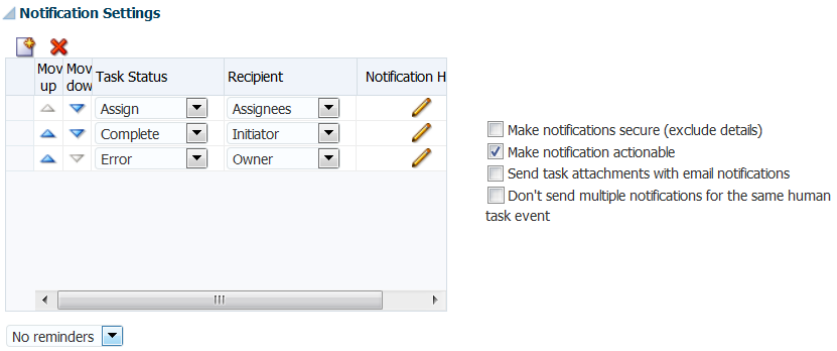
- Never Expire
- Expire After
- Escalate After
- Renew After

Configuring Notification Settings

You can configure notification settings for a task by using the options available in the Notification Settings section of Oracle SOA Composer.

#unique_500/unique_500_Connect_42_BABFDIHH displays the different options available to configure notification settings for a task.

Figure 12-63 Specifying Notification Settings



Configuring Task Access Settings

You can set access-rule settings to control the actions a user can perform. You can also specify content and action permissions based on the logical role of a user, such as creator (initiator), owner, assignee, and reviewers.

In Oracle SOA Composer, you can set access settings by using the options available under Task Access area and tabs, as shown in #unique_502/unique_502_Connect_42_BABFCAHF and #unique_502/unique_502_Connect_42_BABIACHC.

Figure 12-64 Specifying Task Access Settings

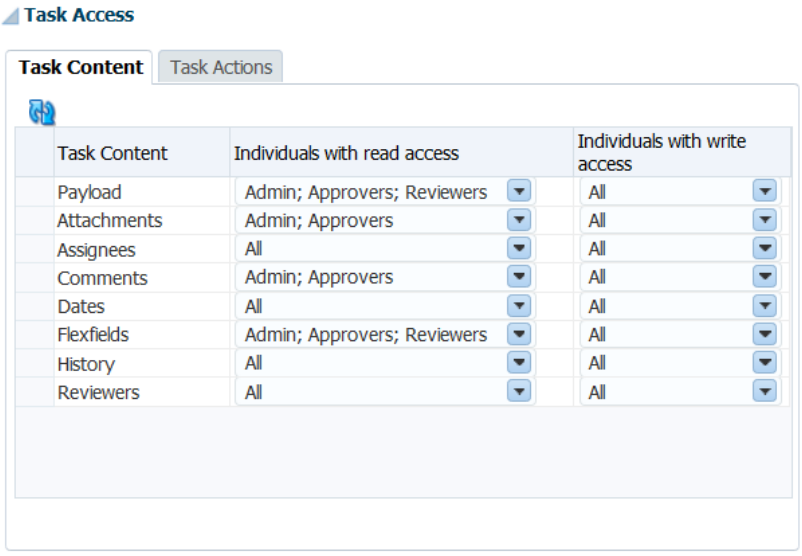
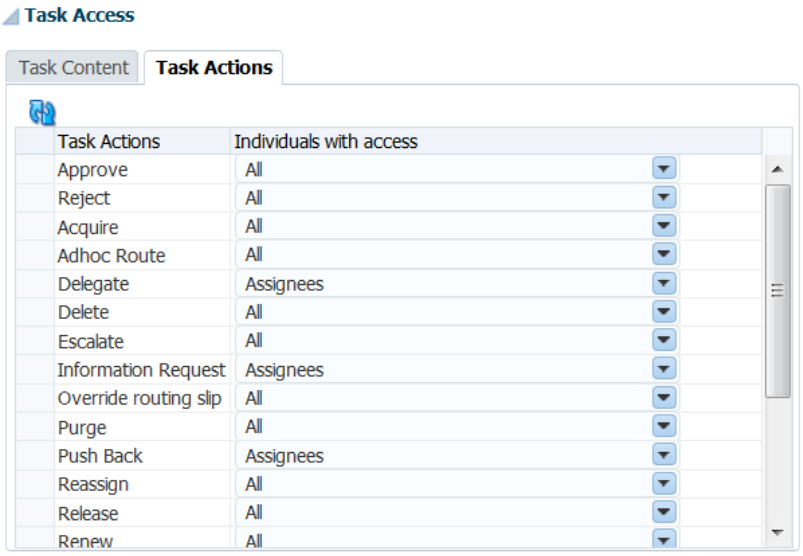


Figure 12-65 Specifying Task Actions Settings



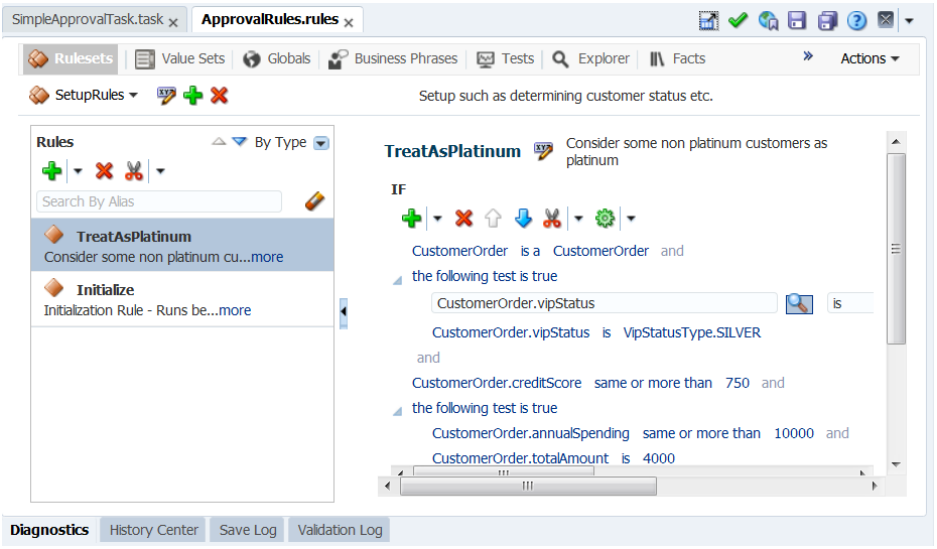
For more information on configuring task access, see How to Define Security Access Rules in *Oracle Fusion Middleware Modeling and Implementation Guide for Oracle Business Process Management*.

Configuring Data-Driven Settings (Rule or Condition)

To configure data-driven settings:

1. Log on to Oracle SOA Composer and open the required task.
2. Click **Edit Session** on the Oracle SOA Composer menu bar to open the selected AMX rule-associated task for editing as shown in Figure 12-66.

Figure 12-66 Editing a Rule-Associated Task



3. Make the relevant edits and click **Save Changes in Current Tab**. If you are ready to apply the changes to the runtime version, click **Publish.**

You can perform the following actions:

- Adding, updating, and deleting a rule
- Changing rule assertions (which depend on the type of list builder for which the rule has been configured)
- Adding a variable

For more information about editing data-driven settings, see How to Edit Data-Driven Settings in *Oracle Fusion Middleware User's Guide for Oracle Business Process Management*.

A

Oracle Business Rules Files and Limitations

This appendix lists known naming constraints for Rules Designer files and names, and certain Rules SDK limitations.

This appendix includes the following sections:

Rules Designer Naming Conventions

This section covers Rules Designer naming conventions.

Some of the naming conventions are for ruleset, dictionary, alias, and, XML schema target packaging naming.

Ruleset Naming

Rules Designer enforces a limitation for ruleset names; a ruleset name must start with a letter and contain only letters, numbers, or the following characters: `"."`, `"-"`, `"_"`,`""`, `":"`, `"/"`, and single spaces. Letters include the characters (`a` to `z` and `A` to `Z`) and numbers (`0` to `9`).

Dictionary Naming

Rules Designer dictionary names can contain only the following characters, upper and lowercase letters (`a` to `z` and `A` to `Z`), numbers (`0` to `9`), and the underscore (_). Special characters are not valid in a dictionary name.

Rules Designer dictionary names are case preserving but case-insensitive. For example, the dictionary names `Dictionary` and `DICT` are both valid. If you create a dictionary named `Test`, then you can create another dictionary named `TEST` only if you first delete the dictionary named `Test`.

Alias Naming

Rules Designer alias names must begin with a letter and contain only letters, numbers, `"."`, `"-"`, `"_"`,`""`, `":"`, `"/"`, and single spaces.

XML Schema Target Package Naming

The **Target Package Name** that you specify for an XMLFact on the XML Schema Selector page is limited to ASCII characters, digits, and the underscore character.

B

Oracle Business Rules Built-in Classes and Functions

This appendix discusses the extensive library of Oracle Business Rules (OBR) built-in classes, methods, and functions that help reasoning about data containing text strings, lists, numbers, dates, times, and so on.

In the following sections, there are multiple tables whose each row has a Kind column that is either **Cl**, **Co**, **M**, **sM**, **P**, or **sP** (Class, Constructor, Method, static Method, Property, or static Property (Java static final field) respectively). The first row in each table specifies the class. When the Java Name is the same as the OBR Name (the rule SDK terms it the Alias), a '-' is displayed. The Signature column provides type information for methods, functions, and properties. The signature of a property is actually the type, for example `BigDecimal`. The signature of a method or function is of the form `return(arg1,arg2,...)`, where `return` is the return type and `arg1,arg2,...` are the argument types.

This appendix includes the following sections:

String Classes

This section covers the String-related classes provided by Oracle Business Rules.

Table Table B-1 lists the `String` class.

Table B-1 Strings-related Classes provided by Oracle Business Rules

OBR Name	Kind	Signature	Java Name	Description	Reference
String	**Cl**	**-**	**java.lang.String**	**Java immutable character strings. Beware, Java uses 0-based indexing for characters in strings, and XML uses 1-based indexing**	`http://java.sun.com/javase/6/docs/api/java/lang/String.html`

Table B-1 (Cont.) Strings-related Classes provided by Oracle Business Rules

OBR Name	Kind	Signature	Java Name	Description	Reference
charAt	S	char(int)	-	Returns the char value at 0-based index arg1. "Oracle".charAt(2)=='a'.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#charAt%28int%29
compareTo	M	int(String)	-	Returns the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument. "a".compareTo("b")<0.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#compareTo%28java.lang.String%29
contains	M	boolean(Strin g)	-	Tests whether this string contains arg1. "Oracle".contains("r ac")==true.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#contains%28java.lang.CharSequence%29
endsWith	M	boolean(Strin g)	-	Tests whether this string ends with arg1. "Oracle".endsWith("le")==true.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#endsWith%28java.lang.String%29
equalsIgnore Case	M	boolean(Strin g)	-	Tests whether this string equals arg1, ignoring case consideration. "Oracle".equalsIgn oreCase("oRaClE")==true.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#equalsIgnoreCase%28java.lang.String%29

Table B-1 (Cont.) Strings-related Classes provided by Oracle Business Rules

OBR Name	Kind	Signature	Java Name	Description	Reference
indexOf	M	int(String,int)	-	Returns the 0-based index of the start of arg1 within this String, but not before the 0-based index arg2. "banana".indexOf("an",2)==3.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#indexOf%28java.lang.String,%20int%29
lastIndexOf	M	int(String,int)	-	Returns the 0-based index within this string of the last occurrence of arg1, searching backward starting at the index arg2. "banana".lastIndexOf("an","banana".length())==3.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#lastIndexOf%28java.lang.String,%20int%29
length	M	int	-	Returns the length of this string. "Oracle".length()==6.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#length%28%29
matches	M	boolean(String)	-	Tests if this string matches the given regular expression. "banana".matches("^b.*a$")==true.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#matches%28java.lang.String%29
replaceAll	M	String(String,String)	-	Replaces each substring of this string that matches arg1 (a regular expression) with arg2. "banana".replaceAll(".a","xo")=="xoxoxo".	http://java.sun.com/javase/6/docs/api/java/lang/String.html#replaceAll%28java.lang.String,%20java.lang.String%29

Table B-1 (Cont.) Strings-related Classes provided by Oracle Business Rules

OBR Name	Kind	Signature	Java Name	Description	Reference
replaceFirst	M	String(String,String)	-	Replaces first substring of this string that matches arg1 (a regular expression) with arg2. "banana".replaceFirst(".a","xo")=="xonana".	http://java.sun.com/javase/6/docs/api/java/lang/String.html#replaceFirst%28java.lang.String,%20java.lang.String%29
startsWith	M	boolean(String)	-	Tests whether this string starts with arg1. "Oracle".startsWith("Or")==true.	http://java.sun.com/javase/6/docs/api/java/lang/String.html#startsWith%28java.lang.String%29
substring	M	String(int,int)	-	Returns the substring of this string, starting with the 0-based index arg1, and ending before the 0-based index arg2. "Oracle".substring(1,4)=="rac".	http://java.sun.com/javase/6/docs/api/java/lang/String.html#substring%28int,%20int%29
toLowerCase	M	String()	-	Converts this string to lower case. "Oracle".toLowerCase()=="oracle".	http://java.sun.com/javase/6/docs/api/java/lang/String.html#toLowerCase%28%29
toUpperCase	M	String()	-	Converts this string to upper case. "Oracle".toUpperCase()=="ORACLE".	http://java.sun.com/javase/6/docs/api/java/lang/String.html#toUpperCase%28%29

Table B-1 (Cont.) Strings-related Classes provided by Oracle Business Rules

OBR Name	Kind	Signature	Java Name	Description	Reference
trim	M	String()	-	Removes leading and trailing whitespace. " Oracle ".trim()=="Oracle".	http://java.sun.com/javase/6/docs/api/java/lang/String.html#trim%28%29

Table B-2 lists the RL class strings methods.

Table B-2 RL class strings methods

OBR Name	Kind	Signature	Java Name	Description	Reference
RL	**Cl**	-	**oracle.rules.rl.extensions.RL**	**Supplement standard Java classes with W3C RIF functionality.**	http://www.w3.org/TR/rif-dtb/
string.join	sM	String(String ...)	stringJoin	Concatenates first n-1 args using the last arg as a separator. RL.string.join("a","b","c","#")=="a#b#c".	http://www.w3.org/TR/rif-dtb/#func:string-join
string.substring	sM	String(String,int,int)	substring	Returns the substring of arg1, beginning at the 1-based index arg2, and continuing for arg3 characters. RL.string.substring("Oracle",2,3)=="rac".	http://www.w3.org/TR/rif-dtb/#func:substring
string.suffix	sM	String(String,int)	substring	Returns the suffix of arg1, beginning at the 1-based index arg2. RL.string.suffix("Oracle",5)=="le".	http://www.w3.org/TR/rif-dtb/#func:substring
string.substring before	sM	String(String,String)	substringBefore	Returns the substring of arg1 that occurs before arg2. RL.string.substring before("Oracle","ac")=="Or".	http://www.w3.org/TR/rif-dtb/#func:substring-before

Table B-2 (Cont.) RL class strings methods

OBR Name	Kind	Signature	Java Name	Description	Reference
string.substring after	sM	String(String,String)	substringAfter	Returns the substring of arg1 that occurs after arg2. RL.string.substring after("Oracle","ac")=="le".	http://www.w3.org/TR/rif-dtb/#func:substring-after
string.iri.encode path	sM	String(String)	encodeForURI	Encodes characters not permitted in an URI path. RL.string.iri encode path("Oracle Business Rules")=="Oracle%20Business%20Rules".	http://www.w3.org/TR/rif-dtb/#func:encode-for-uri
string.iri.to uri	sM	String(String)	iriToUri	Encodes some characters not permitted in a URI. RL.string.iri to uri("http://www.example.com/~bébé")=="http://www.example.com/~b%C3%A9b%C3%A9"	http://www.w3.org/TR/rif-dtb/#func:iri-to-uri
string.iri.to ascii	sM	String(String)	escapeHtmlUri	Encodes non-ascii characters. RL.string.iri to ascii("javascript:if (navigator.browserLanguage == 'fr') window.open('http://www.example.com/~bébé');")=="javascript:if (navigator.browserLanguage == 'fr') window.open('http://www.example.com/~b%C3%A9b%C3%A9');"	http://www.w3.org/TR/rif-dtb/#func:escape-html-uri

Table B-2 (Cont.) RL class strings methods

OBR Name	Kind	Signature	Java Name	Description	Reference
string.is normalized	sM	boolean(String)	isNormalizedString	A normalized string does not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters. RL.string.is normalized(" Business Rules ")==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
string.is token	sM	boolean(String)	isToken	A token is a normalized string with no leading or trailing spaces, and no double spaces. RL.string.is token("Business Rules")==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
string.is language	sM	boolean(String)	isLanguage	A language identifier. RL.string.is language("en")==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
string.is Name	sM	boolean(String)	isName	A name is a token with no spaces (and some other constraints on its characters). RL.string.is Name("xs:Name")==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
string.is NCName	sM	boolean(String)	isNCName	A non-colonized name. RL.string.is NCName("xs:NCName")==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
string.is NMTOKEN	sM	boolean(String)	isNMTOKEN	An NMTOKEN is a Name with no restriction on the initial character. RL.string.is NMTOKEN("-Oracle")==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes

Table B-2 (Cont.) RL class strings methods

OBR Name	Kind	Signature	Java Name	Description	Reference
string.compare	sM	int(String,String)	compare	Returns -1, 0, or 1 if arg1<arg2, arg1==arg2, or arg1>arg2, respectively. RL.string.compare("foo","bar")==1.	http://www.w3.org/TR/rif-dtb/#func:compare_.28adapted_from_fn:compare.29

List Classes

This section covers the List classes provided by Oracle Business Rules.

Table Table B-3 lists the `List` class.

Table B-3 Table lists the List class

OBR Name	Kind	Signature	Java Name	Description	Reference
List	**Cl**	-	**java.util.List**	**Represents mutable and immutable lists. Lists use 0-based indexes. Attempts to modify an immutable list may result in UnsupportedOperationExceptions.**	http://java.sun.com/javase/6/docs/api/java/util/List.html
append	M	void(Object)	add	Appends arg1 to this list. Modifies this list.	http://java.sun.com/javase/6/docs/api/java/util/List.html#add(E)
add	M	void(int,Object)	-	Inserts arg2 into this list at position arg1. Modifies this list.	http://java.sun.com/javase/6/docs/api/java/util/List.html#add(int,%20E)
appendAll	M	void(java.util.Collection)	addAll	Appends the contents of arg1 to this list. Modifies this list.	http://java.sun.com/javase/6/docs/api/java/util/List.html#addAll(java.util.Collection)
addAll	M	void(int,java.util.Collection)	-	Inserts the contents of arg2 into this list at position arg1. Modifies this list.	http://java.sun.com/javase/6/docs/api/java/util/List.html#addAll(int,%20java.util.Collection)
clear	M	void()	-	Removes the contents of this list. Modifies this list.	http://java.sun.com/javase/6/docs/api/java/util/List.html#clear()

Table B-3 (Cont.) Table lists the List class

OBR Name	Kind	Signature	Java Name	Description	Reference
contains	M	boolean(Object)	-	Tests whether this list contains arg1. RL.list.create(1,2,3).contains(2)==true.	http://java.sun.com/javase/6/docs/api/java/util/List.html#contains(java.lang.Object)
containsAll	M	boolean(java.util.Collection)	-	Tests whether this list contains every element in arg1. RL.list.create(1,2,3).containsAll(RL.list.create(3,2,1))==true.	http://java.sun.com/javase/6/docs/api/java/util/List.html#containsAll(java.util.Collection)
get	M	Object(int)	-	Get the element at position arg1. RL.list.create(1,2,3).get(1)==2.	http://java.sun.com/javase/6/docs/api/java/util/List.html#get(int)
indexOf	M	int(Object)	-	Returns first index of arg1 in this list. RL.list.create(1,2,3).indexOf(2)==1.	http://java.sun.com/javase/6/docs/api/java/util/List.html#indexOf(java.lang.Object)
remove	M	boolean(Object)	-	Removes first occurrence of arg1 from this list. Returns whether this list was modified.	http://java.sun.com/javase/6/docs/api/java/util/List.html#remove(java.lang.Object)
remove by index	M	Object(int)	remove	Removes and return the element at position arg1. Modifies this list.	http://java.sun.com/javase/6/docs/api/java/util/List.html#remove(int)
removeAll	M	boolean(java.util.Collection)	-	Removes all elements from this list that are contained in arg1. Returns whether this list was modified.	http://java.sun.com/javase/6/docs/api/java/util/List.html#removeAll(java.util.Collection)
retainAll	M	boolean(java.util.Collection)	-	Removes all elements from this list that are *not* contained in arg1. Returns whether this list was modified.	http://java.sun.com/javase/6/docs/api/java/util/List.html#retainAll(java.util.Collection)
set	M	Object(int, Object)	-	Replaces the item in this list at position arg1 with arg2. Returns the replaced item. Modifies this list.	http://java.sun.com/javase/6/docs/api/java/util/List.html#set(int,%20E)

Table B-3 (Cont.) Table lists the List class

OBR Name	Kind	Signature	Java Name	Description	Reference
size	M	int()	-	Returns the size of this list. RL.list.create(1,2,3).size()==3.	http://java.sun.com/javase/6/docs/api/java/util/List.html#size()
subList	M	List(int,int)	-	Returns a view of the portion of this list between arg1, inclusive, and arg2, exclusive. RL.list.create(1,2,3,4).subList(1,3)==RL.list.create(2,3).	http://java.sun.com/javase/6/docs/api/java/util/List.html#subList(int,%20int)

Table B-4 lists the RL class list methods.

Table B-4 Table lists the RL class list methods

OBR Name	Kind	Signature	Java Name	Description	Reference
RL	**Cl**	-	**oracle.rules.rl.extensions.RL**	-	-
list.append	sM	List(List,Object...)	append	Returns a new immutable list containing the contents of arg1, followed by arg2, arg3, ... RL.list.append(RL.list.create(1),2,3)==RL.list.create(1,2,3).	http://www.w3.org/TR/rif-dtb/#func:append
list.concatenate	sM	List(List...)	concatenate	Returns a new immutable list containing the concatenation of arg1, arg2, ... RL.list.concatenate(RL.list.create(1),RL.list.create(2))==RL.list.create(1,2).	http://www.w3.org/TR/rif-dtb/#func:concatenate
list.distinctvalues	sM	List(List)	distinctValues	Returns a new immutable list like arg1 but with duplicates removed. RL.list.distinctvalues(RL.list.create(2,2))==RL.list.create(2).	http://www.w3.org/TR/rif-dtb/#func:distinct-values
list.except	sM	List(List,List)	except	Returns a new immutable list containing elements from arg1 that are not in arg2. RL.list.except(RL.list.create(1,2,3,4),RL.list.create(1,3))==RL.list.create(2,4).	http://www.w3.org/TR/rif-dtb/#func:except
list.get	sM	Object(List,int)	get	Returns the element at position arg2 in arg1. If arg2<0, return the element at arg1.size()+arg2. RL.list.get(RL.list.create(1,2,3),-1)==3.	http://www.w3.org/TR/rif-dtb/#func:get

Table B-4 (Cont.) Table lists the RL class list methods

OBR Name	Kind	Signature	Java Name	Description	Reference
list.index of	sM	List<Integer>(List,Object)	indexOf	Returns a list of indexes where the arg2 occurs in arg1. RL.list.index of(RL.list.create(1,2,3,2), 2)==RL.list.create(1,3).	http://www.w3.org/TR/rif-dtb/#func:index-of
list.insert before	sM	List(List,int, Object)	insertBefore	Returns a new immutable list containing arg1 with arg3 inserted before the item at position arg2. If arg2<0, arg3 is inserted before the element at arg1.size()+arg2. RL.list.insert before(RL.list.create(1,2,3),-1,99)==RL.list.create(1,2,99,3).	http://www.w3.org/TR/rif-dtb/#func:insert-before
list.intersect	sM	List(List,List)	intersect	Returns a new immutable list containing the intersection of arg1 and arg2. RL.list.intersect(RL.list.create(1,2,3),RL.list.create(3,1))==RL.list.create(1,3).	http://www.w3.org/TR/rif-dtb/#func:intersect
list.create	sM	List(Object...)	list	Returns a new immutable list containing the arguments.	http://www.w3.org/TR/rif-dtb/#func:make-list
list.remove	sM	List(List,int)	remove	Returns a new immutable list containing the elements of arg1, with the element at position arg2 removed. If arg2<0, the element at arg1.size()+arg2 is removed. RL.list.remove(RL.list.create(1,2,3),0)==RL.list.create(2,3).	http://www.w3.org/TR/rif-dtb/#func:remove
list.reverse	sM	List(List)	reverse	Returns a new immutable list containing the elements of arg1 in reverse order. RL.list.reverse(RL.list.create(1,2,3))==RL.list.create(3,2,1).	http://www.w3.org/TR/rif-dtb/#func:reverse
list.union	sM	List(List)	union	Returns a new immutable list containing the concatenation of the arguments with duplicates removed. RL.list.union(RL.list.create(1,2), RL.list.create(2,3))==RL.list.create(1,2,3).	http://www.w3.org/TR/rif-dtb/#func:union

Numeric Classes

Oracle Business Rules support the primitive Java numeric types `byte`, `short`, `int`, `long`, `float`, and `double`.

OBR also supports the "boxed" versions: `Short`, `Int`, `Long`, `Float`, and `Double`. Unlimited precision integers and decimals are supported, using the Java classes `BigInteger` and `BigDecimal`. OBR supports arithmetic expressions (`+`, `-`, `*`, `/`, `**`) on all numeric types. For example, if `*bd` is `BigDecimal`, then you can add one to it by simply writing `bd + 1`. You do not have to write `bd.add(BigDecimal.ONE)`.

Table Table B-5 lists the `Integer` class.

Table B-5 Table lists the Integer class

OBR Name	Kind	Signature	Java Name	Description	Reference
Integer	**Cl**	-	**java.lang.Integer**	**An integer object. Unlike the primitive "int", an Integer can be null and can be in Lists.**	http://java.sun.com/javase/6/docs/api/java/lang/Integer.html
Integer	Co	Integer(int | String)	-	Creates an Integer from an int or from its lexical representation as a String. new Integer(1)==new Integer("1").	http://java.sun.com/javase/6/docs/api/java/lang/Integer.html#Integer(int)
MIN_VALUE	sP	int	-	Smallest primitive int value. Integer.MIN_VALUE<0.	http://java.sun.com/javase/6/docs/api/java/lang/Integer.html#MIN_VALUE
MAX_VALUE	sP	int	-	Largest primitive int value. Integer.MAX_VALUE>0.	http://java.sun.com/javase/6/docs/api/java/lang/Integer.html#MAX_VALUE
intValue	M	int()	-	Converts this Integer to an int. new Integer(1).intValue()==1.	http://java.sun.com/javase/6/docs/api/java/lang/Integer.html#intValue()
toString	M	String()	-	Converts this Integer to its lexical representation. new Integer(1).toString()=="1".	http://java.sun.com/javase/6/docs/api/java/lang/Integer.html#toString()

Table B-6 lists the `Long` class.

Table B-6 Table lists the Long class

OBR Name	Kind	Signature	Java Name	Description	Reference
Long	**Cl**	-	**java.lang.Long**	**A long integer object. Unlike the primitive "long", a Long can be null and can be in Lists.**	http://java.sun.com/javase/6/docs/api/java/lang/Long.html

Table B-6 (Cont.) Table lists the Long class

OBR Name	Kind	Signature	Java Name	Description	Reference
Long	Co	Long(long | String)	-	Creates a Long from a long or from its lexical representation as a String. new Long(1)==new Long("1").	http://java.sun.com/javase/6/docs/api/java/lang/Long.html#Long(long)
MIN_VALUE	sP	long	-	Smallest primitive long value. Long.MIN_VALUE<0.	http://java.sun.com/javase/6/docs/api/java/lang/Long.html#MIN_VALUE
MAX_VALUE	sP	long	-	Largest primitive long value. Long.MAX_VALUE>0.	http://java.sun.com/javase/6/docs/api/java/lang/Long.html#MAX_VALUE
longValue	M	long()	-	Converts this Long to a long. new Long(1).longValue()==1.	http://java.sun.com/javase/6/docs/api/java/lang/Long.html#longValue()
toString	M	String()	-	Converts this Long to its lexical representation. new Long(1).toString()=="1".	http://java.sun.com/javase/6/docs/api/java/lang/Long.html#toString()

Table B-7 lists the `Short` class.

Table B-7 Table lists the Short class

OBR Name	Kind	Signature	Java Name	Description	Reference
Short	**Cl**	-	**java.lang.Short**	**A short integer object. Unlike the primitive "short", a Short can be null and can be in Lists.**	http://java.sun.com/javase/6/docs/api/java/lang/Short.html
Short	Co	Short(short | String)	-	Creates a Short from a short or from its lexical representation as a String. new Short(1)==new Short("1").	http://java.sun.com/javase/6/docs/api/java/lang/Short.html#Short(short)
MIN_VALUE	sP	short	-	Smallest primitive short value. Short.MIN_VALUE<0.	http://java.sun.com/javase/6/docs/api/java/lang/Short.html#MIN_VALUE
MAX_VALUE	sP	short	-	Largest primitive short value. Short.MAX_VALUE>0.	http://java.sun.com/javase/6/docs/api/java/lang/Short.html#MAX_VALUE

Table B-7 (Cont.) Table lists the Short class

OBR Name	Kind	Signature	Java Name	Description	Reference
shortValue	M	short()	-	Converts this Short to a short. new Short(-1).shortValue()==-1.	http://java.sun.com/javase/6/docs/api/java/lang/Short.html#shortValue()
toString	M	String()	-	Converts this Short to its lexical representation. new Short(-1).toString()=="-1".	http://java.sun.com/javase/6/docs/api/java/lang/Short.html#toString()

Table B-8 lists the `Float` class.

Table B-8 Table lists the Float class

OBR Name	Kind	Signature	Java Name	Description	Reference
Float	**Cl**	-	**java.lang.Float**	**A Float object. Unlike the primitive "float", a Float can be null and can be in Lists.**	http://java.sun.com/javase/6/docs/api/java/lang/Float.html
Float	Co	Float(float | double | String)	-	Creates a Float from a float, a double, or from its lexical representation as a String. new Float(1.1)==new Float("1.1").	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#Float(float)
infinite	P	boolean	-	The value of this Float is infinity. new Float(Float.NEGATIVE_INFINITY).infinite==true.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#isInfinite()
naN	P	boolean	-	The value of this Float is not a number. new Float(Float.NaN).naN==true.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#isNaN()()
NaN	sP	float	-	Value representing "not a number".	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#NaN
NEGATIVE_INFINITY	sP	float	-	Value representing negative infinity.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#NEGATIVE_INFINITY

Table B-8 (Cont.) Table lists the Float class

OBR Name	Kind	Signature	Java Name	Description	Reference
POSITIVE_INFINITY	sP	float	-	Value representing positive infinity.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#POSITIVE_INFINITY
floatValue	M	float()	-	Converts this Float to a float. new Float(1.1f).floatValue()==1.1f.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#floatValue()
toString	M	String()	-	Converts this Float to its lexical representation. new Float(1.1f).toString()=="1.1".	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#toString()

Table B-9 lists the `Double` class.

Table B-9 Table lists the Double class

OBR Name	Kind	Signature	Java Name	Description	Reference
Double	**Cl**	-	**java.lang.Double**	A Double object. Unlike the primitive "double", a Double can be null and can be in Lists.	http://java.sun.com/javase/6/docs/api/java/lang/Double.html
Double	Co	Double(double|String)	-	Creates a Double from a double or from its lexical representation as a String. new Double(1.1)==new Double("1.1").	http://java.sun.com/javase/6/docs/api/java/lang/Double.html#Double(double)
infinite	P	boolean	-	The value of this Double is infinity. new Float(Float.POSITIVE_INFINITY).infinite==true.	http://java.sun.com/javase/6/docs/api/java/lang/Double.html#isInfinite()
naN	P	boolean	-	The value of this Double is not a number. new Double(double.NaN).naN==true.	http://java.sun.com/javase/6/docs/api/java/lang/Double.html#isNaN()
NaN	sP	double	-	Value representing "not a number".	http://java.sun.com/javase/6/docs/api/java/lang/Double.html#NaN

Table B-9 (Cont.) Table lists the Double class

OBR Name	Kind	Signature	Java Name	Description	Reference
NEGATIVE_INFINITY	sP	double	-	Value representing negative infinity.	http://java.sun.com/javase/6/docs/api/java/lang/Double.html#NEGATIVE_INFINITY
POSITIVE_INFINITY	sP	double	-	Value representing positive infinity.	http://java.sun.com/javase/6/docs/api/java/lang/Double.html#POSITIVE_INFINITY
doubleValue	M	double()	-	Converts this Double to a double. new Double(1.1).doubleValue()==1.1.	http://java.sun.com/javase/6/docs/api/java/lang/Double.html#doubleValue()
toString	M	String()	-	Converts this Double to its lexical representation. new Double(1.1).toString()=="1.1".	http://java.sun.com/javase/6/docs/api/java/lang/Double.html#toString()

Table B-10 lists the `BigInteger` class.

Table B-10 Table lists the BigInteger class

OBR Name	Kind	Signature	Java Name	Description	Reference
BigInteger	**Cl**	-	**java.math.BigInteger**	**Immutable arbitrary-precision integers.**	http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html
BigInteger	Co	BigInteger(String)	-	Creates a BigInteger from its lexical representation as a String. new BigInteger("1")==1.	http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html#BigInteger(java.lang.String)
doubleValue	M	double()	-	Converts this BigInteger to a double. May lose precision. new BigInteger("1").doubleValue()==1.0.	http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html#doubleValue()
longValue	M	long()	-	Converts this BigInteger to a long. May lose precision. new BigInteger("1").longValue()==1L.	http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html#longValue()

Table B-10 (Cont.) Table lists the BigInteger class

OBR Name	Kind	Signature	Java Name	Description	Reference
max	M	BigInteger(BigInteger)	-	Returns the greater of this or arg1. new BigInteger("1").max(2)==2.	http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html#max(java.math.BigInteger)
min	M	BigInteger(BigInteger)	-	Returns the lesser of this or arg1. new BigInteger("1").min(2)==1.	http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html#min(java.math.BigInteger)
toString	M	String()	-	Returns the lexical representation of this BigInteger. new BigInteger("123").toString()=="123".	http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html#toString()
valueOf	sM	BigInteger(long)	-	Converts arg1 (a long) to a BigInteger. BigInteger.valueOf(123).toString()=="123".	http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html#valueOf(long)

Table B-11 lists the `BigDecimal` class.

Table B-11 Table lists the BigDecimal class

OBR Name	Kind	Signature	Java Name	Description	Reference
BigDecimal	**Cl**	-	**java.math.BigDecimal**	**Immutable, arbitrary-precision signed decimal numbers.**	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html
BigDecimal	Co	BigDecimal(long | double | String)	-	Creates a BigDecimal from a long, a double, or from its lexical representation as a String. new BigDecimal(1.1)==new BigDecimal("1.1").	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#BigDecimal(java.lang.String)
BigDecimal	Co	BigDecimal(BigInteger,int)	-	Creates a BigDecimal from BigInteger arg1 and scale arg2. new BigDecimal(new BigInteger("123"),2)==1.23.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#BigDecimal(java.math.BigInteger,%20int)

Table B-11 (Cont.) Table lists the BigDecimal class

OBR Name	Kind	Signature	Java Name	Description	Reference
doubleValue	M	double()	-	Converts this BigDecimal to a double. May lose precision. new BigDecimal("0.1").doubleValue()==0.1.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#doubleValue()
longValue	M	long()	-	Converts this BigDecimal to a long. May lose precision. new BigDecimal("0.1").longValue()==0L.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#longValue()
max	M	BigDecimal(BigDecimal)	-	Returns the greater of this BigDecimal or arg1. new BigDecimal("0.1").max(0.2)==0.2.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#max(java.math.BigDecimal)
min	M	BigDecimal(BigDecimal)	-	Returns the lesser of this BigDecimal or arg1. new BigDecimal("0.1").min(0.2)==0.1.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#min(java.math.BigDecimal)
scale	M	int()	-	Returns the scale--the number of digits to the right of the decimal point. new BigDecimal("1.00").scale()==2.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#scale()
setScale	M	BigDecimal(int)	-	Sets the scale, but don't change the value. new BigDecimal("1").setScale(2).toString()=="1.00".	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#setScale(int)
toEngineeringString	M	String()	-	Returns the literal representation of this BigDecimal using engineering notation if an exponent is needed. new BigDecimal("123E2").toEngineeringString()=="12.3E+3".	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#toEngineeringString()
toPlainString	M	String	-	Returns the literal representation of this BigDecimal without exponents. new BigDecimal("123E2").toPlainString()=="12300".	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#toPlainString()

Table B-11 (Cont.) Table lists the BigDecimal class

OBR Name	Kind	Signature	Java Name	Description	Reference
valueOf	sM	BigDecimal(long | double)	-	Converts arg1 (a long or double) to a BigDecimal. new BigDecimal(1.3)==BigDecimal.valueOf(1.3).	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#valueOf(double)
ROUND_UP	sP	int	-	Used with divide. new BigDecimal("11").divide(2, BigDecimal.ROUND_UP)==6.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#ROUND_UP
ROUND_DOWN	sP	int	-	Used with divide. new BigDecimal("11").divide(2, BigDecimal.ROUND_DOWN)==5.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#ROUND_DOWN
divide	M	BigDecimal(BigDecimal,int)	-	Returns this/arg1 with scale the same as this BigDecimal. If rounding must be performed to stay within the result scale, use the rounding mode given by arg2 (ROUND_UP or ROUND_DOWN). new BigDecimal("11").divide(2, BigDecimal.ROUND_UP)==6.	http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html#divide(java.math.BigDecimal,%20int)

Table B-12 lists the `Number` class.

Table B-12 Table lists the Number class

OBR Name	Kind	Signature	Java Name	Description	Reference
Number	**Cl**	-	-	**Base class of all numerics (except primitives).**	http://java.sun.com/javase/6/docs/api/java/lang/Number.html
doubleValue	M	double()	-	Converts this number to a double.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#doubleValue()
floatValue	M	float()	-	Converts this number to a float.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#floatValue()

Table B-12 (Cont.) Table lists the Number class

OBR Name	Kind	Signature	Java Name	Description	Reference
intValue	M	int()	-	Converts this number to a int.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#intValue()
longValue	M	long()	-	Converts this number to a long.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#longValue()
shortValue	M	short()	-	Converts this number to a short.	http://java.sun.com/javase/6/docs/api/java/lang/Float.html#shortValue()

Table B-13 lists the RL class number methods.

Table B-13 Table lists the RL class number methods

OBR Name	Kind	Signature	Java Name	Description	Reference
RL	**Cl**	-	**oracle.rules.rl.extensions.RL**	-	-
number.isbyte	sM	boolean(Number)	isByte	arg1 is integral and -128<=arg1<=127. RL.numeric.isbyte(200)==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.isshort	sM	boolean(Number)	isShort	arg1 is integral and -32768<=arg1<=32767. RL.numeric.isshort(0.1)==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.isint	sM	boolean(Number)	isInt	arg1 is integral and -2147483648<=arg1<=2147483647. RL.numeric.isint(-1000)==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.islong	sM	boolean(Number)	isLong	arg1 is integral and -9223372036854775808<=arg1<=9223372036854775807. RL.numeric.is integer(new BigInteger("100")**100)==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes

Table B-13 (Cont.) Table lists the RL class number methods

OBR Name	Kind	Signature	Java Name	Description	Reference
number.is integer	sM	boolean(Number)	isInteger	arg1 is integral. RL.numeric.is integer(new BigInteger("100")**100)==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.is decimal	sM	boolean(Number)	isDecimal	arg1 is neither Double nor Float. RL.numeric.is decimal(1.1)==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.is non-negative integer	sM	boolean(Number)	isNonNegativeInteger	arg1 is integral and arg1>=0. RL.numeric.is non-negative integer(-1)==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.is negative integer	sM	boolean(Number)	isNegativeInteger	arg1 is integral and arg1<0. RL.numeric.is negative integer(-1)==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.is non-positive integer	sM	boolean(Number)	isNonPositiveInteger	arg1 is integral and arg1<=0. RL.numeric.is non-positive integer(-1)==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.is positive integer	sM	boolean(Number)	isPositiveInteger	arg1 is integral and arg1>0. RL.numeric.is positive integer(-1)==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.is unsigned byte	sM	boolean(Number)	isUnsignedByte	arg1 is integral and 0<=arg1<=255. RL.numeric.is unsigned byte(200)==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.is unsigned short	sM	boolean(Number)	isUnsignedShort	arg1 is integral and 0<=arg1<=65535. RL.numeric.is unsigned short(0.1)==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
number.is unsigned int	sM	boolean(Number)	isUnsignedInt	arg1 is integral and 0<=arg1<=4294967295. RL.numeric.is unsigned int(-1000)==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes

Table B-13 (Cont.) Table lists the RL class number methods

OBR Name	Kind	Signature	Java Name	Description	Reference
number.isunsignedlong	sM	boolean(Number)	isUnsignedLong	arg1 is integral and 0<=arg1<=18446744073709551615.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes

Time and Duration Classes

This section lists the time and duration classes provided by Oracle Business Rules.

Table Table B-14 lists the `Calendar` class.

Table B-14 Table lists the Calendar class

OBR Name	Kind	Signature	Java Name	Description	Reference
Calendar	**Cl**	-	**java.util.Calendar**	**A Calendar represents a datetime and timezone. A calendar instance has a number of mutable int fields. The first argument to add, get, isSet, roll, and set is a field number. This class provides a number of static properties that should be used for the field numbers.**	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html
ERA	sP	int	-	Field number for the Calendar era. 1 is for *A.D.* and 0 is for *B.C.* ((Calendar)"2010-02-01").get(Calendar.ERA)==1.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#ERA
YEAR	sP	int	-	Field number for the Calendar year. ((Calendar)"2010-02-01").get(Calendar.YEAR)==2010.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#YEAR
MONTH	sP	int	-	Field number for the Calendar month. Months are 0-based. ((Calendar)"2010-02-01").get(Calendar.MONTH)==1.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#MONTH

Table B-14 (Cont.) Table lists the Calendar class

OBR Name	Kind	Signature	Java Name	Description	Reference
WEEK_OF_YEAR	sP	int	-	Field number for the Calendar week. ((Calendar)"2010-02-01").get(Calendar.WEEK_OF_YEAR)==6.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#WEEK_OF_YEAR
DAY_OF_YEAR	sP	int	-	Field number for the Calendar day of year. ((Calendar)"2010-02-01").get(Calendar.DAY_OF_YEAR)==32.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#DAY_OF_YEAR
DAY_OF_MONTH	sP	int	-	Field number for the Calendar day of month. ((Calendar)"2010-02-01").get(Calendar.DAY_OF_MONTH)==1.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#DAY_OF_MONTH
DAY_OF_WEEK	sP	int	-	Field number for the Calendar day of the week. ((Calendar)"2010-02-01").get(Calendar.DAY_OF_WEEK)==2.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#DAY_OF_WEEK
HOUR	sP	int	-	Field number for the Calendar hour, 12 hour format. ((Calendar)"2010-02-01T20:15:10").get(Calendar.HOUR)==8.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#HOUR
AM_PM	sP	int	-	Field number for the Calendar AM_PM flag. 0 is for *AM* and 1 is for *PM*. ((Calendar)"2010-02-01T20:15:10").get(Calendar.AM_PM)==1.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#AM_PM
HOUR_OF_DAY	sP	int	-	Field number for the Calendar hour, 24 hour format. ((Calendar)"20:15:10").get(Calendar.HOUR)==20.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#HOUR_OF_DAY

Table B-14 (Cont.) Table lists the Calendar class

OBR Name	Kind	Signature	Java Name	Description	Reference
MINUTE	sP	int	-	Field number for the Calendar minutes. JavaDate.from time string("20:15:10").get(Calendar.MINUTE)==15.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#MINUTE
SECOND	sP	int	-	Field number for Calendar seconds. ((Calendar)"20:15:10").get(Calendar.SECOND)==10.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#SECOND
ZONE_OFFSET	sP	int	-	Field number for timezone. Value is millsecond offset from GMT. ((Calendar)"20:15:10-05:30").get(Calendar.ZONE_OFFSET)==-(5*3600+30*60)*1000.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#ZONE_OFFSET
add	M	void(int,int)	add	Adds the amount of time specified by arg2 to the calendar field specified by arg1. Modifies this Calendar.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#add(int,%20int)
clear	M	void()	clear	Clears (unset all fields in) this Calendar. Modifies this Calendar.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#clear()
get	M	int(int)	get	Gets the value of the field specified by *field number* arg1. ((Calendar)"20:15:10").get(Calendar.SECOND)==10.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#get(int)
getInstance	sM	Calendar()	getInstance	Gets a calendar initialized to the current time in the default time zone and locale.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#getInstance()

Table B-14 (Cont.) Table lists the Calendar class

OBR Name	Kind	Signature	Java Name	Description	Reference
roll	M	void(int,int)	roll	Adds the amount of time specified by arg2 to the calendar field specified by arg1. Does not affect any other calendar field. Modifies this Calendar.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#roll(int,%20int)
set	M	void(int,int)	set	Sets the calendar field specified by arg1 to the value specified by arg2. Modifies this Calendar.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#set(int,%20int)
time	P	java.util.Date	time	Returns a Date object representing this Calendar's time value. ((Calendar)"2010-02-01").time<((Calendar)"2010-02-02").time.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#getTime()
timeInMillis	P	long	timeInMillis	Returns this Calendar's time value in milliseconds. ((Calendar)"2010-02-01").timeInMillis<((Calendar)"2010-02-02").timeInMillis.	http://java.sun.com/javase/6/docs/api/java/util/Calendar.html#getTimeInMillis()

Table B-15 lists the `JavaDate` class.

Table B-15 Table lists the JavaDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
JavaDate	**Cl**	-	oracle.rules.rl.extensions.JavaDate	Helper class for working with Calendars as immutable objects. Treating Calendars as immutable objects can help prevent errors.	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html

Table B-15 (Cont.) Table lists the JavaDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
add years to	sM	Calendar(Calendar, int)	addYearsTo	Returns a new Calendar that is arg2 years later than arg1. JavaDate.addyears to("2009-01-01", 1)=="2010-01-01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#addYearsTo_java_util_Calendar__int_
add months to	sM	Calendar(Calendar, int)	addMonthsTo	Returns a new Calendar that is arg2 months later than arg1. JavaDate.addmonths to("2009-01-01", 1)=="2009-02-01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#addMonthsTo_java_util_Calendar__int_
add weeks to	sM	Calendar(Calendar, int)	addWeeksTo	Returns a new Calendar that is 7*arg2 days later than arg1. JavaDate.addweeks to("2009-01-01", 1)=="2009-01-08".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#addWeeksTo_java_util_Calendar__int_
add days to	sM	Calendar(Calendar, int)	addDaysTo	Returns a new Calendar that is arg2 days later than arg1. JavaDate.adddays to("2009-01-01", 1)=="2009-01-02".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#addDaysTo_java_util_Calendar__int_

Table B-15 (Cont.) Table lists the JavaDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
add hours to	sM	Calendar(Calendar, int)	addHoursTo	Returns a new Calendar that is arg2 hours later than arg1. JavaDate.add hours to("01:01:01", 1)=="02:01:01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#addHoursTo_java_util_Calendar__int_
add minutes to	sM	Calendar(Calendar, int)	addMinutesTo	Returns a new Calendar that is arg2 minutes later than arg1. JavaDate.add minutes to("01:01:01", 1)=="01:02:01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#addMinutesTo_java_util_Calendar__int_
add seconds to	sM	Calendar(Calendar, int)	addSecondsTo	Returns a new Calendar that is arg2 seconds later than arg1. JavaDate.add seconds to("01:01:01", 61)=="01:02:02".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#addSecondsTo_java_util_Calendar__int_
add milliseconds to	sM	Calendar(Calendar, int)	addMillisecondsTo	Returns a new Calendar that is arg2 milliseconds later than arg1. JavaDate.add milliseconds to("01:01:01", 61)=="01:01:01.061".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#addMillsecondsTo_java_util_Calendar__int_

Table B-15 (Cont.) Table lists the JavaDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
add duration to	sM	Calendar(Calendar, XMLDuration)	addDurationTo	Returns a new Calendar that is later than arg1 by the duration arg2. JavaDate.add duration to("2009-12-30T23:59:00",Duration.from string("P1DT1M"))=="2010-01-01".	http://www.w3.org/TR/rif-dtb/#func:add-day TimeDuration-to-dateTime_.28adapted_from_op: add-dayTimeDuration-to-dateTime.29 http://www.w3.org/TR/rif-dtb/#func: add-yearMonthDuration-to-dateTime_.28adapted_from_op: add-yearMonthDuration-to-dateTime.29
from date string	sM	Calendar(String)	fromDateString	Creates a Calendar for the extended ISO 8601 date literal arg1. Extended to allow YYYY-MM-DD@TimeZoneId. JavaDate.from date string("2010-02-06@PST")=="2010-02-06-08:00".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#fromDateString_java_lang_String_
from datetime string	sM	Calendar(String)	fromDateTimeString	Creates a Calendar for the extended ISO 8601 datetime literal arg1. Extended to allow YYYY-MM-DDTHH:MM:SS@TimeZoneId. JavaDate.from datetime string("2010-02-06T14:15:00@PST")=="2010-02-06T14:15:00-08:00".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#fromDateTimeString_java_lang_String_

Table B-15 (Cont.) Table lists the JavaDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
from time string	sM	Calendar(String)	fromTimeString	Creates a Calendar for the extended ISO 8601 time literal arg1. Extended to allow HH:MM:SS@TimeZoneId. Warning: the date portion of the Calendar will be initialized to the current date. JavaDate.from time string("14:15:00@PST")=="14:15:00-08:00".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#fromTimeString_java_lang_String_
subtract years from	sM	Calendar(Calendar, int)	subtractYearsFrom	Returns a new Calendar that is arg2 years earlier than arg1. JavaDate.subtract years from("2009-01-01", 1)=="2008-01-01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#subtractYearsFrom_java_util_Calendar__int_
subtract months from	sM	Calendar(Calendar, int)	subtractMonthsFrom	Returns a new Calendar that is arg2 months earlier than arg1. JavaDate.subtract months from("2009-01-01", 1)=="2008-12-01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#subtractMonthsFrom_java_util_Calendar__int_

Table B-15 (Cont.) Table lists the JavaDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
subtract weeks from	sM	Calendar(Calendar, int)	subtract WeeksF rom	Returns a new Calendar that is 7*arg2 days earlier than arg1. JavaDate.subt ract weeks from("2009-01 -01", 1)=="2008-12- 25".	http://download.oracle.com/ docs/cd/E12839_01/apirefs.1111/ e10663/oracle/rules/rl/extensions/ JavaDate.html#subtractWeeksFrom_ja va_util_Calendar__int_
subtract days from	sM	Calendar(Calendar, int)	subtract DaysFr om	Returns a new Calendar that is arg2 days earlier than arg1. JavaDate.subt ract days from("2009-01 -01", 1)=="2008-12- 31".	http://download.oracle.com/ docs/cd/E12839_01/apirefs.1111/ e10663/oracle/rules/rl/extensions/ JavaDate.html#subtractDaysFrom_jav a_util_Calendar__int_
subtract hours from	sM	Calendar(Calendar, int)	subtract HoursF rom	Returns a new Calendar that is arg2 hours earlier than arg1. JavaDate.subt ract hours from("01:01:0 1", 1)=="00:01:01 ".	http://download.oracle.com/ docs/cd/E12839_01/apirefs.1111/ e10663/oracle/rules/rl/extensions/ JavaDate.html#subtractHoursFrom_ja va_util_Calendar__int_
subtract minutes from	sM	Calendar(Calendar, int)	subtract Minutes From	Returns a new Calendar that is arg2 minutes earlier than arg1. JavaDate.subt ract minutes from("01:01:0 1", 1)=="01:00:01 ".	http://download.oracle.com/ docs/cd/E12839_01/apirefs.1111/ e10663/oracle/rules/rl/extensions/ JavaDate.html#subtractMinutesFrom_ java_util_Calendar__int_

Table B-15 (Cont.) Table lists the JavaDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
subtract seconds from	sM	Calendar(Calendar, int)	subtract Seconds From	Returns a new Calendar that is arg2 seconds earlier than arg1. JavaDate.subtract seconds from("01:01:01", 61)=="01:00:00".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#subtractSecondsFrom_java_util_Calendar__int_
subtract milliseconds from	sM	Calendar(Calendar, int)	subtract MillisecondsFrom	Returns a new Calendar that is arg2 milliseconds earlier than arg1. JavaDate.subtract milliseconds from("01:01:01", 61)=="01:01:00.939".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#subtractMillisecondsFrom_java_util_Calendar__int_
subtract duration from	sM	Calendar(Calendar, XMLDuration)	subtract DurationFrom	Returns a new Calendar that is earlier than arg1 by the duration arg2. JavaDate.subtract duration from("2009-12-30T23:59:00", Duration.from string("P1DT1M"))=="2009-12-29T23:58:00".	http://www.w3.org/TR/rif-dtb/#func: add-dayTimeDuration-to-dateTime_.28adapted_from_op: subtract-dayTimeDuration-from-dateTime.29 http://www.w3.org/TR/rif-dtb/#func: subtract-yearMonthDuration-from-dateTime_.28adapted_from_op: add-yearMonthDuration-to-dateTime.29

Table B-15 (Cont.) Table lists the JavaDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
to date string	sM	String(Calendar)	toDateString	Returns the ISO 8601 lexical representation of arg1, ignoring time components. JavaDate.to date string("2010-07-04T12:30:00Z")=="2010-07-04Z"	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#toDateString_java_util_Calendar_
to datetime string	sM	String(Calendar)	toDateTimeString	Returns the ISO 8601 lexical representation of arg1. JavaDate.to datetime string("2010-07-04T12:30:00Z")=="2010-07-04T12:30:00.000Z"	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#toDateTimeString_java_util_Calendar_
to time string	sM	String(Calendar)	toTimeString	Returns the ISO 8601 lexical representation of arg1, ignoring date components. JavaDate.to time string("2010-07-04T12:30:00Z")=="12:30:00.000Z"	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/JavaDate.html#toTimeString_java_util_Calendar_

Table B-16 lists the `XMLGregorianCalendar` class.

Table B-16 Table lists the XMLGregorianCalendar class.

OBR Name	Kind	Signature	Java Name	Description	Reference
XMLGregorianCalendar	**Cl**	-	**javax.xml.datatype.XMLGregorianCalendar**	**Representation for W3C XML Schema 1.0 date/time datatypes.**	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html
normalize	M	XMLGregorianCalendar()	-	Normalizes this instance to UTC. XMLDate.fromstring("2000-03-04T23:00:00+03:00").normalize()==XMLDate.fromstring("2000-03-04T20:00:00Z")	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#normalize()
toGregorianCalendar	M	java.util.GregorianCalendar()	-	Converts this XMLGregorianCalendar to a (superclass of) Calendar. XMLDate.fromstring("2010-02-03").toGregorianCalendar()==(Calendar)"2010-02-03".	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#toGregorianCalendar()

Table B-16 (Cont.) Table lists the XMLGregorianCalendar class.

OBR Name	Kind	Signature	Java Name	Description	Reference
year	P	int	-	The year of this calendar, or Integer.MIN_VALUE if undefined. XMLDate.fromstring("2011-12-31").year==2011.	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#getYear()
month	P	int	-	The month of this calendar, or Integer.MIN_VALUE if undefined. Months are 1-based, e.g. Jan is month 1. XMLDate.fromstring("2011-12-31").month==12.	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#getMonth()
day	P	int	-	The day of this calendar, or Integer.MIN_VALUE if undefined. XMLDate.fromstring("2011-12-31").day==31.	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#getDay()

Table B-16 (Cont.) Table lists the XMLGregorianCalendar class.

OBR Name	Kind	Signature	Java Name	Description	Reference
hour	P	int	-	The hour of this calendar, or Integer.MIN_VALUE if undefined. XMLDate.fromstring("2011-12-31").hour==Integer.MIN_VALUE.	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#getHour()
minute	P	int	-	The minute of this calendar, or Integer.MIN_VALUE if undefined. XMLDate.fromstring("2011-12-31T09:30:00").minute==30.	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#getMinute()
second	P	int	-	The second of this calendar, or Integer.MIN_VALUE if undefined. XMLDate.fromstring("09:30:05Z").second==5.	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#getSecond()

Table B-16 (Cont.) Table lists the XMLGregorianCalendar class.

OBR Name	Kind	Signature	Java Name	Description	Reference
timezone	P	int	-	The timezone offset in minutes of this calendar, or Integer.MIN_VALUE if undefined. XMLDate.fromstring("09:30:00-09:00").timezone==-540.	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/XMLGregorianCalendar.html#getTimezone()

Table B-17 lists the `XMLDate` class.

Table B-17 Table lists the XMLDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
XMLDate	**Cl**	-	**oracle.rules.rl.extensions.XMLDate**	**Helper class for working with XMLGregorianCalendars as immutable objects. Treating calendars as immutable objects can help prevent errors.**	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html
add years to	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	addYearsTo	Returns a new XMLGregorianCalendar that is arg2 years later than arg1. XMLDate.addyearsto("2009-01-01", 1)=="2010-01-01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#addYearsTo_javax_xml_datatype_XMLGregorianCalendar__int_

Table B-17 (Cont.) Table lists the XMLDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
add months to	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	addMonthsTo	Returns a new XMLGregorianCalendar that is arg2 months later than arg1. XMLDate.add months to("2009-01-01", 1)=="2009-02-01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#addMonthsTo_javax_xml_datatype_XMLGregorianCalendar__int_
add weeks to	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	addWeeksTo	Returns a new XMLGregorianCalendar that is 7*arg2 days later than arg1. XMLDate.add weeks to("2009-01-01", 1)=="2009-01-08".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#addWeeksTo_javax_xml_datatype_XMLGregorianCalendar__int_
add days to	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	addDaysTo	Returns a new XMLGregorianCalendar that is arg2 days later than arg1. XMLDate.add days to("2009-01-01", 1)=="2009-01-02".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#addDaysTo_javax_xml_datatype_XMLGregorianCalendar__int_
add hours to	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	addHoursTo	Returns a new XMLGregorianCalendar that is arg2 hours later than arg1. XMLDate.add hours to("01:01:01", 1)=="02:01:01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#addHoursTo_javax_xml_datatype_XMLGregorianCalendar__int_
add minutes to	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	addMinutesTo	Returns a new XMLGregorianCalendar that is arg2 minutes later than arg1. XMLDate.add minutes to("01:01:01", 1)=="01:02:01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#addMinutesTo_javax_xml_datatype_XMLGregorianCalendar__int_

Table B-17 (Cont.) Table lists the XMLDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
add seconds to	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	addSecondsTo	Returns a new XMLGregorianCalendar that is arg2 seconds later than arg1. XMLDate.add seconds to("01:01:01", 61)=="01:02:02".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#addSecondsTo_javax_xml_datatype_XMLGregorianCalendar__int_
add milliseconds to	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	addMillisecondsTo	Returns a new XMLGregorianCalendar that is arg2 milliseconds later than arg1. XMLDate.add milliseconds to("01:01:01", 61)=="01:01:01.061".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#addMillisecondsTo_javax_xml_datatype_XMLGregorianCalendar__int_
add duration to	sM	XMLGregorianCalendar(XMLGregorianCalendar,XMLDuration)	addDurationTo	Returns a new XMLGregorianCalendar that is later than arg1 by the duration arg2. XMLDate.add duration to("2009-12-30T23:59:00",Duration.from string("P1DT1M"))=="2010-01-01".	http://www.w3.org/TR/rif-dtb/#func:add-yearMonthDuration-to-dateTime_.28 adapted_from_op:add-yearMonthDuration-to-dateTime.29 http://www.w3.org/TR/rif-dtb/#func:add-dayTimeDuration-to-dateTime_.28 adapted_from_op:add-dayTimeDuration-to-dateTime.29
from string	sM	XMLGregorianCalendar(String)	fromString	Creates an XMLGregorianCalendar for the ISO 8601 date literal arg1. XMLDate.from string("2010-02-06-08:00")=="2010-02-06-08:00".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#fromString_java_lang_String_

Table B-17 (Cont.) Table lists the XMLDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
subtract years from	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	subtractYearsFrom	Returns a new XMLGregorianCalendar that is arg2 years earlier than arg1. XMLDate.subtract years from("2009-01-01", 1)=="2008-01-01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#subtractYearsFrom_javax_xml_datatype_XMLGregorianCalendar__int_
subtract months from	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	subtractMonthsFrom	Returns a new XMLGregorianCalendar that is arg2 months earlier than arg1. XMLDate.subtract months from("2009-01-01", 1)=="2008-12-01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#subtractMonthsFrom_javax_xml_datatype_XMLGregorianCalendar__int_
subtract weeks from	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	subtractWeeksFrom	Returns a new XMLGregorianCalendar that is 7*arg2 days earlier than arg1. XMLDate.subtract weeks from("2009-01-01", 1)=="2008-12-25".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#subtractWeeksFrom_javax_xml_datatype_XMLGregorianCalendar__int_
subtract days from	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	subtractDaysFrom	Returns a new XMLGregorianCalendar that is arg2 days earlier than arg1. XMLDate.subtract days from("2009-01-01", 1)=="2008-12-31".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#subtractDaysFrom_javax_xml_datatype_XMLGregorianCalendar__int_
subtract hours from	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	subtractHoursFrom	Returns a new XMLGregorianCalendar that is arg2 hours earlier than arg1. XMLDate.subtract hours from("01:01:01", 1)=="00:01:01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#subtractHoursFrom_javax_xml_datatype_XMLGregorianCalendar__int_

Table B-17 (Cont.) Table lists the XMLDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
subtract minutes from	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	subtractMinutesFrom	Returns a new XMLGregorianCalendar that is arg2 minutes earlier than arg1. XMLDate.subtract minutes from("01:01:01", 1)=="01:00:01".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#subtractMinutesFrom_javax_xml_datatype_XMLGregorianCalendar__int_
subtract seconds from	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	subtractSecondsFrom	Returns a new XMLGregorianCalendar that is arg2 seconds earlier than arg1. XMLDate.subtract seconds from("01:01:01", 61)=="01:00:00".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#subtractSecondsFrom_javax_xml_datatype_XMLGregorianCalendar__int_
subtract milliseconds from	sM	XMLGregorianCalendar(XMLGregorianCalendar,int)	subtractMillisecondsFrom	Returns a new XMLGregorianCalendar that is arg2 milliseconds earlier than arg1. XMLDate.subtract milliseconds from("01:01:01", 61)=="01:01:00.939".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#subtractMillisecondsFrom_javax_xml_datatype_XMLGregorianCalendar__int_
subtract duration from	sM	XMLGregorianCalendar(XMLGregorianCalendar,XMLDuration)	subtractDurationFrom	Returns a new XMLGregorianCalendar that is earlier than arg1 by the duration arg2. XMLDate.subtract duration from("2009-12-30T23:59:00",Duration.from string("P1DT1M"))=="20009-12-29T23:58:00".	http://www.w3.org/TR/rif-dtb/#func:subtract-yearMonthDuration-from-dateTime_.28 adapted_from_op:subtract-yearMonthDuration-from-dateTime.29 http://www.w3.org/TR/rif-dtb/#func:subtract-dayTimeDuration-from-dateTime_.28 adapted_from_op:subtract-dayTimeDuration-from-dateTime.29

Table B-17 (Cont.) Table lists the XMLDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
to string	sM	String(XMLGregorianCalendar)	toString	Returns the ISO 8601 lexical representation of arg1. XMLDate.to string("2010-04-15T11:00:00-09:00")==="2010-04-15T11:00:00-09:00".	http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10663/oracle/rules/rl/extensions/XMLDate.html#toString_javax_xml_datatype_XMLGregorianCalendar_
is datetime	sM	boolean(XMLGregorianCalendar)	isDateTime	Checks if this calendar have both date and time fields. XMLDate.is datetime("2009-12-30T23:59:00")==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
is datetime stamp	sM	boolean(XMLGregorianCalendar)	isDateTimeStamp	Checks if this calendar have date, time, and timezone fields. XMLDate.is datetime stamp("2009-12-30T23:59:00")==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
is date	sM	boolean(XMLGregorianCalendar)	isDate	Checks if this calendar have date fields and no time fields. XMLDate.is date("2009-12-30")==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
is time	sM	boolean(XMLGregorianCalendar)	isTime	Checks if this calendar have time fields and no date fields. XMLDate.is time("2009-12-30T23:59:00")==false.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
get timezone	sM	XMLDuration(XMLGregorianCalendar)	getTimezone	Gets the timezone from the calendar as a duration. XMLDate.get timezone("11:00:00+05:30")==Duration.from string("PT5H30M").	-

Table B-17 (Cont.) Table lists the XMLDate class

OBR Name	Kind	Signature	Java Name	Description	Reference
get seconds	sM	BigDecimal(XMLGregorianCalendar)	getSeconds	Gets the seconds, including fractional part, from the calendar as a BigDecimal. XMLDate.get seconds("00:00:12.345")==12.345.	-

Table B-18 lists the `OracleDate` class.

Table B-18 Table lists the OracleDate class

OBR Name	Kind	Signature	Java Name	Description
OracleDate	**Cl**	-	oracle.rules.sdk2.extensions.OracleDate	Helper class for working with oracle.jbo.domain.Timestamp. For examples of method use, see like-named XMLDate methods.
add years to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	addYearsTo	Returns a new oracle.jbo.domain.Timestamp that is arg2 years later than arg1.
add months to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	addMonthsTo	Returns a new oracle.jbo.domain.Timestamp that is arg2 months later than arg1.
add weeks to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	addWeeksTo	Returns a new oracle.jbo.domain.Timestamp that is 7*arg2 days later than arg1.
add days to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	addDaysTo	Returns a new oracle.jbo.domain.Timestamp that is arg2 days later than arg1.
add hours to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	addHoursTo	Returns a new oracle.jbo.domain.Timestamp that is arg2 hours later than arg1.
add minutes to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	addMinutesTo	Returns a new oracle.jbo.domain.Timestamp that is arg2 minutes later than arg1.
add seconds to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	addSecondsTo	Returns a new oracle.jbo.domain.Timestamp that is arg2 seconds later than arg1.
add milliseconds to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	addMillisecondsTo	Returns a new oracle.jbo.domain.Timestamp that is arg2 milliseconds later than arg1.

Table B-18 (Cont.) Table lists the OracleDate class

OBR Name	Kind	Signature	Java Name	Description
add duration to	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,XMLDuration)	addDurationTo	Returns a new oracle.jbo.domain.Timestamp that is later than arg1 by the duration arg2.
from string	sM	oracle.jbo.domain.Timestamp(String)	fromString	Creates an oracle.jbo.domain.Timestamp for the ISO 8601 date literal arg1.
subtract years from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	subtractYearsFrom	Returns a new oracle.jbo.domain.Timestamp that is arg2 years earlier than arg1.
subtract months from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	subtractMonthsFrom	Returns a new oracle.jbo.domain.Timestamp that is arg2 months earlier than arg1.
subtract weeks from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	subtractWeeksFrom	Returns a new oracle.jbo.domain.Timestamp that is 7*arg2 days earlier than arg1.
subtract days from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	subtractDaysFrom	Returns a new oracle.jbo.domain.Timestamp that is arg2 days earlier than arg1.
subtract hours from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	subtractHoursFrom	Returns a new oracle.jbo.domain.Timestamp that is arg2 hours earlier than arg1.
subtract minutes from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	subtractMinutesFrom	Returns a new oracle.jbo.domain.Timestamp that is arg2 minutes earlier than arg1.
subtract seconds from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	subtractSecondsFrom	Returns a new oracle.jbo.domain.Timestamp that is arg2 seconds earlier than arg1.
subtract milliseconds from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,int)	subtractMillisecondsFrom	Returns a new oracle.jbo.domain.Timestamp that is arg2 milliseconds earlier than arg1.
subtract duration from	sM	oracle.jbo.domain.Timestamp(oracle.jbo.domain.Timestamp,XMLDuration)	subtractDurationFrom	Returns a new oracle.jbo.domain.Timestamp that is earlier than arg1 by the duration arg2.
to string	sM	String(oracle.jbo.domain.Timestamp)	toString	Returns the ISO 8601 lexical representation of arg1.

Table B-19 lists the `Duration` class.

Table B-19 Table lists the Duration class

OBR Name	Kind	Signature	Java Name	Description	Reference
Duration	**Cl**	-	oracle.rules.sdk2.extensions.OracleDuration	Helper class for comparing and subtracting dates. Can convert the difference of 2 dates into an XMLDuration. Can also create an XMLDuration from its literal (String) representation. Only day time and year month XMLDurations are supported.	-
compare	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	-	Returns -1, 0, or 1 according to whether arg1<arg2, arg1==arg2, or arg1>arg2, respectively. Duration.compare("2010-01-01","2010-02-02")==-1	http://www.w3.org/TR/rif-dtb/#pred:date Time-less-than_.28adapted_from_op:dateTime-less-than.29
years between	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	yearsBetween	Subtracts arg1 from arg2, where the args are some kind of date/time. Duration.years between("2008-01-01", "2009-02-02")==1.	-
months between	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	monthsBetween	Subtracts arg1 from arg2, where the args are some kind of date/time. Duration.months between("2009-01-01","2008-02-02")==-10.	-

Table B-19 (Cont.) Table lists the Duration class

OBR Name	Kind	Signature	Java Name	Description	Reference
weeks between	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	weeksBetween	Subtracts arg1 from arg2, where the args are some kind of date/time. Duration.weeksbetween("2000-01-01","2000-02-04")==4.	-
days between	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	daysBetween	Subtracts arg1 from arg2, where the args are some kind of date/time. Duration.daysbetween("2000-01-01","2000-02-04")==34.	-
hours between	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	hoursBetween	Subtracts arg1 from arg2, where the args are some kind of date/time. Duration.hoursbetween("2000-01-04T03:30:00","2000-01-01T00:00:00")==-75	-
minutes between	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	minutesBetween	Subtracts arg1 from arg2, where the args are some kind of date/time. Duration.minutesbetween("03:30:00","04:45:00")==75.	-

Table B-19 (Cont.) Table lists the Duration class

OBR Name	Kind	Signature	Java Name	Description	Reference
seconds between	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	secondsBetween	Subtracts arg1 from arg2, where the args are some kind of date/time. Duration.seconds between("03:30:00","03:31:15")==75.	-
milliseconds between	sM	int(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	millisecondsBetween	Subtracts arg1 from arg2, where the args are some kind of date/time. Duration.milliseconds between("03:30:00","03:31:15")==75000.	
between	sM	XMLDuration(Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp, Calendar | XMLGregorianCalendar | oracle.jbo.domain.Timestamp)	between	Subtracts arg1 from arg2, where the args are some kind of date/time. Returns day-time Duration. Duration.between("2009-01-01T01:15:00","2009-02-02T11:30:00")==Duration.from string("P32DT10H15M").	http://www.w3.org/TR/rif-dtb/#func:subtract-date Times_.28adapted_from_op:subtract-dateTimes.29
from string	sM	XMLDuration(String)	fromString	Parses a duration from an ISO 8601 duration literal. "P1DT2H3M" is the duration of 1 day, 2 hours, and 3 minutes.	http://www.w3.org/TR/xpath-functions/#duration-subtypes

Table B-19 (Cont.) Table lists the Duration class

OBR Name	Kind	Signature	Java Name	Description	Reference
compare durations	sM	int(XMLDuration,XMLDuration)	compareDurations	Compares two durations. Both must be either day-time or year-month durations. Returns -1, 0, or 1 according to whether arg1<arg2, arg1==arg2, or arg1>arg2, respectively. Duration.compare(Duration.from string("P1Y"),Duration.from string("P13M"))==-1.	http://www.w3.org/TR/rif-dtb/#pred:dayTimeDuration-less-than_.28 adapted_from_op:dayTimeDuration-less-than.29 http://www.w3.org/TR/rif-dtb/#pred:yearMonthDuration-less-than_.28 adapted_from_op:yearMonthDuration-less-than.29
is day-time duration	sM	boolean(XMLDuration)	isDayTimeDuration	Checks if arg1 a day-time duration. Only day-time and year-month durations are supported. Duration.is day-time duration(Duration.from string("P2DT1S"))==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
is year-month duration	sM	boolean(XMLDuration)	isYearMonthDuration	Checks if arg1 a year-month duration. Only day-time and year-month durations are supported. Duration.is year-month duration(Duration.from string("P13M"))==true.	http://www.w3.org/TR/rif-dtb/#Guard_Predicates_for_Datatypes
get seconds	sM	BigDecimal(XMLDuration)	getSeconds	Gets the seconds field from the duration as a BigDecimal, including fractional seconds. Duration.get seconds(Duraton.from string("PT12.345S"))==12.345.	http://www.w3.org/TR/rif-dtb/#func:seconds-from-duration_.28 adapted_from_fn:seconds-from-duration.29

Table B-19 (Cont.) Table lists the Duration class

OBR Name	Kind	Signature	Java Name	Description	Reference
divide	sM	XMLDuration(XMLDuration,int|double)	-	Divides a duration by an integral or double divisor. Duration.divide(Duration.from string("P1Y"), 4)==Duration.from string("P3M").	http://www.w3.org/TR/rif-dtb/#func:divide-dayTimeDuration_.28adapted_from_op:divide-dayTimeDuration.29 http://www.w3.org/TR/rif-dtb/#func:divide-yearMonthDuration_.28adapted_from_op:divide-yearMonthDuration.29
ratio	sM	BigDecimal(XMLDuration,XMLDuration)	-	Computes the ratio of 2 durations as a BigDecimal. Duration.ratio(Duration.from string("P1Y"),Duration.from string("P3M"))==4	http://www.w3.org/TR/rif-dtb/#func:divide-dayTimeDuration-by-dayTimeDuration_.28adapted_from_op:divide-dayTimeDuration-by-dayTimeDuration.29 http://www.w3.org/TR/rif-dtb/#func:divide-yearMonthDuration-by-yearMonthDuration_.28adapted_from_op:divide-yearMonthDuration-by-yearMonthDuration.29

Table B-20 lists the `XMLDuration` class.

Table B-20 Table lists the XMLDuration class

OBR Name	Kind	Signature	Java Name	Description	Reference
XMLDuration	**Cl**	-	**javax.xml.datatype.Duration**	**Immutable representation of a time span as defined in the W3C XML Schema 1.0 specification. Only day-time and year-month XMLDurations are supported.**	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/Duration.html http://www.w3.org/TR/xpath-functions/#duration-subtypes
years	P	int	-	Years field of the duration. Duration.from string("P2Y3M").years==2.	http://www.w3.org/TR/rif-dtb/#func: years-from-duration_.28adapted_from_fn:years-from-duration.29

Table B-20 (Cont.) Table lists the XMLDuration class

OBR Name	Kind	Signature	Java Name	Description	Reference
months	P	int	-	Months field of the duration. Duration.from string("P2Y3M").months==2.	http://www.w3.org/TR/rif-dtb/#func: months-from-duration_.28adapted_from_fn:months-from-duration.29
days	P	int	-	Days field of the duration. Duration.from string("P1DT2H3M4S").days==1.	http://www.w3.org/TR/rif-dtb/#func: days-from-duration_.28adapted_from_fn:days-from-duration.29
hours	P	int	-	Hours field of the duration. Duration.from string("P1DT2H3M4S").hours==2.	http://www.w3.org/TR/rif-dtb/#func: hours-from-duration_.28adapted_from_fn:hours-from-duration.29
minutes	P	int	-	Minutes field of the duration. Duration.from string("P1DT2H3M4S").minutes==3.	http://www.w3.org/TR/rif-dtb/#func: minutes-from-duration_.28adapted_from_fn:minutes-from-duration.29
seconds	P	int	-	Seconds field of the duration. Duration.from string("P1DT2H3M4S").seconds==4.	http://www.w3.org/TR/rif-dtb/#func: seconds-from-duration_.28adapted_from_fn:seconds-from-duration.29
sign	P	int	-	Returns the sign of this duration in -1,0, or 1. Duration.from string("-P1Y").sign==-1.	-
add	M	XMLDuration(XMLDuration)	-	Adds two durations. Duration.from string("P6M").add(Duration.from string("P6M"))==Duration.from string("P1Y").	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/Duration.html#add(javax.xml.datatype.Duration)

Table B-20 (Cont.) Table lists the XMLDuration class

OBR Name	Kind	Signature	Java Name	Description	Reference
subtract	M	XMLDuration(XMLDuration)	-	Subtracts two durations. Duration.from string("P6M").subtract(Duration.from string("P6M"))==Duration.from string("P0Y").	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/Duration.html#subtract(javax.xml.datatype.Duration)
multiply	M	XMLDuration(BigDecimal | int)	-	Multiplies arg1 duration by arg2 factor. Duration.from string("P6M").multiply(2)==Duration.from string("P1Y").	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/Duration.html#multiply(java.math.BigDecimal)
negate	M	XMLDuration()	-	Durations can be negative, e.g. if you reverse the arguments to Duration.between(arg1,arg2). Duration.from string("P6M").negate()==Duration.from string("-P6M").	http://java.sun.com/javase/6/docs/api/javax/xml/datatype/Duration.html#negate()
to string	M	String()	toString	Gets the ISO8601 literal representation for this duration. Duration.from string("P6M").to string()=="P6M".	-

Table B-21 lists the `CurrentDate` class.

Table B-21 Table lists the CurrentDate class

OBR Name	Kind	Signature	Java Name	Description
CurrentDate	**Cl**	-	oracle.rules.rl.extensions.CurrentDate	Fact type of a holder for the current date. Can be used in rule patterns.
date	P	Calendar	-	Returns the current date.

Miscellaneous Classes

This section covers the miscellaneous classes provided by Oracle Business Rules.

Table Table B-22 lists the `JAXBElement` class.

Table B-22 Table lists the JAXBElement class

OBR Name	Kind	Signature	Java Name	Description	Reference
JAXBElement	**Cl**	**-**	**javax.xml.bind.JAXBElement**	**Represents XML element information in XML Fact Types.**	`http://java.sun.com/javase/6/docs/api/javax/xml/bind/JAXBElement.html`
nil	P	boolean	-	A nil element is not the same thing (in XML) as an absent element.	`http://java.sun.com/javase/6/docs/api/javax/xml/bind/JAXBElement.html#isNil()`
value	P	Object	-	This is a reference to an XML Fact Type	`http://java.sun.com/javase/6/docs/api/javax/xml/bind/JAXBElement.html#getValue()`

Table B-23 lists the `Object` class.

Table B-23 Table lists the Object class

OBR Name	Kind	Signature	Java Name	Description	Reference
Object	**Cl**	**-**	**java.lang.Object**	**Base class of all Java objects.**	`http://java.sun.com/javase/6/docs/api/java/lang/Object.html`

Functions

This section lists the Oracle Business Rules functions.

Table Table B-24 lists the different functions provided by Oracle Business Rules.

Table B-24 Table lists the different functions provided by Oracle Business Rules

OBR Name	Signature	RL Name	Description	Reference
print	void(Object)	println	Prints the string value of arg1.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.assert a tree of facts	Object(Object)	assertTree	Asserts (insert into working memory) the tree of visible fact types with arg1 as the root. Returns arg1.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules

Table B-24 (Cont.) Table lists the different functions provided by Oracle Business Rules

OBR Name	Signature	RL Name	Description	Reference
RL.assert	Object(Object)	assert	Asserts arg1 (insert arg1 into working memory). Returns arg1.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.retract	void(Object)	retract	Removes the fact associated with the object arg1 from working memory.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.get fact ID	int(Object)	id	Returns the fact id associated with the object arg1. If arg1 is not associated with a fact, return -1.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.get fact by ID	Object(int)	object	Returns the object associated with the given fact id. If there is no such fact id, returns null.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.get firing rule name	String	-	Returns the name of the currently firing rule if it is invoked in a rule action. Returns null otherwise.	-
RL.contains	boolean(List,Object)	contains	The contains() function is similar to the contains() method on Java Collection but includes the ability to handle the presence of JAXBElement in the collection.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.suppress rule test errors	boolean(boolean)	-	Update if errors during rule test evaluation should be suppressed by the rules engine.	-
RL.are rule test errors suppressed	boolean()	-	Query if errors during rule test evaluation are suppressed by the rules engine.	-
RL.ruleset stack.push	void(String)	pushRuleset	Pushes arg1, the name of a ruleset, onto the ruleset stack.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules

Table B-24 (Cont.) Table lists the different functions provided by Oracle Business Rules

OBR Name	Signature	RL Name	Description	Reference
RL.ruleset stack.pop	String()	popRuleset	Pops and returns the top of the ruleset stack, the name of a ruleset.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.ruleset stack.get	String[]()	getRulesetStack	Returns the ruleset stack as a String array.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.ruleset stack.set	void(String[])	setRulesetStack	Sets the ruleset stack to arg1, a String array.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.ruleset stack.clear	void()	clearRulesetStack	Pops all ruleset names off the ruleset stack.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.date.get current	Calendar()	getCurrentDate	Returns the date associated with the CurrentDate fact.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.date.set current	void(Calendar)	setCurrentDate	Sets the date for reasoning on an engine managed fact representing the "current" date (with the CurrentDate fact).	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.date.get effective	Calendar()	getEffectiveDate	Returns the current value of the effective date.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.date.set effective	void(Calendar)	setEffectiveDate	Updates the effective date in the rules engine.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.watch.rules	void()	watchRules	Prints information about rule firings (execution of activations).	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules

Table B-24 (Cont.) Table lists the different functions provided by Oracle Business Rules

OBR Name	Signature	RL Name	Description	Reference
RL.watch.activations	void()	watchActivations	Prints information about addition or removal of activations from the agenda.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.watch.facts	void()	watchFacts	Prints information about assertion, retraction, or modification of facts in working memory.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.watch.focus	void()	watchFocus	Prints information about pushing and popping of the ruleset stack.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.watch.compilations	void()	watchCompilations	Prints information about how the condition parts of a rule are shared with existing rules.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.watch.all	void()	watchAll	Prints information about rules, facts, activations, focus, and compilations.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.stop watching.rules	void()	clearWatchRules	Stops printing information about rule firings.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.stop watching.activations	void()	clearWatchActivations	Stops printing information about addition or removal of activations from the agenda.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.stop watching.facts	void()	clearWatchFacts	Stops printing information about assertion, retraction, or modification of facts in working memory.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.stop watching.focus	void()	clearWatchFocus	Stops printing information about pushing and popping of the ruleset stack.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules

Table B-24 (Cont.) Table lists the different functions provided by Oracle Business Rules

OBR Name	Signature	RL Name	Description	Reference
RL.stop watching.compilations	void()	clearWatchCompilations	Stops printing information about how the condition parts of a rule are shared with existing rules.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.stop watching.all	void()	clearWatchAll	Stops printing information about rules, facts, activations, focus, and compilations.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.show.facts	void()	showFacts	Prints all facts in working memory.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules
RL.show.activations	void()	showActivations	Prints all activations on the agenda.	Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules

C

Oracle Business Rules Frequently Asked Questions

This appendix contains frequently asked questions about Oracle Business Rules.

Why Do Rules Not Fire When A Java Object is Asserted as a Fact and Then Changed Without Using the Modify Action?

When a Java object has been asserted and then the object is changed without using the modify action, the object must be re-asserted in the Rules Engine.

Therefore, if a rule associated with the changed Java object does not fire, this means that the Rules Engine did not reevaluate any rule conditions and did not activate any rules. Thus, when a Java object changes without using the modify action, the object must be re-asserted in the Rules Engine.

What are the Differences Between Oracle Business Rules RL Language and Java?

There are many differences between Oracle Business Rules RL Language and Java.

For more information on the differences between Oracle Business Rules RL Language and Java, see Appendix A in *Rules Language Reference for Oracle Business Process Management*.

How Does a RuleSession Handle Concurrency and Synchronization?

Method calls on an Oracle Business Rules RuleSession object are thread-safe such that calls by multiple threads do not cause exceptions at the RuleSession level. However, there are no exclusivity or transactional guarantees on the execution of methods. The lowest-level `run` method in the Rules Engine is synchronized, so two threads with a shared `RuleSession` cannot both simultaneously execute `run`. One call to `run` must wait for the other to finish.

Oracle Business Rules functions are not synchronized by default. Like Java methods, Oracle Business Rules functions can execute concurrently and it is the programmer's responsibility to use synchronized blocks to protect access to shared data (for instance, a `HashMap` containing results data).

Any set of actions that a user wants to be executed as in a transaction-like form must synchronize around the shared object. Users should not synchronize around a `RuleSession` object because exceptions thrown when calling `RuleSession` methods may require the `RuleSession` object to be discarded.

For most uses of a `RuleSession` object in Oracle Business Rules, each thread or servlet instance should create and use a local `RuleSession` object. This usage pattern is roughly analogous to using a JDBC connection in this manner.

The following examples demonstrate how to use a shared `RuleSession` object.

For the case where Thread-1 includes the following:

```
ruleSession.callFunctionWithArgument("assert", singleFact1);
ruleSession.callFunctionWithArgument("assert", singleFact2);
```

and Thread-2 includes the following:

```
ruleSession.callFunction("run");
ruleSession.callFunction("clear");
```

In this case, execution of the two threads might proceed as shown below in code example showing a shared `rulesession` object in Oracle Business Rules:

```
Thread-1:  ruleSession.callFunctionWithArgument("assert", singleFact1);
Thread-2:  ruleSession.callFunction("run");
Thread-2:  ruleSession.callFunction("clear");
Thread-1:  ruleSession.callFunctionWithArgument("assert", singleFact2);
```

In the example above, the two facts Thread-1 asserted are never both in the `RuleSession` during a call to `run`. Notice also that only one thread calls the `run` method. If you use a design where multiple threads can call `run` on a shared `RuleSession`, this can create extremely hard to find bugs and there is usually no gain in performance.

All accesses to a shared `RuleSession` object must be synchronized to ensure the intended behavior. However, a `RuleSession` instance may throw an exception and

not be recoverable, so do not use this object as the synchronization object. Instead, use another shared object as the synchronization point.

One can envision a shared server process producer-consumer model for `RuleSession` use. In this model, multiple threads assert facts to a shared `RuleSession` and one thread periodically calls `run`, reads any results, and outputs them. This ensures that thread conflicts cannot occur, because the two code segments must be executed serially and cannot be intermingled. For example, the code with shared objects, producer code, and consumer code in Example C-1, Example C-2, and Example C-3.

Sample RuleSession Shared Objects

Example C-1 shows the code with shared objects.

Example C-1 RuleSession Shared Objects

```
RuleSession ruleSession;
Object ruleSessionLock = new Object();
```

Sample RuleSession Producer Code

Example C-2 shows the producer code.

Example C-2 RuleSession Producer Code

```
public String addFacts(FactTypeA fa, FactTypeB fb, FactTypeC fc){
   String status = "";
   synchronized(ruleSessionLock){
     try {
       ruleSession.callFunctionWithArgument("assert", fa);
       ruleSession.callFunctionWithArgument("assert", fb);
       status = "success";
     } catch (Exception e) {
       // a method that creates a new RuleSession loads it with rules
       initializeRuleSession();
       status = "failure";
     }
     return status;
}
```

Sample RuleSession Consumer Code

Example C-3 shows the consumer code.

Example C-3 RuleSession Consumer Code

```
public List exec(){
  synchronized(ruleSessionLock){
    try {
      ruleSession.callFunction("run");
      List results = (List)ruleSession.callFunction("getResults");
      ruleSession.callFunction("clearResults");
      return results;
    }  catch (Exception e) {
      // a method that creates a new RuleSession loads it with rules
      initializeRuleSession();
      return null;
    }
  }
}
```

Note:

When multiple threads are sharing a `RuleSession` object, if more than one of the threads calls the `run` method, this can create extremely hard to find bugs and there is usually no gain in performance.

How Do I Correctly Express a Self-Join?

When working with facts, there are cases where the runtime behavior of Oracle RL may produce surprising results.

Consider the Oracle RL code in the following self-join example:

```
class F {int i; };
rule r1 {
  if (fact F f1 && fact F f2) {
    println("Results: " + f1.i + ", " + f2.i);
  }
}
assert(new F(i:1));
assert(new F(i:2));
run();
```

How many lines print in the above example output? The answer is 4 lines because the same fact instance can match for both `f1` and `f2`.

Thus, the example gives the following output:

```
Results: 2, 2
Results: 2, 1
Results: 1, 2
Results: 1, 1
```

Using the same example with a third `F`, for example (`assert(new F(i:3));`) then nine lines are printed and if, at the same time, a third term `&& fact F F3` is added then 27 lines are printed.

Sample Find All Combinations of Fact F

If you are attempting to find all combinations and orders of distinct facts, you need an additional term to in the test, as shown in Example C-4.

Example C-4 Find All Combinations of Fact F

```
rule r1 {
  if (fact F F1 && fact F F2 && F1 != F2) {
    println("Results: " + F1.i + ", " + F2.i);
  }
}
```

The above code gives the following output:

```
Results: 2, 1
Results: 1, 2
```

Sample Finding Combinations of Fact F

The simplest, although not the fastest way to find all combinations of facts, regardless of their order, is to use the code shown in Example C-5.

Example C-5 Finding Combinations of Fact F

```
rule r1 {
  if (fact F F1 && fact F F2 && id(F1) < id(F2)) {
```

```
    println("Results: " + F1.i + ", " + F2.i);
  }
}
```

Sample Fast Complete Comparison

The function `id()` shown in Example C-5 takes longer to execute in a test pattern than a direct comparison, the fastest method is to test on a unique value in each object. For example, you could add an integer value property "oid" to your class that is assigned a unique value for each instance of the class.

Example C-6 shows the same rule using the oid value.

Example C-6 Fast Complete Comparison

```
rule r1 {
  if (fact F F1 && fact F F2 && F1.oid < F2.oid) {
    println("Results: " + F1.i + ", " + F2.i);
  }
}
```

This problem may also arise if you attempt to remove all duplicate facts from the Oracle Rules Engine, using a function as shown below:

```
rule rRemoveDups {
  if (fact F F1 && fact F F2 && F1.i == F2.i) {
    retract(F2);
  }
}
```

However, this rule removes all facts of type `F`, not just the duplicates because `F1` and `F2` may be the same fact instance. The following example shows the correct version of this rule:

```
rule rRemoveDups {
  if (fact F F1 && fact F F2 && F1 != F2 && F1.i == F2.i) {
    retract(F2);
  }
}
```

How Do I Use a Property Change Listener in Oracle Business Rules?

The Oracle Rules Engine supports the Java `PropertyChangeListener` design pattern. This allows an instance of a Java fact that uses the `PropertyChangeSupport` class to automatically notify the Oracle Rules Engine when property values have changed. Java facts are not required to implement this pattern to be used by Oracle Rules Engine.

Typically, changes made to values of a property of a Java object that has previously been asserted to the Oracle Rules Engine requires that the object be re-asserted in order for rules to be reevaluated with the new property value. For properties that fire `PropertyChangeEvent`, changing the value of those properties both changes the value and re-asserts the fact to the Oracle Rules Engine.

To implement the `PropertyChangeListener` design pattern in a class, do the following:

1. Import this package in the class:

   ```
   import java.beans.PropertyChangeSupport;
   ```

2. Add a private member variable to the class:

   ```
   private PropertyChangeSupport m_pcs = null;
   ```

3. In the constructor, create a new `PropertyChangeSupport` object:

```
m_pcs = new PropertyChangeSupport(this);
```

4. Then for each setter, add the call to `firePropertyChange`:

```
public void setName( String name ){
    String oldVal =  m_name;
    m_name = name;
    m_pcs.firePropertyChange( "name", oldVal, m_name );
}
```

5. Implement `addPropertyChangeListener` method (delegate to `m_pcs`):

```
public void addPropertyChangeListener(PropertyChangeListener pcl){
    m_pcs.addPropertyChangeListener( pcl );
}
```

6. Implement `removePropertyChangeListener` method (delegate to `m_pcs`):

```
public removePropertyChangeListener(PropertyChangeListener pcl){
    m_pcs.removePropertyChangeListener( pcl );
}
```

When deciding whether to design your application to always explicitly re-assert modified objects or implement the `PropertyChangeListener` design pattern, consider the following:

- Explicitly re-asserting modified objects allows a user to group several property changes and making them visible to the rules all at once. This is most useful when a concurrent thread is executing rules, and the rules should see only a complete group of property changes.
- Explicit assert reduces the computational cost of rule re-evaluation when multiple properties are changed. If multiple properties are changed at the same time, this results in multiple re-evaluations of rule conditions that reference the fact type. This occurs because each property change event results in a re-assertion of the object. Using an explicit assert instead of the `PropertyChangeListener` pattern eliminates this extra computational cost.
- Explicit assert is required when a rule modifies a fact that is also tested in its condition, but the automatic reassert triggered by the `PropertyChangeListener` before a guard condition property is set would cause the rule to refire itself endlessly.
- Explicit assert must be used when modifying Oracle RL facts and XML facts, because these cannot be defined to support the `PropertyChangeListener` design pattern.
- `PropertyChangeListener`-enabled facts allow a Java application to communicate property changes to the rule engine without having to change the application to perform explicit asserts. This also means that code that modifies a property of an object does not need to have a reference to the `RuleSession` object in scope.
- `PropertyChangeListener` support prevents the common error of neglecting to re-assert a fact after changing its properties.

What Are the Limitations on a Decision Service with Oracle Business Rules?

There are some limitations for using Business Rules with a BPEL process.

Some of the limitations include the following:

- Only visible XML fact types may be specified as the input for a decision service.
- Only visible XML fact types may be specified as the output of a decision service.

For an additional restriction, see How Are Decision Service Input Output Element Types Restricted?.

For information on setting XML fact type visible option, see Working with XML Facts.

How Do I Put Java Code in a Rule?

You do not actually put Java code in a rule.

However, you can invoke a Java method from a rule condition or action.

Can I Use Java Based Facts in a Decision Service with BPEL?

Oracle BPEL PM can invoke only decision functions exposed as a decision service, and this means that the decision function inputs and outputs must be XML fact types.

You can use an existing ruleset or decision function that uses Java fact types if you convert the input XML facts to Java facts. For example, you could create some rules in a ruleset, named `convertFromXML`, and put this ruleset before the Java ruleset in the decision function ruleflow. Similarly, you could create a ruleset to convert from Java facts to output XML facts and put this ruleset after the Java ruleset in the decision function ruleflow.

Alternatively, if your rules use only properties, and no methods or fields, from the Java fact types you can replace the Java fact types with XML fact types as follows:

1. Delete the Java fact types (first making careful note of the aliases of the fact types and properties).
2. Import similar XML fact types and edit the aliases of the fact types and properties to be the same as the deleted Java fact types and properties.

How Do I Enable Debugging in a BPEL Decision Service?

To enable debugging output during ruleset execution for a BPEL Decision Service, you enable the SOA rules logger. When the SOA rules logger is set to `TRACE` level then the output of `watchAll` is logged to the SOA diagnostic log. When you change the logging level using Fusion Middleware Control Console, you do not need to redeploy the application to use the specified level.

For information on using the SOA oracle.soa.service.rules and oracle.soa.services.rules.obrtrace loggers, see *Administering Oracle SOA Suite and Oracle Business Process Management Suite*.

How Do I Support Versioning with Oracle Business Rules?

Versioning is supported in Oracle Business Rules.

The two possible ways are:

- At design time, the dictionary is stored as an XML file in a JDeveloper project. The dictionary can be versioned in a source control system in the same way as any other source file.
- At runtime, the dictionary is stored in MDS. If MDS is database backed then versioning is supported using MDS.

Note: It is possible for a server application to respond to dictionary changes as they are made visible to the application in MDS. The rule service engine (decision service) does this automatically. For non-SCA application, this can be done using the RuleRepository interface. At this time, they way to support an "in-draft" version is by using the sandbox feature of MDS. The Oracle Business Rules RuleRepository interface supports this.

What is the Priority Order Using Priorities with Rules and Decision Tables?

The priority for rules and decision tables is highest to lowest, with the higher priority rule or Decision Table executing first. For example, if you create rules with priorities 1-4, they would be executed in the execution priority order 4,3,2,1.

Using Rules Designer you can select a priority from a predefined named priority list or enter a positive or negative integer to specify your own priority level. The default priority is `medium` (with the integer value 0). For more information, see How to Set a Priority for a Rule.

Note, however, you should try to avoid priorities as much as possible since they break the purely declarative model of rules. If you find yourself using a lot of priorities, then generally it is best to try to restructure your rule patterns and tests to avoid conflicts, or divide the rules into multiple rulesets using ruleflow if they are intended to be run in a certain order. A conflict is a case when more than one rule in a ruleset is able to fire. For example, if a "gold customer" rule says to make a customer that spends over $1000 a gold customer, and a "silver customer" rule says to make a customer that spends over $500 a silver customer, then when a customer spends $1100 there is a conflict. Rather than prioritize the rules, it is more declarative to change the "silver customer" rule to test for customers that spend *between* $500 and $1000. This conflict analysis and conflict avoidance is particularly easy if you use Decision Tables. For more information on Decision Tables, see Working with Decision Tables.

You use ruleflow, that is the ruleset stack, to order rulesets. For information on working with the ruleset stack, see *Rules Language Reference for Oracle Business Process Management*.

Why do XML Schema with xsd:string Typed Elements Import as Type JAXBElement?

According to the JAXB 2.0 spec, the default type mapping for elements that have `minOccurs="0"` and `nillable="true"` is `JAXBElement<T>`, where *T* is the default mapping of the type defined for the element. For example, `xsd:string` maps to `JAXBElement<String>`, `xsd:int` maps to `JAXBElement<Integer>`, and `xsd:integer` maps to `JAXBElement<BigInteger>`.

This is because `nillable="true"` means the user has defined a semantic difference between a element not being defined in a document, with `minOccurs=0`, it does not have to be defined, and an element being defined but having the attribute `nil="true"`. This is a subtle difference and is often used to define the difference between an unknown value and a value known to be "no value".

To use the JAXBElement-typed property in a rule, the property must be first checked for non-null, and then the "value" property or `getValue()` method can be used retrieve a value of the underlying type:

```
fact FactType1 &&
    FactType1.prop1 != null &&
    FactType1.prop1.value == "abc"
```

Alternatively, you may want to define a customized JAXB binding so nillable elements are mapped to type *T* rather than `JAXBElement<`*T*`>`. However, this is a lossy conversion, as you no longer are able to determine the difference between a non-existent element and a nil one. This does make the nillable attribute less useful, but it does allow you to explicitly define an element as nil in your document, similarly to how in Java an Object-typed field is initialized to null by default or you can explicitly initialize it to null.

There are several ways to do this. In both cases, add these attributes to the top-level `xsd:schema` element start tag:

```
xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
jaxb:version="2.0"
```

1. To specify ALL properties to use the binding, add this immediately inside the xsd:schema opening tag:

   ```
   <xsd:annotation>
       <xsd:appinfo>
           <jaxb:globalBindings generateElementProperty="false"/>
       </xsd:appinfo>
   </xsd:annotation>
   ```

2. To specify only specific properties use the binding, add an annotation like this to each desired element:

   ```
   <xsd:element name="stringElement2" type="xsd:string" minOccurs="0"
   nillable="true">
       <xsd:annotation>
           <xsd:appinfo>
                <jaxb:property generateElementProperty="false" />
           </xsd:appinfo>
       </xsd:annotation>
   </xsd:element>
   ```

3. Add the definitions to an external customizations file and pass it as an argument when adding the schema to the datamodel. This can only be done when programmatically calling the SchemaBrowser class and is not exposed in Rule Designer.

Why Are Changes to My Java Classes Not Reflected in the Data Model?

Do not import classes that have been compiled into the "SCA-INF/classes" directory.

Classes in this directory cannot be reloaded into the data model when they change.

How Do I Use Rules SDK to Include a null in an Expression?

You can use Rules SDK code to include a null value.

You can use the following Rules SDK code:

```
SimpleTest test = pattern.getSimpleTestTable().add();
```

```
test.getLeft().setValue(attr);
test.setOperator(Util.TESTOP_NE);
test.getRight().setValue("null");
```

Is WebDAV Supported as a Repository to Store a Dictionary?

The Web Distributed Authoring and Versioning (WebDAV) repository is not supported to store a dictionary in Oracle Fusion Middleware 11*g* Release 1 (11.1.1) Oracle Business Rules. Oracle Business Rules supports using an MDS (file backed or Database backed) repository for storing dictionaries.

Using a Source Code Control System with Rules Designer

There are special considerations when you use Rules Designer and a source control system, such as CVS or Subversion. When you use a source code control system with Rules Designer you need to specify that rule dictionary files in your project are recognized as "binary" files instead of "text" files. The rule dictionary files are XML documents and by default the source code control system treats these files as text files.

However, rule dictionary files cannot be merged because the files contain semantic structure. If a rule dictionary file is treated as a text file and then changed, the source control system attempts to merge the file with a "trivial" merge. Using a trivial merge creates a semantically invalid dictionary file which cannot be unmarshalled into a RuleDictionary object.

Thus, when you use a source code control system with rule dictionary files, .rules files, you need to make sure the source code control system treats the files as binary files. There are configuration options you need to set to specify that the system treats dictionary files as binary files. For example, in the Subversion source code control system you can set the MIME type with the `svn:mime-type` file property. For more information, see

`http://svnbook.red-bean.com/nightly/en/svn.advanced.props.file-portability.html#svn.advanced.props.special.mime-type`

When you set the source code control system options to specify the binary file type, this allows the source code control system, for example tortoiseSVN, to treat the rules dictionary files correctly, as binary files.

D

Oracle Business Rules Troubleshooting

This appendix contains workarounds and solutions for issues you may encounter when using Oracle Business Rules.

This appendix includes the following sections:

Getter and Setter Methods are not Visible

Rules Designer does not list the methods supporting a Java bean property in choice lists; only the bean properties are visible. For example, a Java bean with a property named `Y` must have at least a getter method `getY()` and may also have a setter method `setY(y-type-param)`.

All of properties and methods (including getter and setter that compose the properties) are displayed when viewing the Java FactType. Only the properties of Java Classes (not the getter and setter methods) are displayed in choice lists. When attempting to control the visibility of the property it is best to use the properties visibility flag. Marking a getter or a setter method as not visible may not remove the properties from choice lists.

Java Class with Only a Property Setter

In Java the Java Bean introspector includes write-only properties. Oracle RL does not include such properties as Beans, because they cannot be reasoned on in a rule.

Thus, in order for Java fact type bean properties to be properly accessed in Oracle RL they must have both a getter and setter. Properties which have a setter but not a getter, that is write-only properties, are not allowed in the Oracle RL "new" syntax.

For example, if a bean `Foo` only has the method `setProp1(int i)`, then you cannot use the following in Oracle RL:

```
Foo f = new Foo(prop1: 0)
```

Runtime NoClassDefFound Error

Sometimes when working with XML facts, you might receive an error.

You may receive an error of the form:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
```

The `java.lang.NoClassDefFoundError` is very likely due to required classes not in classpath. Try checking the following:

- Add `xml.jar` to your classpath when executing.
- Add the directory where the generated and compiled JAXB classes are placed to the classpath.

RL Specific Keyword Naming Conflict Errors

Oracle Business Rules escapes RL specific keywords when generating RL from Rules Designer.

In most cases, RL specific keywords can be used without causing errors. One exception is using a keyword as the name of a class. This is unlikely for Java classes because by convention they start with an upper case letter and RL specific keywords are all lowercase..

java.lang.IllegalAccessError from Business Rules Service Runtime

There may be errors

Problem: I receive an error such as the following:

```
java.lang.IllegalAccessError: tried to access class
com.sun.xml.bind.v2.runtime.reflect.opt.Const from class:...
```

Reason: This can be due to JAXB 2.1.6 issue 490, caused when unmarshalling incorrect, for example letter characters when float is expected, data as described at the following site,

http://java.net/jira/browse/JAXB-490

Workaround: the workaround for this problem is to assign a value to the appropriate element, as shown in Figure D-1 and Figure D-2 where `approvalRequired` is assigned a default value `false()`.

Note that the screen shots reflect a previous version, however, the content is applicable to the current release.

Figure D-1 Adding an Expression to Initialize a Value for a Business Rules Service Input

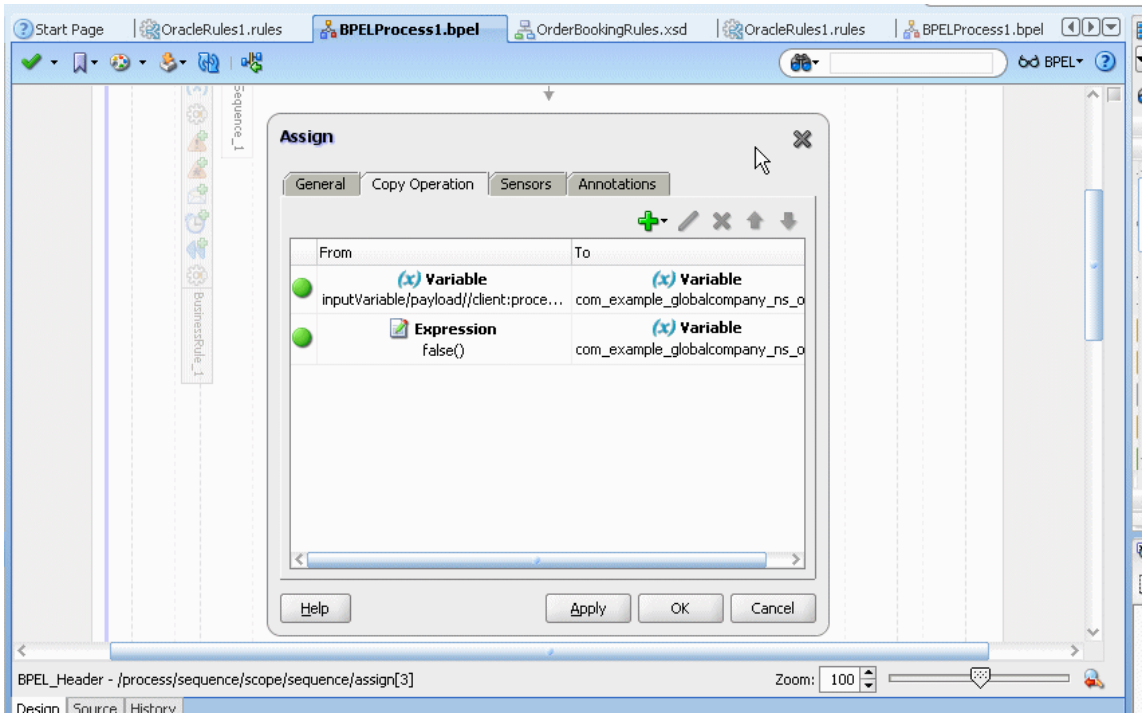
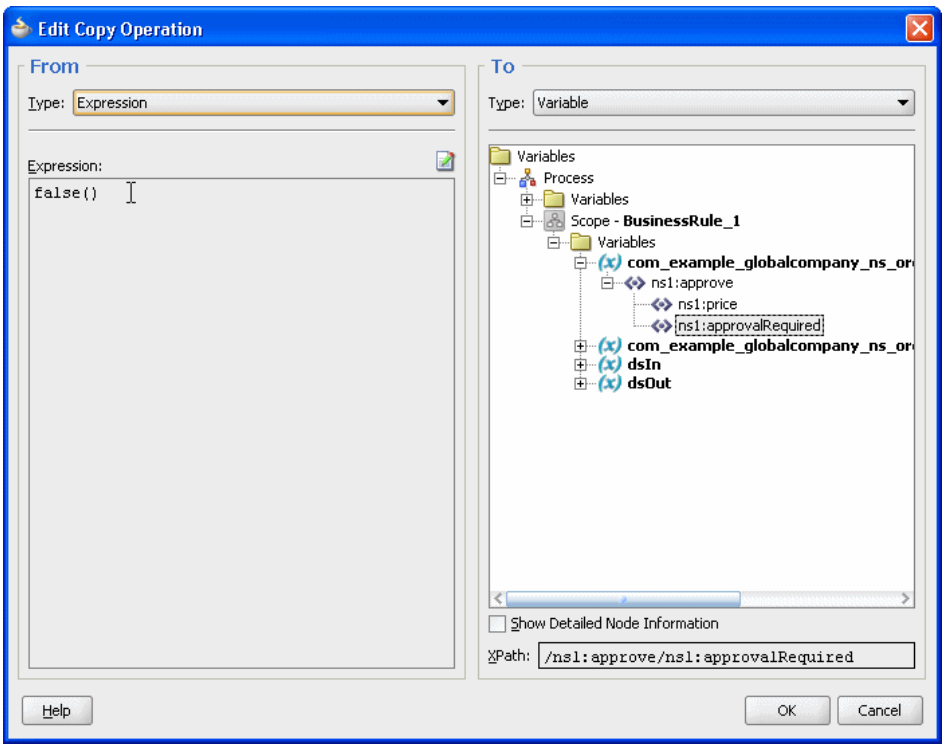


Figure D-2 Expression Assigned to Input Variable for Business Rules Service



JAXB 1.0 Dictionaries and RL MultipleInheritanceException

Dictionaries which have been migrated from 10.1.3 use JAXB 1.0 instead of JAXB 2.0, which is the default for Oracle Fusion Middleware 11*g* Release 1 (11.1.1) dictionaries.

Because of this use of JAXB 1.0, the migrated dictionaries may contain Element types. If your dictionary has Element types marked as visible, generated RL may throw `MultipleInheritanceException`.

The solution to this issue is to mark the Element fact types non-visible or remove them from the datamodel. Only the Type classes generated by JAXB should be used to write rules, so there is no functionality loss from deleting the Element types.

Why Does XML Schema with Underscores Fail JAXB Compilation?

The defined behavior of JAXB is to fail when a name of the form `'_'` + number is found. In this case JAXB cannot generate an "obvious" Java class name from this string. The default behavior of JAXB for `'_'` + char is to treat it as a word boundary (`underscoreBinding="asWordSeparator"`), which means the underscore is stripped and the char is UpperCamelCased. For example, `_fooBar` is mapped to `FooBar`.

To fix this problem, you need to provide a schema customization to direct JAXB to generate the names differently. The default value for `underscoreBinding` is specified as `"asWordSeparator"`, which does not allow an underscore to be used at the beginning of a name.

The global annotation `underscoreBinding="asCharInWord"` causes the `'_'` to be preserved in the classname and UpperCamelCase after the number:

```
<xsd:annotation><xsd:appinfo>
      <jaxb:globalBindings underscoreBinding="asCharInWord" />
</xsd:appinfo></xsd:annotation>
```

With this global annotation, the mapping for `_1foo_bar_baz` is `_1Foo_Bar_Baz`.

How Are Decision Service Input Output Element Types Restricted?

Using the Decision Service to run business rules with XML schema defining the input, for any given `complexType` "*tFoo*" in an XML-Schema file `Foo.xsd` there can only be one XML-Schema element "`foo`" of type "*tFoo*".

The Decision Service does not allow you to use two elements "`foo`" and "`bar`" of the same type "*tFoo*.

How Are Decision Service Input Output Schema Restricted?

When you use the Decision Service a schema must define a `complexType` or import another schema which defines a `complexType`.

You cannot use schemas which do not define `complexType`, such as the following:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="http://www.example.org"
            targetNamespace="http://www.example.org"
            elementFormDefault="qualified">
  <xsd:element name="count" type="xsd:int"/>
</xsd:schema>
```

How Do I Handle Java Reserved Names in an Imported Fact Type?

In Oracle Business Rules, when you import fact type properties which would have the same name as a Java language reserved word are excluded.

For a complete list of Java reserved words, see

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html

A workaround is to rename the getter and setter method pair that produce the excluded property. If these methods names cannot be changed, the methods should be used in rules instead of the properties.

For example, if a property named `continue` is excluded, you can create rules that use `getContinue()` and `setContinue()` methods instead of using the property.You can do this by rewriting a pattern. For example, replace:

```
fact IncrCount ic && ic.continue == "foo"
```

with:

```
fact IncrCount ic && ic.getContinue() == "foo"
```

For another example, in an action, replace:

```
[assert new] IncrCount(continue:"bar")
```

with:

```
[assign new] IncrCount ic = new IncrCount( )
[call] ic.setContinue("bar")
[assert] ic
```

E

Working with Oracle Business Rules and JSR-94 Execution Sets

This appendix describes the Java Rule Engine API (JSR-94) specification that defines a standard Java runtime API to access a rule engine from a Java SE or Java EE client.

The appendix includes the following sections:

- Introduction to Oracle Business Rules and JSR-94 Execution Sets
- Creating JSR-94 Rule Execution Sets from Oracle Business Rules Rulesets
- Using the JSR-94 Interface with Oracle Business Rules

For more information, see:

- `http://jcp.org/en/jsr/detail?id=94`
- `http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html`

Introduction to Oracle Business Rules and JSR-94 Execution Sets

Oracle Business Rules provides JSR-94 support. This allows you to create more portable rule-enabled applications.

You can create JSR-94 execution sets from Oracle Business Rules rulesets and you can create JSR-94 rule sessions from these execution sets. For more information, see Creating JSR-94 Rule Execution Sets from Oracle Business Rules Rulesets.

You can access Oracle Business Rules rulesets and execute them against the Oracle Business Rules Engine using the JSR-94 API. For more information, see Using the JSR-94 Interface with Oracle Business Rules.

Oracle Business Rules also provides extensions to the JSR-94 API that you may find useful. For more information, see Using Oracle Business Rules JSR-94 Extensions.

Creating JSR-94 Rule Execution Sets from Oracle Business Rules Rulesets

To use JSR-94 with rules in RL Language text, you must map the rules to a JSR-94 rule execution set.

A JSR-94 rule execution set (rule execution set) is a collection of rules that are intended to be executed together. You also must register a rule execution set before running it. A registration associates a rule execution set with a URI; using the URI, you can create a JSR-94 rule session.

Note:

In Oracle Business Rules, a JSR-94 rule execution set registration is not persistent. Thus, you must register a rule execution set programmatically using a JSR-94 `RuleExecutionSetProvider` interface.

For more information, see Creating a Rule Execution Set with createRuleExecutionSet.

Creating Rule Execution Set with Oracle Business Rules RL Language Text

You can use JSR-94 with RL Language rulesets saved as text, where the Oracle RL text is directly included in the rule execution set. For more information, see Using the Extended createRuleExecutionSet to Create a Rule Execution Set for information about JSR-94 extensions that assist you in including RL Language text.

To create a rule execution set from Oracle Business Rules Oracle RL language text:

1. Specify the RL Language mapping information in an XML document. Table E-1 shows the mapping elements required to construct a rule execution set. The following code example shows a sample XML document for mapping RL Language text to a JSR-94 rule execution set.

```
<rule-execution-set xmlns="http://xmlns.oracle.com/rules/jsr94/configuration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
    <name>CarRentalDemo</name>
    <description>The Car Rental Demo</description>
    <rule-source>
        <rl-text>
         ruleset DM {
            fact class carrental.Driver {
               hide property ableToDrive, driverLicNum, licIssueDate, licenceType,
               llicIssueDate, numPreAccidents, numPreConvictions,
               numYearsSinceLicIssued, vehicleType;
            };

               final String DeclineMessage = &amp;quot;Rental declined &amp;quot;;;

               public class Decision  supports xpath {
                  public String driverName;
                  public String type;
                  public String message;
              }

               function assertXPath(String package,
                                    java.lang.Object element, String xpath) {
                  //RL literal statement
                  main.assertXPath( package, element, xpath );
               }

               function println(String message) {
                  //RL literal statement
                  main.println(message);
               }

               function showDecision(DM.Decision decision) {
                  //RL literal statement
                  DM.println( &amp;quot;Rental decision is &amp;quot; + decision.type +
                              &amp;quot; for driver &amp;quot; + decision.driverName +
                              &amp;quot; for reason &amp;quot; + decision.message);
               }
         }
        </rl-text>
```

```
    </rule-source>
    <rule-source>
        <rl-text>
            ruleset vehicleRent {
                rule UnderAge {
                    priority = 0;
                    if ((fact carrental.Driver v0_Driver &amp;amp;&amp;amp;
                        (v0_Driver.age &amp;lt; 19))) {
                            DM.println( &amp;quot;Rental declined: &amp;quot; + v0_Driver.name +
                            &amp;quot; Under age, age is: &amp;quot; + v0_Driver.age);
                            retract(v0_Driver);
                    }
                }
            }
        </rl-text>
    </rule-source>
    <ruleset-stack>
        <ruleset-name>vehicleRent</ruleset-name>
    </ruleset-stack>
</rule-execution-set>
```

2. You then use the XML document with the JSR-94 administration APIs to create a rule execution set. The resulting rule execution set is registered with a JSR-94 runtime (using a `RuleAdministration` instance).

Table E-1 Oracle Business Rules Oracle RL Language Text XML Mapping Elements for JSR-94

Element	Description
`<rule-source>`	Includes an `<rl-text>` tag containing explicit RL Language text containing an Oracle Business Rules ruleset. Multiple `<rule-source>` tags can be used to specify multiple rulesets (specified in the order in which they are interpreted).
`<ruleset-stack>`	Specifies a list of rulesets that form the initial ruleset stack. The order of the rulesets in the list is from the top of the stack to the bottom of the stack.

Note:

In the `<rl-text>` element the contents must escape XML predefined entities. This includes the characters '&', '>', '<', "", and '\''.

Creating a Rule Execution Set from Oracle RL Text Specified in a URL

You can use JSR-94 with Oracle RL rulesets specified using a URL. For more information, see Using the Extended createRuleExecutionSet to Create a Rule Execution Set for information about JSR-94 extensions that assist you in specifying a URL.

To create a rule execution set from Oracle RL text specified in a URL:

1. Specify the Oracle RL mapping information in an XML document. Table E-2 shows the mapping elements required to construct a rule execution set. The following code example shows a sample XML document for mapping Oracle RL text to a JSR-94 rule execution set.

```
<?xml version="1.0" encoding="UTF-8"?>
<rule-execution-set xmlns="http://xmlns.oracle.com/rules/jsr94/configuration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
    <name>CarRentalDemo</name>
    <description>The Car Rental Demo</description>
    <rule-source>
        <rl-url>
            file:rl/DM.r1
        </rl-url>
    </rule-source>
    <rule-source>
        <rl-url>
            file:r1/VehicleRent.r1
        </rl-url>
    </rule-source>
    <ruleset-stack>
        <ruleset-name>vehicleRent</ruleset-name>
    </ruleset-stack>
</rule-execution-set>
```

2. You then use the XML document with the JSR-94 administration APIs to create a rule execution set. The resulting rule execution set is registered with a JSR-94 runtime (using a `RuleAdministration` instance).

Table E-2 Oracle Business Rules Oracle RL URL XML Mapping Elements for JSR-94

Element	Description
`<rule-source>`	Includes an `<rl-url>` tag containing a URL that specifies the location of RL Language text. Multiple `<rule-source>` tags can be used to specify multiple rulesets (in the order in which they are interpreted).
`<ruleset-stack>`	Specifies a list of rulesets that form the initial ruleset stack. The order of the rulesets in the list is from the top of the stack to the bottom of the stack.

Creating Rule Execution Sets with Rulesets from Multiple Sources

A rule execution set may contain rules that are derived from multiple sources and the sources may be a mix of Rules Designer defined rulesets and RL Language rulesets. In this case, the XML element `<rule-execution-set>` set contains multiple `<rule-source>` elements, one for each source of rules. You must list each `<rule-source>` in the order in which they are to be interpreted in Rules Engine.

Note:

For this Oracle Business Rules release, a JSR-94 rule execution set can only reference one Rules Designer dictionary.

Using the JSR-94 Interface with Oracle Business Rules

This section describes the Oracle Business Rules specific details for JSR-94 interfaces.

This section contains the following sections:

- Creating a Rule Execution Set with createRuleExecutionSet

- Creating a Rule Session with createRuleSession
- Working with JSR-94 Metadata
- Using Oracle Business Rules JSR-94 Extensions

Creating a Rule Execution Set with createRuleExecutionSet

The `RuleExecutionSetProvider` and `LocalRuleExecutionSetProvider` interfaces in `javax.rules.admin` include the `createRuleExecutionSet` to create a `RuleExecutionSet` object.

For the remaining `createRuleExecutionSet` methods, the first argument is interpreted as shown in Table E-3.

Table E-3 First Argument Types for createRuleExecutionSet Method

Argument	Description
`org.w3c.dom.Element`	Specifies an instance of the `<rule-execution-set>` element from the configuration schema.
`java.lang.String`	Specifies a URL that specifies the location of an XML document that is an instance of the `<rule-execution-set>` element from the configuration schema.
`java.io.InputStream`	Specifies an input stream that is used to read an XML document that is an instance of the `<rule-execution-set>` element from the configuration schema.
`java.io.Reader`	Specifies a character reader that is used to read an XML document that is an instance of the `<rule-execution-set>` element from the configuration schema.

Note:

JSR-94 also includes `createRuleExecutionSet` methods that take a `java.lang.Object` argument, which is intended to be an abstract syntax tree for the rule execution set. In Oracle Business Rules for Oracle Fusion Middleware 11*g* Release 1 (11.1.1), using these methods with this argument is not supported. Invoking these methods with a `java.lang.Object` argument gives a `RuleExecutionSetCreateException` exception.

The second argument to the `createRuleExecutionSet` methods is a `java.util.Map` of vendor-specific properties.

Creating a Rule Session with createRuleSession

Clients create a JSR-94 rule session using the `createRuleSession` method in the `RuleRuntime` class. This method takes a `java.util.Map` argument of vendor-specific properties. This argument can be used to pass in any of the properties defined for the Oracle Business Rules `oracle.rules.rl.RuleSession`.

If a rule execution set contains URL or repository rule sources, the rules from those sources are fetched on the creation of each new `RuleSession`.

Working with JSR-94 Metadata

JSR-94 allows for metadata for rule execution sets and rules within a rule execution set. The Oracle Business Rules implementation does not add any additional metadata beyond what is in the JSR-94 specification.

The rule execution set description is an optional item and thus may not be present. If it is not present, the empty string is returned. For rules, only the rule name is available and the description is initialized with an empty string.

Using Oracle Business Rules JSR-94 Extensions

This section covers the extensions provided in the JSR-94 implementation classes.

This section covers the following extensions:

- Using the Extended createRuleExecutionSet to Create a Rule Execution Set
- Invoking an RL Language Function in JSR-94

Using the Extended createRuleExecutionSet to Create a Rule Execution Set

Oracle Business Rules provides a helper function to facilitate creating the XML control file required as input to create a `RuleExecutionSet`.

The helper method `createRuleExecutionSet` is available in the `RLLocalRuleExecutionSetProvider` class. The `createRuleExecutionSet` method has the following signature:

```
RuleExecutionSet createRuleExecutionSet(String name,
                                        String description,
                                        RuleSource[] sources,
                                        String[] rulesetStack,
                                        Map properties)
```

Table E-4 describes the `createRuleExecutionSet` arguments.

Table E-4 createRuleExecutionSet Arguments

Argument	Description
`name`	Specifies the name of the rule execution set.
`description`	Specifies the description of the rule execution set.
`sources`	Specifies an array of specifications for the sources of rules. The `RuleSource` is an interface that the following classes implement: • RLTextSource: RL Language text for RL Language text. • `RLUrlSource`: RL Language URL for a URL to RL Language text. For more information, see the `oracle.rules.jsr94.admin` package in *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*.
`rulesetstack`	Specifies the initial contents of the RL Language ruleset stack to be set before each time the rules are executed. The contents of the array should be ordered from the top of stack (0th element) to the bottom of stack (last element).
`properties`	Oracle specific properties.

Invoking an RL Language Function in JSR-94

In a stateful interaction with a JSR-94 rule session, a user may want the ability to invoke an arbitrary RL Language function. The class that implements the JSR-94 `StatefulRuleSession` interface provides access to the `callFunction` methods in the `oracle.rules.rl.RuleSession` class.

The following example shows how you can invoke an RL Language function with no arguments in a JSR-94 `StatefulRuleSession`.

```
import javax.rules.*;
...
StatefulRuleSession session;
...
((oracle.rules.jsr94.RLStatefulRuleSession) session).callFunction("myFunction");
```