Oracle® Fusion Middleware

User's Guide for Oracle Business Rules

11*g* Release 1 (11.1.1)

E10228-01

May 2009

ORACLE®

Oracle Fusion Middleware User's Guide for Oracle Business Rules 11*g* Release 1 (11.1.1)

E10228-01

Primary Author: Thomas Van Raalte

Contributing Author: Peter Purich

Contributors: Chris Cowell-Shah, Ching Luan Chung, Kathryn Gruenefeldt, Gary Hallmark, Joe Rosinski, Phil Varner, Neal Wyse

Contents

Preface

This Preface contains these topics:

- Audience
- Documentation Accessibility
- Related Documentation
- Conventions

Audience

Oracle Fusion Middleware User's Guide for Oracle Business Rules 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

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Related Documentation

Printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://www.oracle.com/technology/membership/index.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://www.oracle.com/technology/documentation/index.html`

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.

1 Overview of Oracle Business Rules

Oracle Business Rules enable dynamic decisions at runtime allowing you to automate policies, computations, and reasoning while separating rule logic from underlying application code. This allows more agile rule maintenance and empowers business analysts with the ability to modify rule logic without programmer assistance and without interrupting business processes.

This guide describes how to:

- Work with the Oracle Business Rules Designer (Rules Designer) extension to Oracle JDeveloper to create Oracle Business Rules artifacts
- Use Oracle Business Rules as part of an Oracle SOA Suite composite application as a Decision component
- Use Oracle Business Rules as part of Java EE application with the Oracle Business Rules SDK
- Use the Oracle Business Rules SDK (Rules SDK)
- Access the Oracle Business Rules Rules Engine using the JSR-94 Java Rule Engine API

This chapter includes the following sections:

- Section 1.1, "What are Business Rules?"
- Section 1.2, "Oracle Business Rules Runtime and Design Time Elements"
- Section 1.3, "Oracle Business Rules Engine Architecture"

For more information, see:

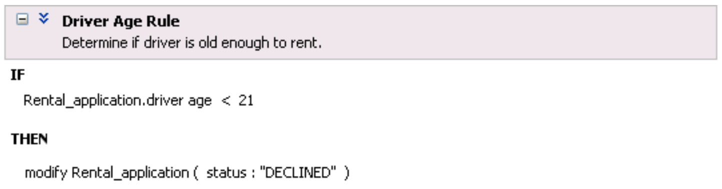
- *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*
- *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*
- *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*

1.1 What are Business Rules?

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

- Business policies such as spending policies and approval matrices.
- Constraints such as valid configurations or regulatory requirements.
- Computations such as discounts or premiums.
- Reasoning capabilities such as offers based on customer value.

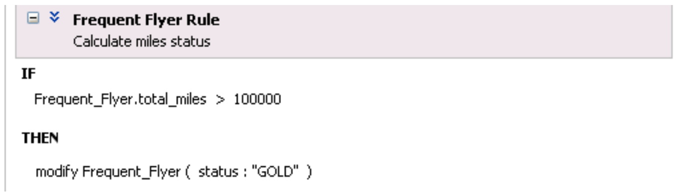
For example, a car rental company might use the following business rule:



Driver Age Rule
Determine if driver is old enough to rent.

```
IF
  Rental_application.driver age  <  21
THEN
  modify Rental_application (  status : "DECLINED"  )
```

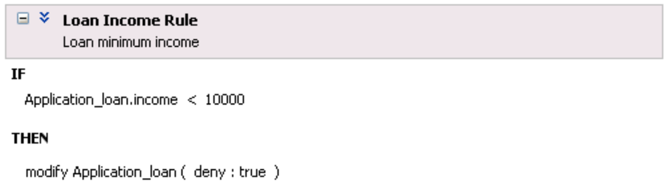
An airline might use a business rule such as the following:



Frequent Flyer Rule
Calculate miles status

```
IF
  Frequent_Flyer.total_miles  >  100000
THEN
  modify Frequent_Flyer (  status : "GOLD"  )
```

A financial institution could use a business rule such as:



Loan Income Rule
Loan minimum income

```
IF
  Application_loan.income  <  10000
THEN
  modify Application_loan (  deny : true  )
```

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 is not easy for nontechnical users to read and can be difficult 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. Using Oracle Business Rules, this process can be simplified because a business rules application is built to support easily changing business rules.

Oracle Business Rules allows a business analyst to 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.

A rule follows an if-then structure and consists of the following parts:

- **IF** part: a condition or pattern match (see Section 1.1.1, "What Are Rule Conditions?").
- **THEN** part: a list of actions (see Section 1.1.2, "What Are Rule Actions?").

Alternatively, you can express rules in a spreadsheet-like format called a Decision Table (see Section 1.1.3, "What Are Decision Tables?").

You write rules and Decision Tables in terms of fact types and properties. 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 bucketset. A bucketset splits the value space of the data type into buckets that can be used in Decision Tables, choice lists, and for design time validation (see Section 1.1.4, "What Are Facts and Bucketsets?").

You group rules and Decision Tables in an Oracle Business Rules object called a ruleset (see Section 1.1.5, "What Are Rulesets?").

You group one or more rulesets and their facts and bucketsets in an Oracle Business Rules object called a dictionary (see Section 1.1.8, "What Are Dictionaries?").

For more information, see Section 1.2, "Oracle Business Rules Runtime and Design Time Elements".

1.1.1 What Are Rule Conditions?

The rule **IF** part is composed of conditional expressions, rule conditions, that refer to facts. For example:

```
IF Rental_application.driver age < 21
```

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

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:

- Chapter 4, "Working with Rulesets and Rules"
- "Rule Conditions" in the *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*

1.1.2 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 change the next rule to fire.

For more information, see:

- Section 1.3.4, "Rule Firing and Rule Sessions"
- Chapter 4, "Working with Rulesets and Rules"

- "Ordering Rule Firing" in the *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*

1.1.3 What Are 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.

For more information, see Chapter 5, "Working with Decision Tables".

1.1.4 What Are Facts and Bucketsets?

In Oracle Business Rules, facts are the objects that rules reason on. Each fact is an instance of a fact type. You must import or create one or more fact types before you can create rules.

In Oracle Business Rules a fact is an asserted instance of a class. 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 Engine.

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. In the Oracle Business Rules runtime such fact type instances are called facts.

You can create bucketsets to define a list of values or a range of values of a specified type. After you create a bucketset you can associate the bucketset with a fact property of matching type. Oracle Business Rules uses the bucketsets that you define to specify constraints on the values associated with fact properties in rules or in Decision Tables. You can also use bucketsets to specify constraints for variable initial values and function return values or function argument values.

For more information, see:

- Section 1.3, "Oracle Business Rules Engine Architecture"
- Chapter 3, "Working with Facts and Bucketsets"

1.1.5 What Are Rulesets?

A ruleset is an Oracle Business Rules container for 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:

- Chapter 4, "Working with Rulesets and Rules"
- "Ordering Rule Firing" in the *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*

1.1.6 What Are Decision Functions?

A decision function provides a contract for invoking rules from Java or SOA (from an SOA composite application or from a BPEL process). The contract includes input fact types, rulesets to run, and output fact types. For more information, see Chapter 6, "Working with Decision Functions".

1.1.7 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 Chapter 7, "Working with Rules SDK Decision Point API".

1.1.8 What Are Dictionaries?

A dictionary is an Oracle Business Rules container for facts, functions, globals, bucketsets, 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 Section 2.2, "Working with a Dictionary and Dictionary Links".

1.2 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 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 an 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

1.2.1 Decision Component (Business Rules) in an 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 a 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 an SOA composite application using the SOA Business Rule service engine that efficiently applies rules to facts and defines and processes rules.

Rules Engine has the following features:

- High performance: Rules Engine implements specialized matching algorithms for facts that are defined in the system.
- Thread-safe execution suitable for a parallel processing architecture: Rules Engine provides one thread that can assert facts while another is evaluating the network.

For more information, see Section 1.3, "Oracle Business Rules Engine Architecture".

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

Oracle Business Rules Rules Engine (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.

Oracle JDeveloper Rules Designer allows business rules to be specified separately from application code. Separating the business rules from application code allows business analysts 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.

Rules Engine has the following features:

- High performance: Rules Engine implements specialized matching algorithms for facts that are defined in the system.
- Thread-safe execution suitable for a parallel processing architecture: Rules Engine provides one thread that can assert facts while another is evaluating the network.

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 Section 1.3, "Oracle Business Rules Engine Architecture" and Section 1.2.4, "Oracle Business Rules SDK".

1.2.3 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 Section 1.2.5, "Rules Designer".

For detailed information about RL Language, see *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*.

1.2.4 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.

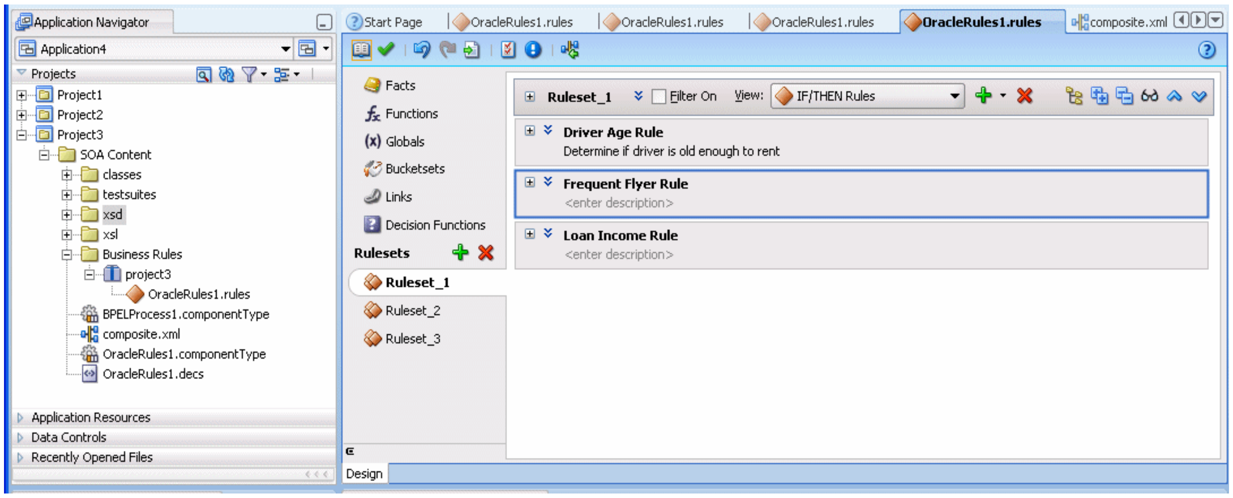
This guide describes the Oracle Business Rules SDK Decision Point API. Using a Decision Point you can access a dictionary and run the rules in the dictionary. For complete Oracle Business Rules SDK API information, see *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*.

For more information, see Chapter 7, "Working with Rules SDK Decision Point API".

1.2.5 Rules Designer

The Oracle Business Rules Designer (Rules Designer) extension to JDeveloper is an editor that enables you to create and edit rules as Figure 1–1 shows.

Figure 1–1 Oracle JDeveloper with Rules Designer



Rules Designer provides a point-and-click interface for creating rules and editing existing rules. Using Rules Designer 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 provides an easy way for you to create, view, and modify business rules.

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.

1.3 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 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. When a rule fires that matches a set of facts, the rule may add facts. 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**.

1.3.1 Declarative Rules

With Oracle Business Rules you can use declarative rules, where you create rules that make declarations based on facts rather than coding. For 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.

1.3.2 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.

1.3.3 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.

1.3.4 Rule Firing and Rule Sessions

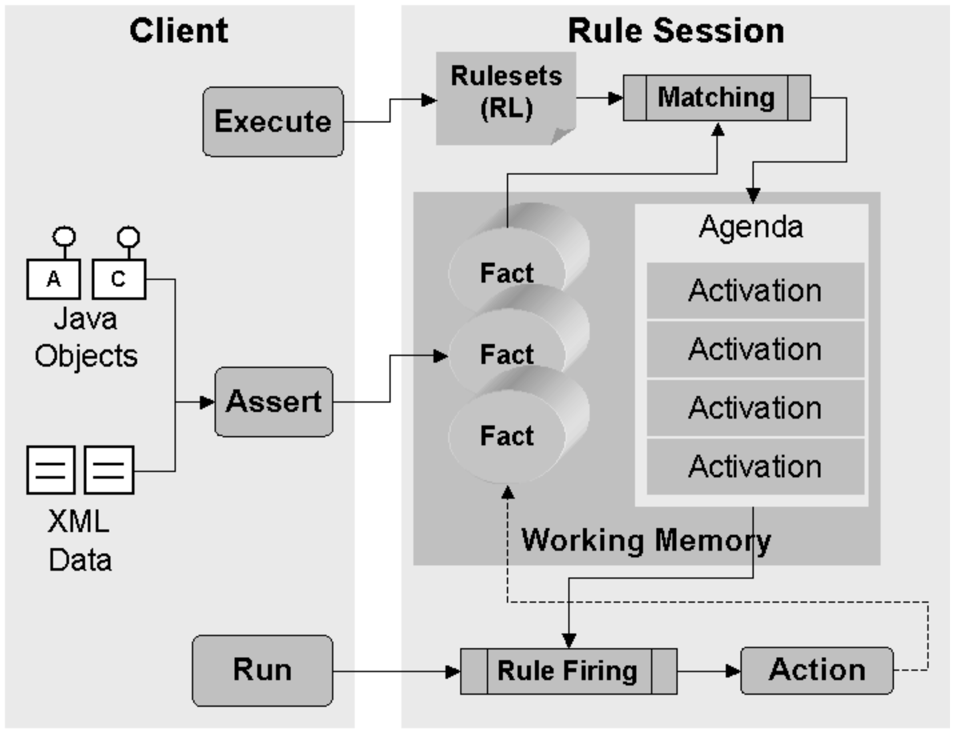
A Rule Session consists of rules, facts and an agenda. An assert or retract adds or removes fact instances from working memory.

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 1–2 shows these parts of Oracle Business Rules runtime.

Figure 1–2 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 most recently activated rule is fired first

Only rules within rulesets on the stack are fired, but all rules in a rule session are matched and, if matched, activated.

2 Working with Data Model Elements

In Oracle Business Rules the data model consists of fact types, functions, globals, bucketsets, decision functions, and dictionary links.

This chapter includes the following sections:

- Section 2.1, "Introduction to Working with Data Model Elements"
- Section 2.2, "Working with a Dictionary and Dictionary Links"
- Section 2.3, "Working with Oracle Business Rules Globals"
- Section 2.4, "Working with Decision Functions"
- Section 2.5, "Working with Oracle Business Rules Functions"

For more information, see Section 1.1.8, "What Are Dictionaries?".

2.1 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:

- Chapter 3, "Working with Facts and Bucketsets"
- Chapter 4, "Working with Rulesets and Rules"
- Chapter 5, "Working with Decision Tables"

2.2 Working with a Dictionary and Dictionary Links

A dictionary is an Oracle Business Rules container for facts, functions, globals, bucketsets, 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. Using Oracle Business Rules, a data model is contained in one or more dictionaries. All the data model elements referenced by the rulesets must be available in the dictionary.

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

2.2.1 Introduction to Dictionaries and Dictionary Links

Each Oracle Business Rules dictionary lets you include links to other dictionaries. Each dictionary that you create also includes the **built-in dictionary**; this dictionary includes standard functions and types that all Oracle Business Rules applications need. In addition to the main dictionary, you create one or more application-specific dictionaries, such as `PurchaseItems.rules`. You can read and write the properties of these 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; in this case, all but the first link is ignored.

For more information, see Section 2.2.8, "What You Need to Know About Dictionary Linking".

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

Oracle JDeveloper provides many ways to create a dictionary for Oracle Business Rules. This shows one way you can create a dictionary in an SOA project. You can create a dictionary for use in an SOA application.

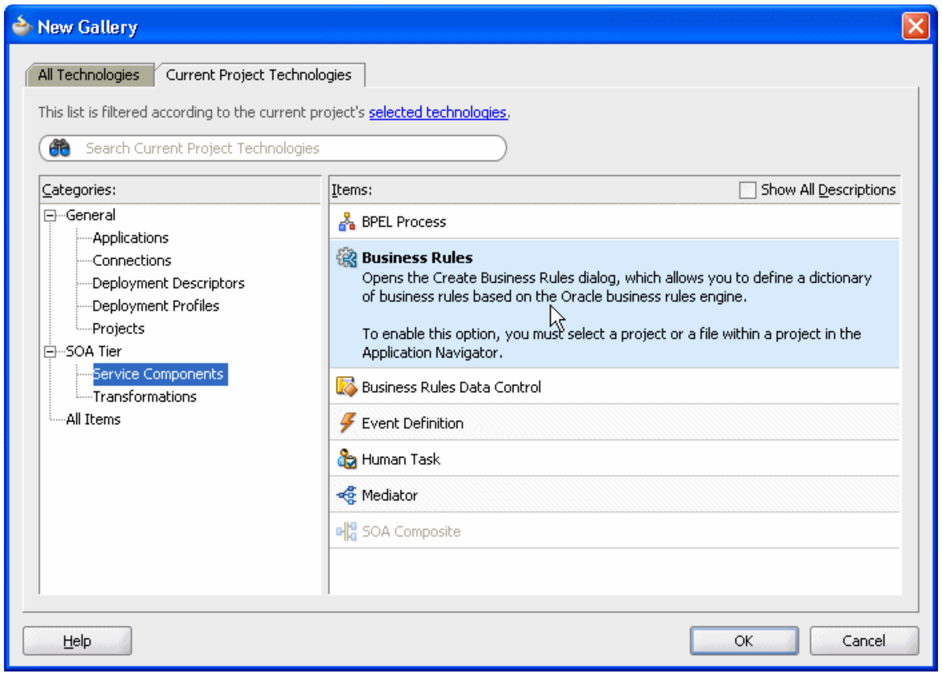
A typical SOA composite design pattern is to provide each application with its own dictionary. When different applications need access to the same parts of a common data model, you can use dictionary links to include a target application's dictionary in the dictionary of a source application. Doing so copies the target application's dictionary into the source application. Therefore, when you work with a dictionary that contains links the linked contents are referred to as local contents.

You may also create a dictionary in the business tier, for use outside of an SOA application. For more information, see Section 9.2.4, "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 an SOA application and select or create an SOA project.
2. Right-click, and from the dropdown list select **New...**.
3. In the New Gallery select the **Current Project Technologies** tab and, in the **Categories** area, expand **SOA Tier** as shown in Figure 2–1.

Figure 2–1 Creating a Business Rules Dictionary for an SOA Project



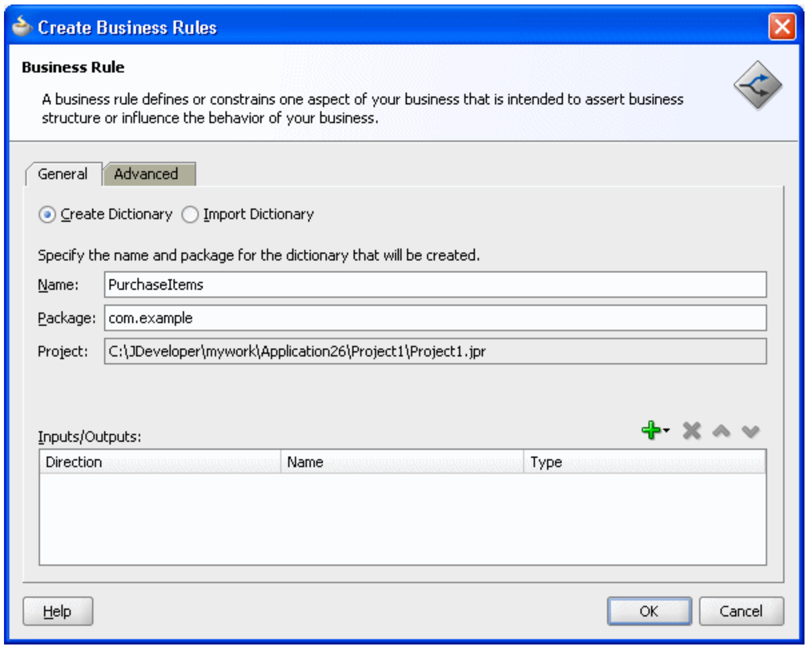
This list is filtered according to the current project's selected technologies.

Opens the Create Business Rules dialog, which allows you to define a dictionary of business rules based on the Oracle business rules engine.

To enable this option, you must select a project or a file within a project in the Application Navigator.

4. In the New Gallery window, in the **Items** area, 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


Create Business Rules

Business Rule

A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business.

General | Advanced

[✓ Create Dictionary] Import Dictionary

Specify the name and package for the dictionary that will be created.

Name: PurchaseItems

Package: com.example

Project: C:\JDeveloper\mywork\Application26\Project1\Project1.jpr

Inputs/Outputs:

Direction	Name	Type

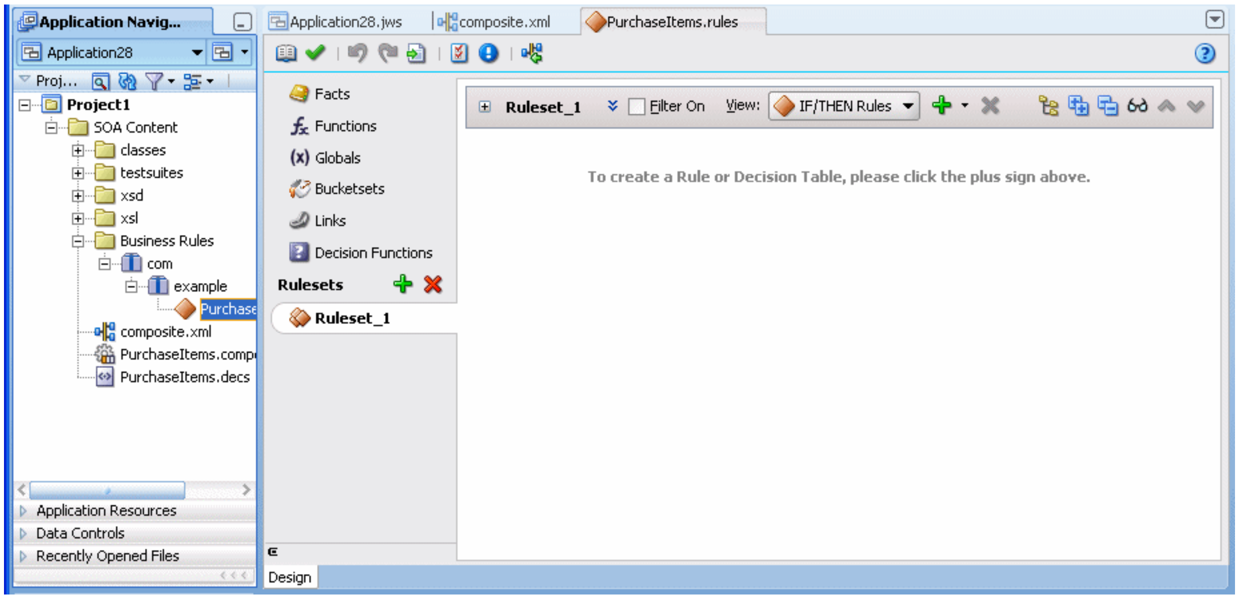
7. To specify the inputs and outputs:
   a. Click the **Add** icon and select **Input** to create an input or **Output**, to create an output.
   b. In the Type Chooser dialog, expand the appropriate XSD and select the appropriate type.
   c. Click **OK** to close the Type Chooser dialog.

   You can later add inputs or outputs, or remove the inputs or outputs. For more information, see Chapter 6, "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.

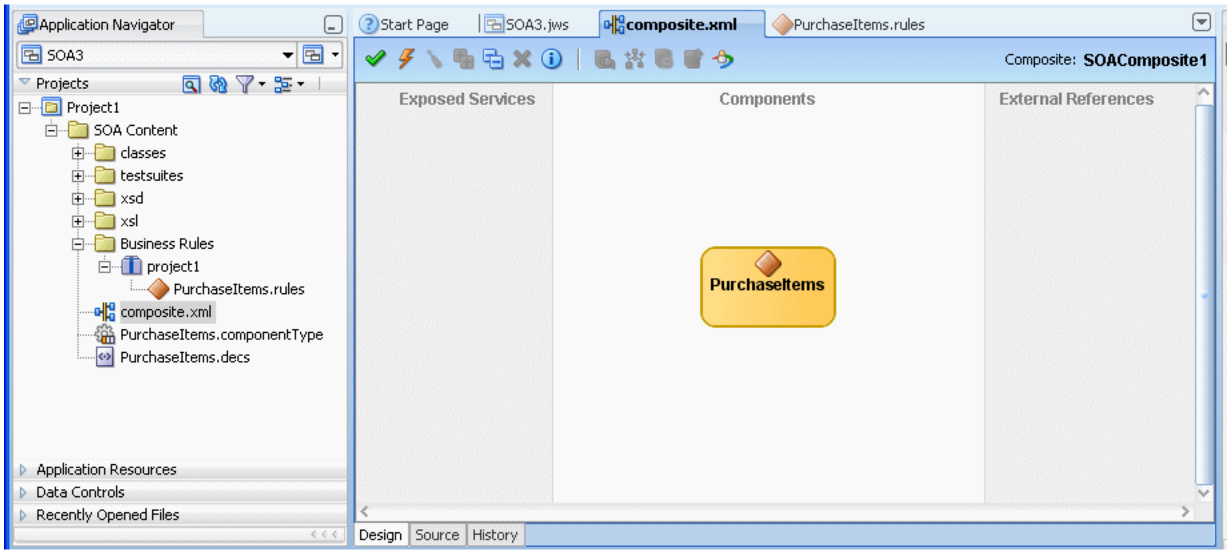
Figure 2–3 Creating a New Oracle Business Rules Dictionary PurchaseItems



To create a Rule or Decision Table, please click the plus sign above.

9. Oracle JDeveloper also creates a Decision component in composite.xml. To view this component double-click the `composite.xml` file, as Figure 2–4 shows.

Figure 2–4 Decision Component Shown in Composite Editor



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

The simplest way to create a rules dictionary is using Rules Designer. You can create a dictionary for use in the business tier, outside of an SOA application. For information on using Oracle Business Rules without SOA, see Chapter 9, "Creating a Rule-enabled Non-SOA Java EE Application".

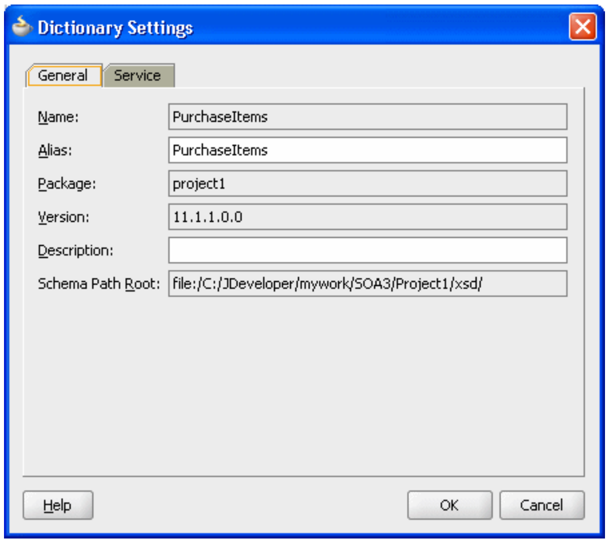
2.2.4 How to View and Edit Dictionary Settings

You can view and edit dictionary settings using the **Dictionary Settings** icon.

To change the dictionary alias:

1. In Oracle JDeveloper, select an Oracle Business Rules dictionary.
2. In Rules Designer, click the **Dictionary Settings** icon.
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 2–5.

Figure 2–5 Dictionary Settings Dialog to Change Dictionary Alias or Description



Name:	PurchaseItems
Alias:	PurchaseItems
Package:	project1
Version:	11.1.1.0.0
Description:	
Schema Path Root:	file:/C:/JDeveloper/mywork/SOA3/Project1/xsd/

4. Click **OK**.

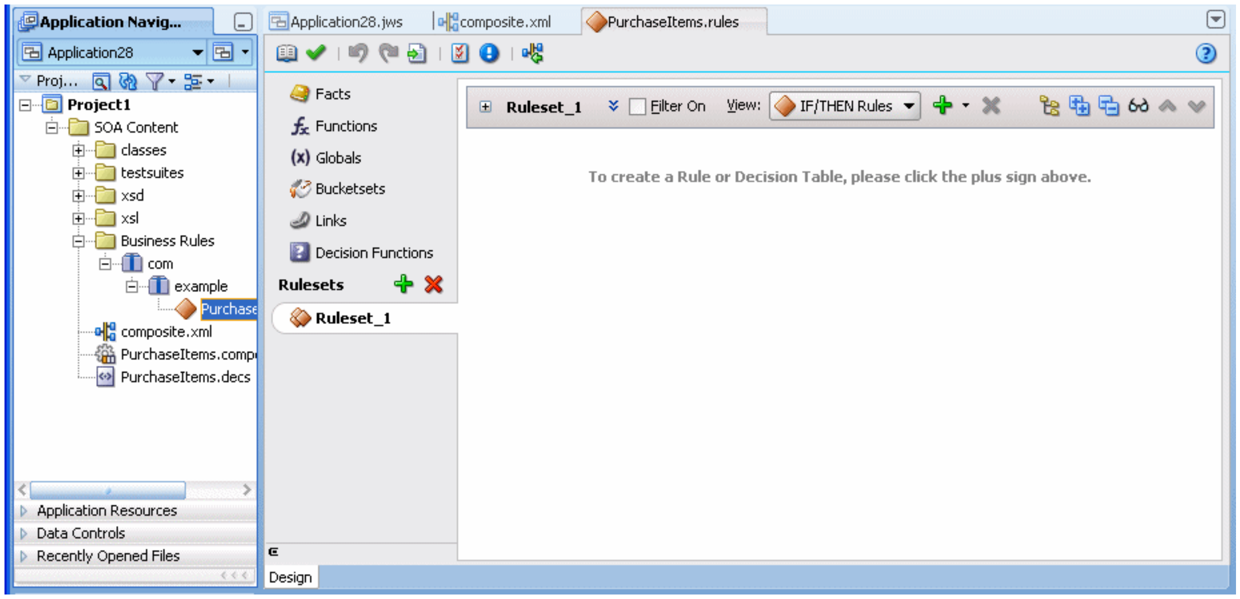
2.2.5 How to Rename a Dictionary or Rename a Dictionary Package

In the Application Navigator you can rename a dictionary or rename a dictionary package name. Note that this the rename operation changes the name of the dictionary but not the alias. Change the alias from the Dictionary Settings dialog. In general, these should be set the same value. For more information, see Section 2.2.4, "How to View and Edit Dictionary Settings".

To rename a dictionary package:

1. In Oracle JDeveloper, in Application Navigator select the dictionary, as shown in Figure 2–6.

Figure 2–6 Selecting an Oracle Business Rules Dictionary to Rename



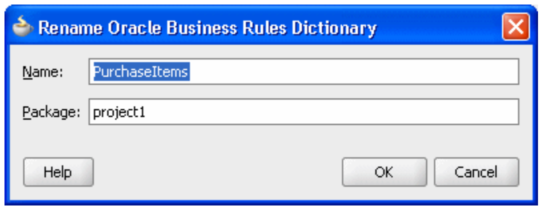
To create a Rule or Decision Table, please click the plus sign above.

2. If the dictionary is open, select the tab showing the dictionary name and click the **Close** icon to close the dictionary.

3. In the Application Navigator, select the dictionary name.

4. From the **File** menu select **Rename...**. This displays the Rename Oracle Business Rules Dictionary dialog, as shown in Figure 2–7.

Figure 2–7 Rename Oracle Business Rules Dialog



Name: PurchaseItems

Package: project1

5. To rename the dictionary, in the **Name** field, enter a name.

6. To rename the dictionary package, in the **Package** field, enter a name.

7. Click **OK**.

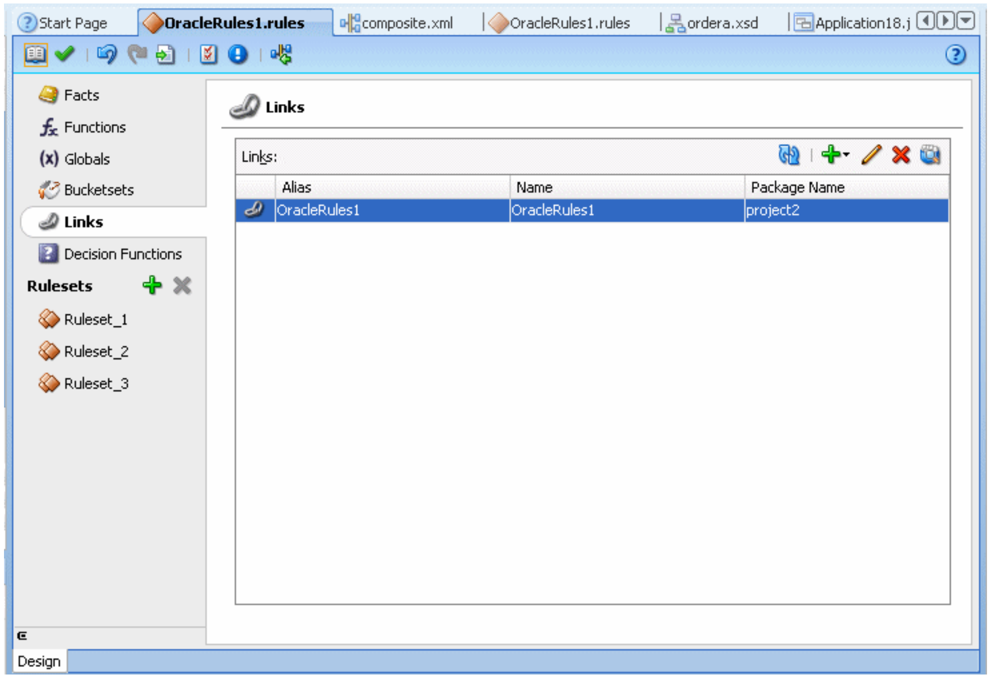
2.2.6 How to Link to a Dictionary

You can link to a dictionary in the same application or in another 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 2–8.

Figure 2–8 Rules Designer Links Tab

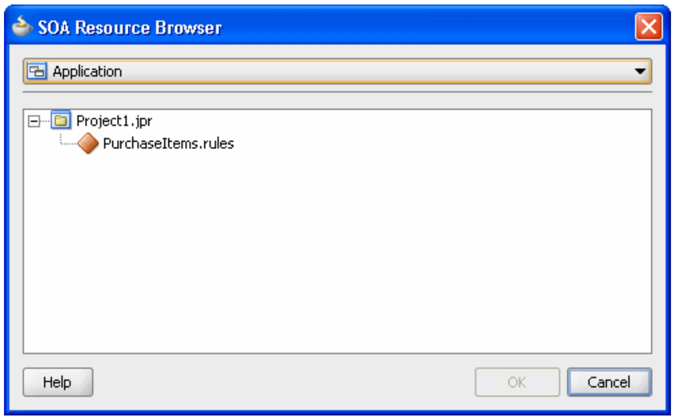


Alias	Name	Package Name
OracleRules1	OracleRules1	project2

2. In the **Links** area, click the **Create** icon and from the dropdown list select **Resource Picker**. 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 2–9.

Figure 2–9 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 Chapter 10, "Working with Oracle Business Rules and ADF Business Components".

To link to the decision point dictionary:

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

2. In the **Links** area, click **Create** and from the dropdown list select **Decision Point Dictionary**. This operation takes awhile. You need to wait for the Decision Point Dictionary to load.

2.2.7 How to Update a Linked Dictionary

When you have a dictionary, for example a dictionary named Project_rules1 that links to a another dictionary, for example a dictionary named Shared_rules you need to see any changes made to either dictionary in both dictionaries. Using Rules Designer you can modify the Shared_rules dictionary and see those modifications in Project1_rules1 by updating the Project_rules1 dictionary, or you can close and then reopen Rules Designer.

To update a linked dictionary:

1. Using these sample dictionary names click the **Save** icon to save the Shared_rules dictionary.
2. Select the Project_rules1 dictionary.
3. Select the **Links** navigation tab.
4. Click the **Dictionary Cache...** icon.
5. In the Dictionary Finder Cache dialog, select the appropriate linked dictionary.
6. Click the **Clear** icon.
7. In the Dictionary Finder Cache dialog, click **Close**.
8. Click the **Validate** icon.

2.2.8 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.

Note the following naming constraints for combined dictionaries:

- Within a combined dictionary the full names of the dictionaries, including the package and name, must be distinct. In addition, the dictionary aliases must be distinct.

- Oracle Business Rules requires that the aliases of data model definitions of a particular kind, for example, function, Oracle RL class, or bucketset, must be unique within a dictionary.
- Within a combined dictionary, a definition may be qualified by the alias of its immediately containing dictionary. Definitions in the top and main dictionaries do not have to be qualified. Definitions in other dictionaries must be qualified.
- 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 Appendix A, "Oracle Business Rules Files and Limitations".

2.2.9 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.

2.2.10 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.

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

Oracle Business Rules accesses input and output variables by type only, and not by name. Thus, if you have two inputs of the same type, *input1* and *input2*, the rules are not able to distinguish which is *input1* and which is *input2*. The variable names are only useful in the BPEL process definition. The mapping for the Oracle Business Rules business terms default to *fact type.property*, and there may be no relationship to the BPEL variable name.

2.3 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.

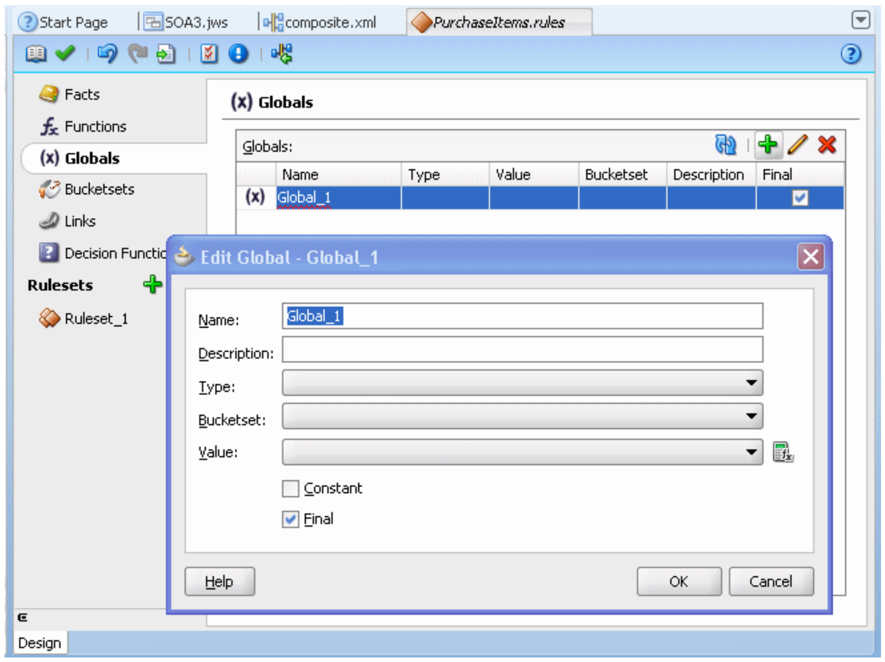
2.3.1 How to Add Oracle Business Rules Globals

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** icon. This adds a global and displays the Edit Global dialog, as shown in Figure 2–10.

Figure 2–10 Adding a Global in Rules Designer



	Name	Type	Value	Bucketset	Description	Final
(x)	Global_1					✓

Name: Global_1
Description:
Type:
Bucketset:
Value:
☐ Constant
☑ Final

3. In the **Name** field, enter a name or accept the default value.
4. In the **Type** field, select the type from the dropdown list.
5. Optionally, in the **Bucketset** field, select a value from the dropdown list.
6. In the **Value** field, enter a value, select a value from the dropdown list, or click the **Expression Builder** icon to enter an expression. For more information, see Section 4.10, "Working with Expression Builder".
7. If the global is a constant, then select the **Constant** checkbox. When selected, this option specifies that the global is a constant value. For more information, see Section 2.3.3, "What You Need to Know About the Final and Constant Options".

8. If the global is a nonfinal, then deselect the **Final** checkbox. When unselected, this option specifies that the global is modifiable, for instance, in an assign action.

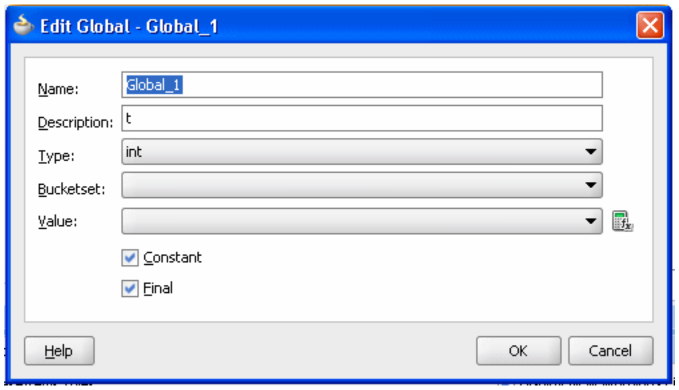
2.3.2 How to Edit Oracle Business Rules Globals

You can use Rules Designer to edit globals.

To edit a Global:

1. In Rules Designer, select the **Globals** navigation tab.
2. Double-click the globals icon in a row in the Globals table. When you double-click the globals icon in a row this displays the **Edit Global -** *Global Name* window as shown in Figure 2–11. In this window you can edit a global and change field values, including the **Final** field and the **Constant** field (the **Constant** field is only shown when you double-click a global to display the Edit Global dialog.

Figure 2–11 Edit Global Window


Edit Global - Global_1

Name: Global_1
Description: t
Type: int
Bucketset:
Value:
[✓] Constant
[✓] Final

2.3.3 What You Need to Know About the Final and Constant Options

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

Note the following when you use globals:

- When you deselect **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 bucket values, bucket range endpoints, and ruleset filter values are always constant.

2.4 Working with Decision Functions

The data model includes decision functions. For information on working with decision functions, see Section 6.1, "Introduction to Decision Functions".

2.5 Working with 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.

2.5.1 Introduction to Oracle Business Rules Functions

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.
- Bucketset: The bucketset to associate with the Oracle Business Rules function.
- Arguments: The function arguments. Each function argument includes a name and a type.
- 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 Section 8.1.1, "How to Test Rules Using a Test Function in Rules Designer".

2.5.2 How to Add an Oracle Business Rules Function

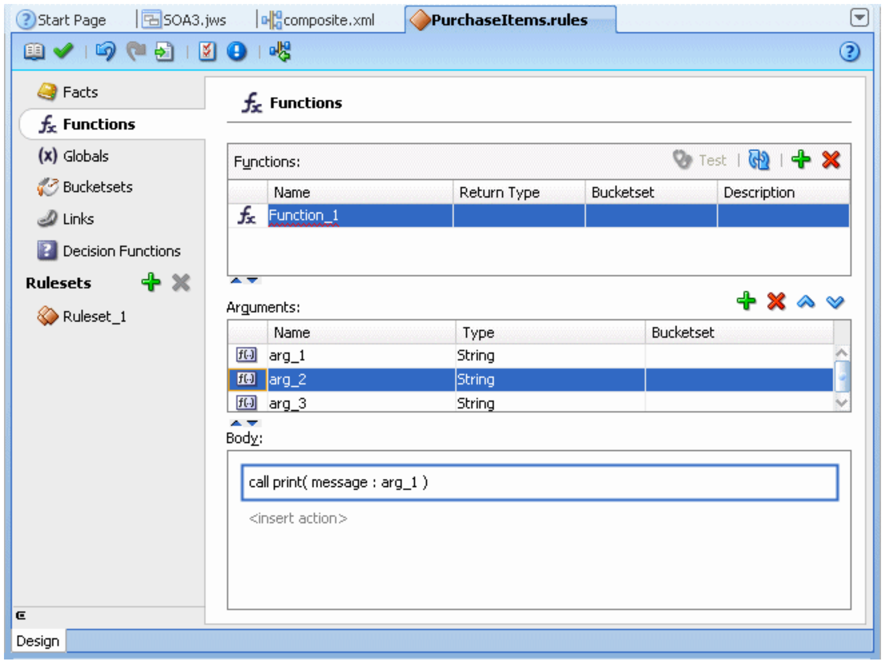
You use Rules Designer to add an Oracle Business Rules function.

To add an Oracle Business Rules Function:

1. In Rules Designer, select the **Functions** navigation tab.
2. Select the **Create...** icon.
3. Enter the function name in the **Name** field, or use the default name.
4. Select the return type from the **Return Type** dropdown list. For example, select `void`.

5. Optionally, select a bucketset to associate with the function return type from the dropdown list in the **Bucketset** 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 dropdown list.
9. For each argument in the **Bucketset** field, to limit the argument values as specified by a bucketset constraint, select a bucketset from the dropdown list.
10. In the **Body** area, enter actions and arguments for the function body. For example, see Figure 2–12.

Figure 2–12 Adding an Oracle Business Rules Function



Name	Return Type	Bucketset	Description
Function_1			

Name	Type	Bucketset
arg_1	String	
arg_2	String	
arg_3	String	

```
call print( message : arg_1 )
<insert action>
```

3 Working with Facts and Bucketsets

Facts are the objects that rules reason on and bucketsets define groupings of fact property values.

This chapter includes the following sections:

3.1 Introduction to Working with Facts and Bucketsets

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.

In Rules Designer, you make business objects and their methods known to Oracle Business Rules using fact definitions that are part of a data model.

You must create one or more facts, and optionally bucketsets, before you can create rules.

In Rules Designer you can work with the following types 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 Section 3.2, "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 Section 3.3, "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 without methods. An RL Fact contains a list of properties of types available in the data model, either RL Facts, Java Facts, or primitive types. You can use RL Facts to extend a Java application object model by providing virtual dynamic types.

  For more information, see Section 3.4, "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 Section 3.5, "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.

You can create bucketsets to define a list of values or a range of values of a specified type. After you create a bucketset you can associate the bucketset with a fact property of matching type. When a bucketset is associated with a fact property Oracle Business Rules uses the buckets that you define as constraints for the values for the fact properties in rules or in Decision Tables.

For more information, see:

- Section 3.6, "Working with Bucketsets"
- Section 3.7, "Associating a Bucketset with Facts and Functions"

3.2 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 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`).

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.

To create XML fact types, perform the following steps:

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 Section 3.2.1, "How to Import 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 Section 4.3, "Working with Rules" and Section 5.2, "Creating Decision Tables".

Elements and types defined in 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 Section 4.8, "Working with Tree Mode Rules".

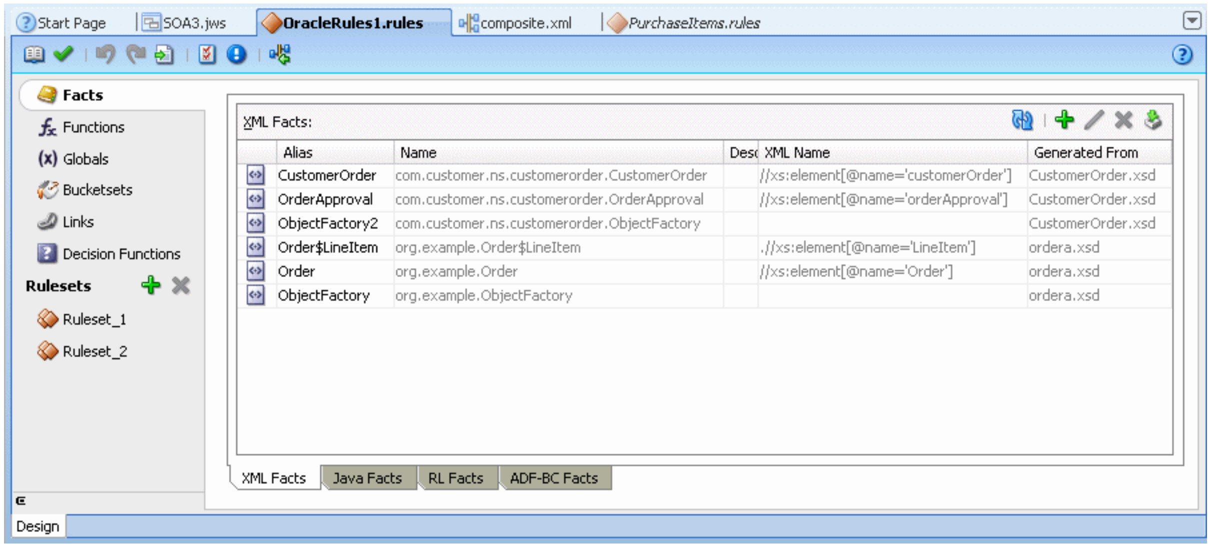
3.2.1 How to Import XML Schema and Add XML Facts

Before you can use XML Schema definitions in a data model you must import 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.

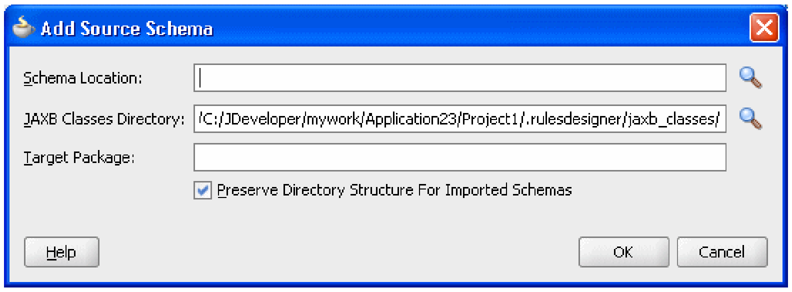
Figure 3–1 The XML Facts Tab in Rules Designer



	Alias	Name	Desc	XML Name	Generated From
	CustomerOrder	com.customer.ns.customerorder.CustomerOrder		//xs:element[@name='customerOrder']	CustomerOrder.xsd
	OrderApproval	com.customer.ns.customerorder.OrderApproval		//xs:element[@name='orderApproval']	CustomerOrder.xsd
	ObjectFactory2	com.customer.ns.customerorder.ObjectFactory			CustomerOrder.xsd
	Order$LineItem	org.example.Order$LineItem		.//xs:element[@name='LineItem']	ordera.xsd
	Order	org.example.Order		//xs:element[@name='Order']	ordera.xsd
	ObjectFactory	org.example.ObjectFactory			ordera.xsd

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...**. This displays the Add Source Schema dialog, as shown in Figure 3–2.

Figure 3–2 XML Fact: Add Source Schema Dialog


Add Source Schema

Schema Location:

JAXB Classes Directory: /C:/JDeveloper/mywork/Application23/Project1/.rulesdesigner/jaxb_classes/

Target Package:

☑ Preserve Directory Structure For Imported Schemas

5. In the Add Source Schema dialog, in the **Schema Location** field, enter the location of the XML Schema you want to import, or click **Browse** to locate the XML schema. During the import the file is copied into the project.

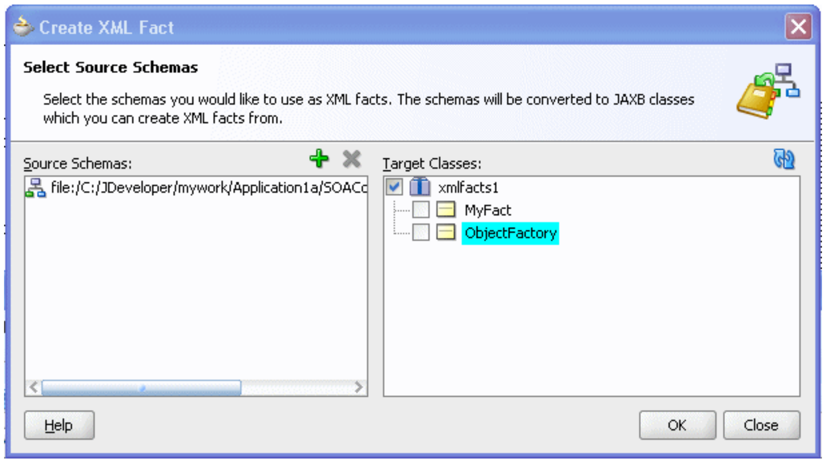
6. In the Add Source Schema dialog, in the **JAXB Classes Directory** field, accept the default or enter the directory where you want Rules Designer to store the JAXB-generated Java source and class files.

7. In the Add Source Schema dialog, in the **Target Package** field, enter a target package name or leave this field empty. 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`.

8. 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, as shown in Figure 3–3.

Figure 3–3 XML Fact: Create XML Fact Dialog


Create XML Fact

Select Source Schemas

Select the schemas you would like to use as XML facts. The schemas will be converted to JAXB classes which you can create XML facts from.

Source Schemas:

file:/C:/JDeveloper/mywork/Application1a/SOACc

Target Classes:

☑ xmlfacts1
- ☐ MyFact
- ☐ ObjectFactory

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

10. Click **OK**.

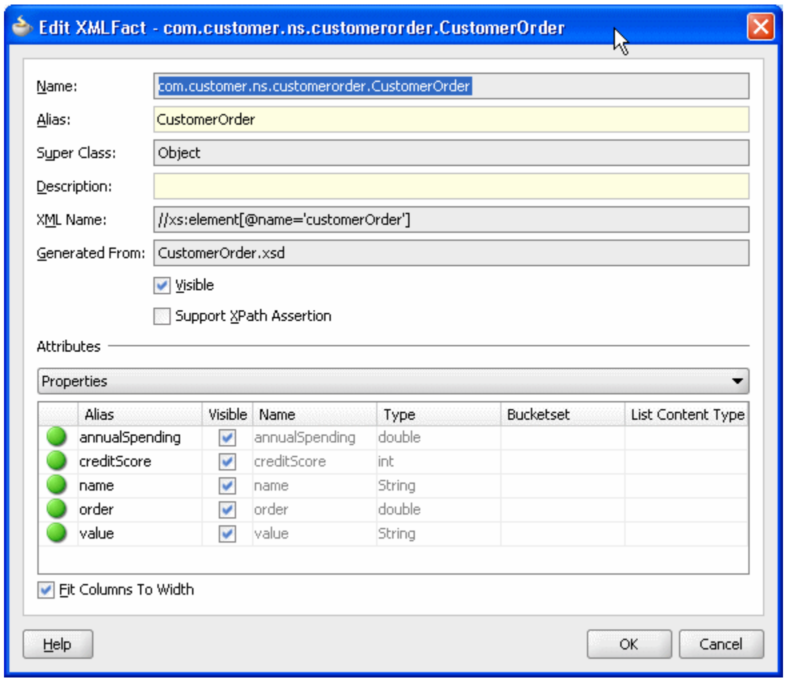
3.2.2 How to Display and Edit XML Facts

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 3–4.

Figure 3–4 Edit XML Fact Dialog


Edit XMLFact - com.customer.ns.customerorder.CustomerOrder

Name:	com.customer.ns.customerorder.CustomerOrder
Alias:	CustomerOrder
Super Class:	Object
Description:	
XML Name:	//xs:element[@name='customerOrder']
Generated From:	CustomerOrder.xsd

✓ Visible
Support XPath Assertion

Attributes

Properties

	Alias	Visible	Name	Type	Bucketset	List Content Type
●	annualSpending	✓	annualSpending	double		
●	creditScore	✓	creditScore	int		
●	name	✓	name	String		
●	order	✓	order	double		
●	value	✓	value	String		

✓ Fit Columns To Width

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.
Alias	Enter the XML Fact alias. You can change this value. This defaults to the unqualified name of the class.
Super Class	Displays Java super class associated with this fact.
Description	Enter the XML Fact description.
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 dropdown 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.

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

Field	Description
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.
Attributes area	Select the available constructors, properties, methods, or fields associated with the JAXB class for the XML Fact to display or edit.

3.2.3 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.

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 the **Reload XML Facts from Updated Schemas** icon.

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).

3.2.4 What You Need to Know About XML Facts

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

- When writing rules, the `assertTree` action type is available only in advanced mode. For more information on creating rules using `assertTree`, see Section 4.8, "Working with Tree Mode Rules".
- When creating a decision function, the `tree` option for the input types defines whether `assert` or `assertTree` is used to put the input facts in working memory. For more information on `assertTree`, see Section 4.8, "Working with Tree Mode Rules".
- 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 Example 3–1.

Example 3–1 XML Schema Restriction Example

```
<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 bucketset for each enum containing all of the enum values and null. For more information on bucketsets, see Section 3.6, "Working with Bucketsets".

- 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 Section 3.2.3, "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 Section C.13, "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 Section D.8, "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 Section D.9, "How Are Decision Service Inputs and Outputs Restricted?".

3.3 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).

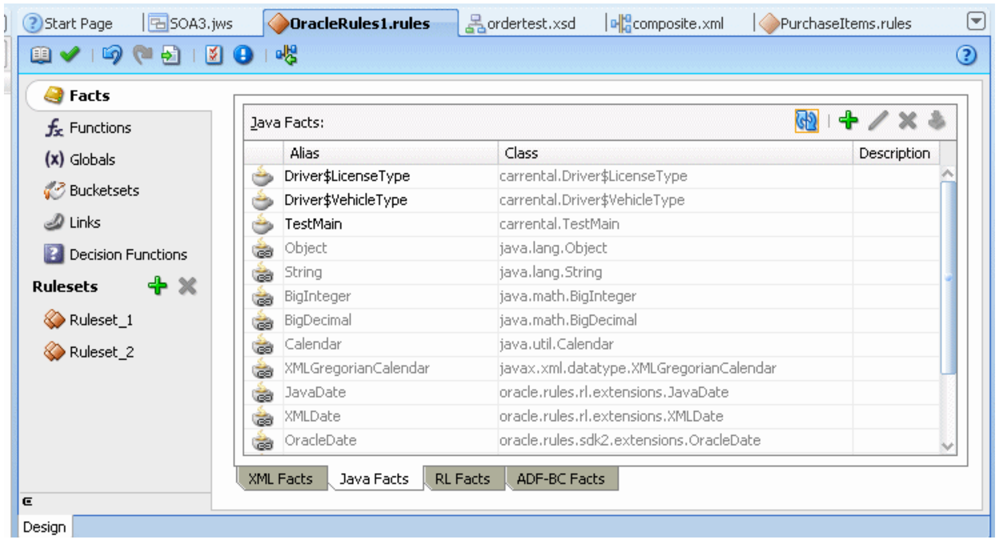
3.3.1 How to Import Java Classes and Define Java Facts

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.

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 3–5.

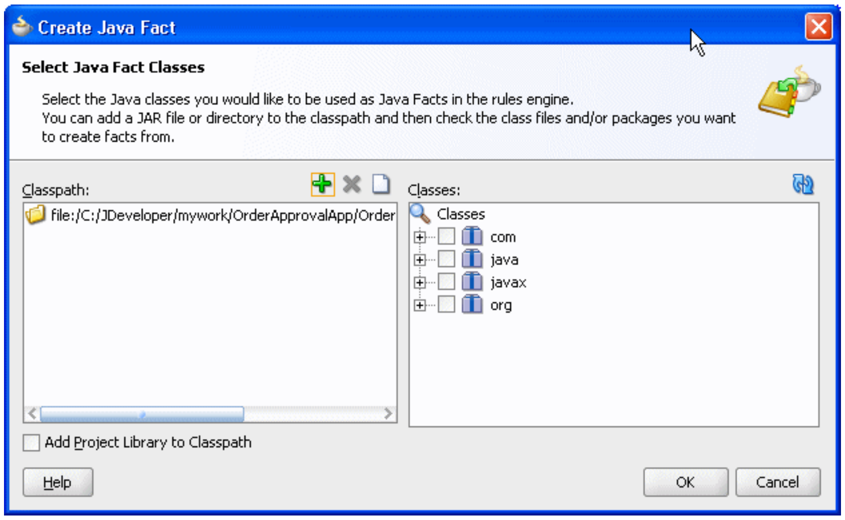
Figure 3–5 The Java Facts Table in the Facts Navigation Tab



Alias	Class	Description
Driver$LicenseType	carrental.Driver$LicenseType	
Driver$VehicleType	carrental.Driver$VehicleType	
TestMain	carrental.TestMain	
Object	java.lang.Object	
String	java.lang.String	
BigInteger	java.math.BigInteger	
BigDecimal	java.math.BigDecimal	
Calendar	java.util.Calendar	
XMLGregorianCalendar	javax.xml.datatype.XMLGregorianCalendar	
JavaDate	oracle.rules.rl.extensions.JavaDate	
XMLDate	oracle.rules.rl.extensions.XMLDate	
OracleDate	oracle.rules.sdk2.extensions.OracleDate	

3. In the **Java Facts** tab, click **Create...**. This displays the Create Java Fact dialog, as shown in Figure 3–6.

Figure 3–6 Adding a Java Fact


Create Java Fact

Select Java Fact Classes

Select the Java classes you would like to be used as Java Facts in the rules engine.
You can add a JAR file or directory to the classpath and then check the class files and/or packages you want to create facts from.

Classpath:
file:/C:/JDeveloper/mywork/OrderApprovalApp/Order

Classes:
Classes
com
java
javax
org

Add Project Library to Classpath

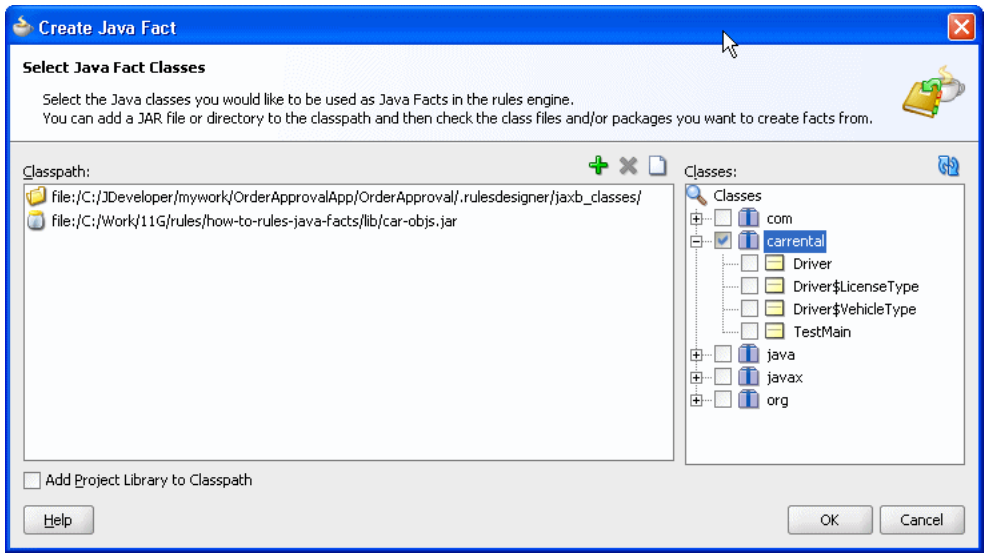
4. In the Create Java Fact dialog, if the classpath that contains the classes you want to import is not shown in the **Classpath** area, then click **Add to Classpath**. This displays the Choose Directory/Jar dialog.

   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 as shown in Figure 3–7.

Figure 3–7 Selecting Java Classes for Java Facts


Create Java Fact

Select Java Fact Classes

Select the Java classes you would like to be used as Java Facts in the rules engine.
You can add a JAR file or directory to the classpath and then check the class files and/or packages you want to create facts from.

Classpath:
file:/C:/JDeveloper/mywork/OrderApprovalApp/OrderApproval/.rulesdesigner/jaxb_classes/
file:/C:/Work/11G/rules/how-to-rules-java-facts/lib/car-objs.jar

Classes:
Classes
- com
- carrental (checked)
  - Driver
  - Driver$LicenseType
  - Driver$VehicleType
  - TestMain
- java
- javax
- org

Add Project Library to Classpath (unchecked)

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

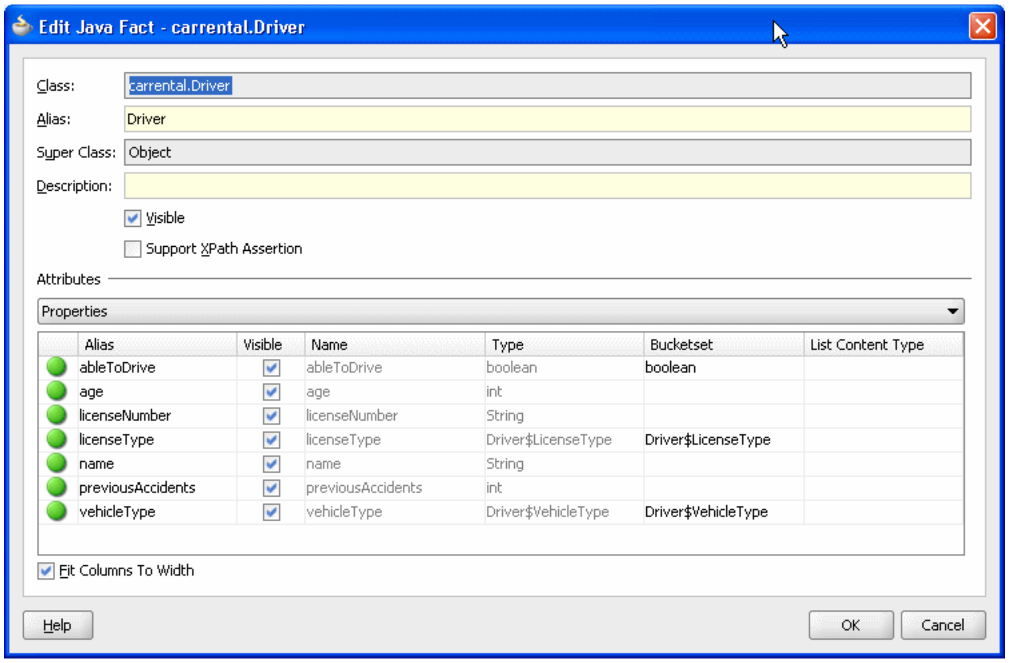
3.3.2 How to Display and Edit Java Facts

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 3–8.

Figure 3–8 Edit Java Fact Dialog



Class: carrental.Driver

Alias: Driver

Super Class: Object

Description:

[✓] Visible

[ ] Support XPath Assertion

Attributes

Properties

	Alias	Visible	Name	Type	Bucketset	List Content Type
●	ableToDrive	✓	ableToDrive	boolean	boolean	
●	age	✓	age	int		
●	licenseNumber	✓	licenseNumber	String		
●	licenseType	✓	licenseType	Driver$LicenseType	Driver$LicenseType	
●	name	✓	name	String		
●	previousAccidents	✓	previousAccidents	int		
●	vehicleType	✓	vehicleType	Driver$VehicleType	Driver$VehicleType	

[✓] Fit Columns To Width

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.
Super Class	Displays Java super class associated with this fact.
Description	Enter the Java Fact description.
Visible	Select to show the Java Fact in dropdown lists in Rules Designer.
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.
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.

3.3.3 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).

3.4 Working with 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 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:

IF customer spent $500 within past 3 months

THEN customer is a Gold Customer

This rule might use a Java Fact to specify the customer data and also use an action that creates an RL Fact, Gold Customer. A rule might be defined to use a Gold Customer fact, as follows:

IF customer is a Gold customer

THEN offer 10% discount

This rule uses the RL Fact named Gold Customer. This rule then infers, using the Gold Customer fact, that if a customer spent $500 within the past 3 months, then the customer is eligible for a 10% discount. In addition rules could specify other ways that a customer becomes a Gold Customer.

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.

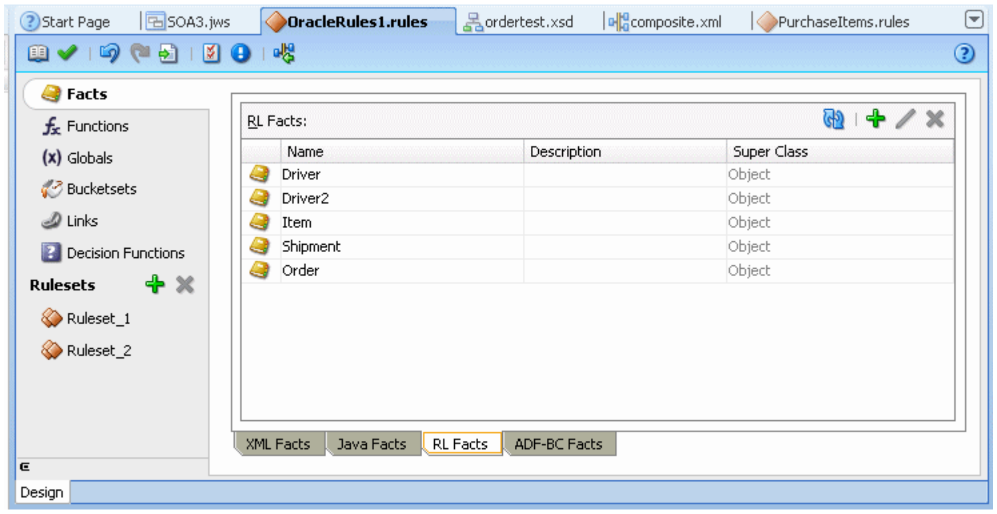
3.4.1 How to Define RL Facts

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 3–9.

Figure 3–9 RL Facts Tab in Rules Designer



Name	Description	Super Class
Driver		Object
Driver2		Object
Item		Object
Shipment		Object
Order		Object

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.

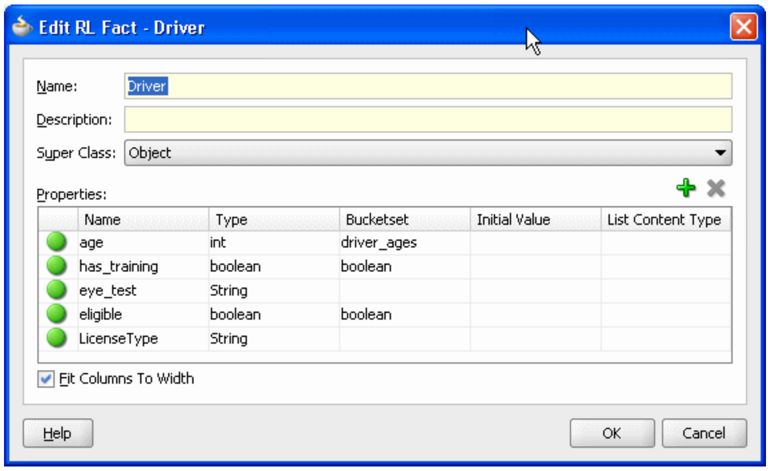
3.4.2 How to Display and Edit RL Facts and Add RL Fact Properties

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 3–10.

Figure 3–10 Edit RL Fact Dialog



Name: Driver

Description:

Super Class: Object

Properties:

	Name	Type	Bucketset	Initial Value	List Content Type
●	age	int	driver_ages		
●	has_training	boolean	boolean		
●	eye_test	String			
●	eligible	boolean	boolean		
●	LicenseType	String			

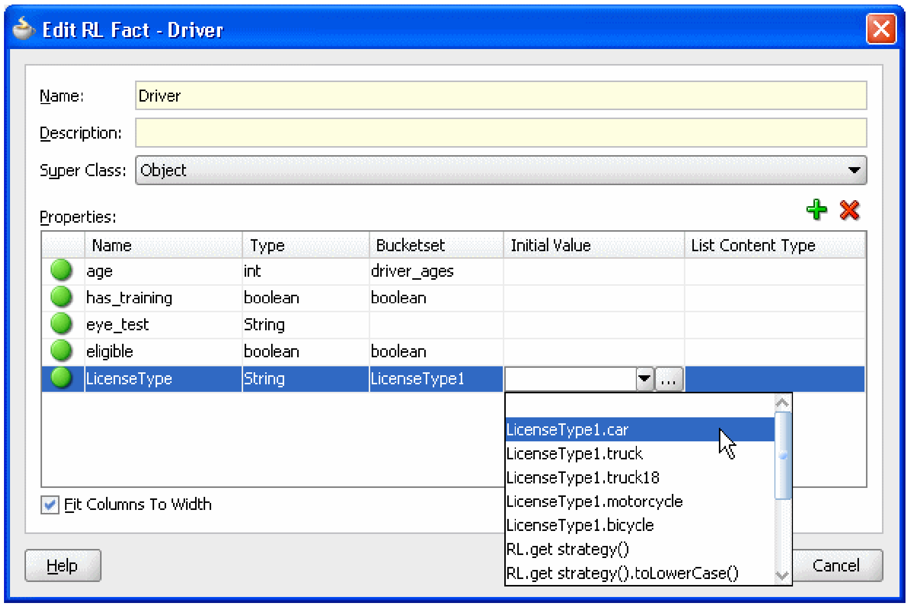
[✓] Fit Columns To Width

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 dropdown list or enter a property type.
   c. To associate a bucketset with the property, from the dropdown list in the **Bucketset** field, select a bucketset.
   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**.

3.4.3 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 dropdown list of possible values as shown in Figure 3–11.

Figure 3–11 Setting RL Fact Property Initial Value



Name: Driver
Description:
Super Class: Object
Properties:

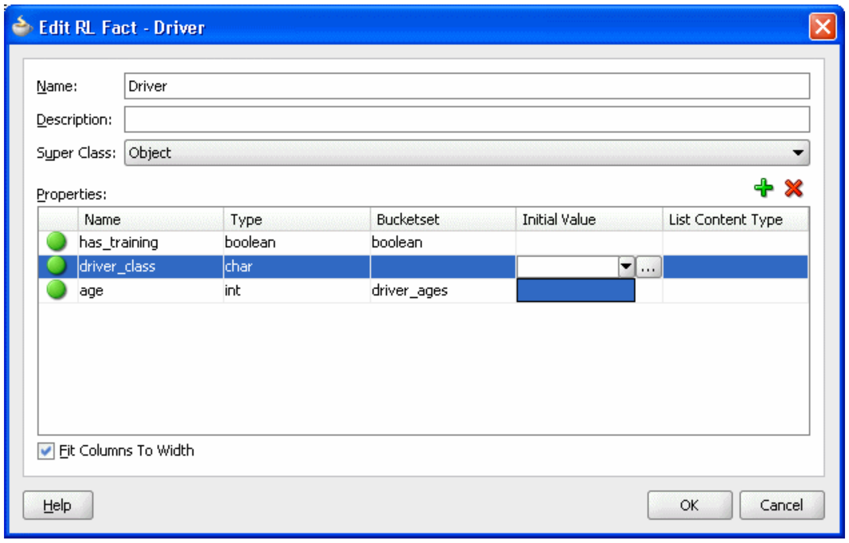
	Name	Type	Bucketset	Initial Value	List Content Type
●	age	int	driver_ages		
●	has_training	boolean	boolean		
●	eye_test	String			
●	eligible	boolean	boolean		
●	LicenseType	String	LicenseType1		

LicenseType1.car
LicenseType1.truck
LicenseType1.truck18
LicenseType1.motorcycle
LicenseType1.bicycle
RL.get strategy()
RL.get strategy().toLowerCase()

☑ Fit Columns To Width

When you are working with some fields in Rules Designer, the initial values dropdown list or other dropdown lists may be empty as shown in Figure 3–12. In this case the dropdown 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** icon to display the expression builder dialog.

Figure 3–12 RL Fact Empty List Options for Initial Value Field



Name: Driver
Description:
Super Class: Object
Properties:

	Name	Type	Bucketset	Initial Value	List Content Type
●	has_training	boolean	boolean		
●	driver_class	char			
●	age	int	driver_ages		

☑ Fit Columns To Width

3.5 Working with 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 Chapter 10, "Working with Oracle Business Rules and ADF Business Components".

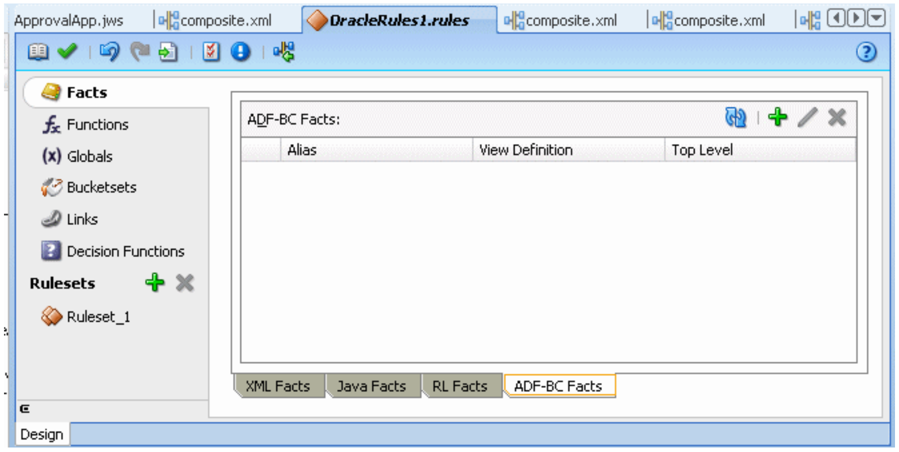
3.5.1 How to Import and Define ADF Business Components Facts

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:

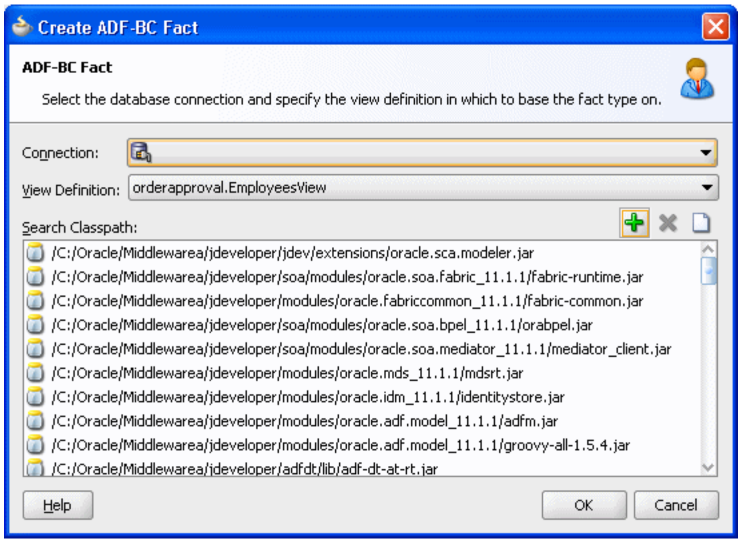
1. Click the **Facts** navigation tab and select the **ADF-BC Facts** tab. This displays the **ADF-BC Facts** table, as shown in Figure 3–13.

Figure 3–13 ADF Business Components Facts Tab



2. Click **Create...**. This displays the ADF Business Components Fact dialog, as shown in Figure 3–14.

Figure 3–14 Create ADF-BC Fact Dialog



Select the database connection and specify the view definition in which to base the fact type on.

Connection:

View Definition: orderapproval.EmployeesView

Search Classpath:

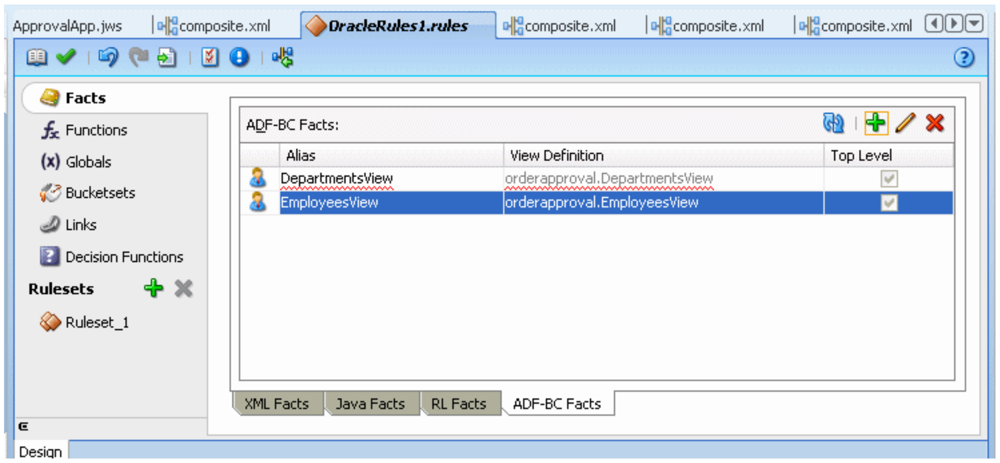
/C:/Oracle/Middlewarea/jdeveloper/jdev/extensions/oracle.sca.modeler.jar
/C:/Oracle/Middlewarea/jdeveloper/soa/modules/oracle.soa.fabric_11.1.1/fabric-runtime.jar
/C:/Oracle/Middlewarea/jdeveloper/modules/oracle.fabriccommon_11.1.1/fabric-common.jar
/C:/Oracle/Middlewarea/jdeveloper/soa/modules/oracle.soa.bpel_11.1.1/orabpel.jar
/C:/Oracle/Middlewarea/jdeveloper/soa/modules/oracle.soa.mediator_11.1.1/mediator_client.jar
/C:/Oracle/Middlewarea/jdeveloper/modules/oracle.mds_11.1.1/mdsrt.jar
/C:/Oracle/Middlewarea/jdeveloper/modules/oracle.idm_11.1.1/identitystore.jar
/C:/Oracle/Middlewarea/jdeveloper/modules/oracle.adf.model_11.1.1/adfm.jar
/C:/Oracle/Middlewarea/jdeveloper/modules/oracle.adf.model_11.1.1/groovy-all-1.5.4.jar
/C:/Oracle/Middlewarea/jdeveloper/adfdt/lib/adf-dt-at-rt.jar

3. In the **Connection** field, from the dropdown list, select the connection which your ADF Business Components objects use. The **Search Classpath** area shows a list of

classpaths. For more information, see Section 3.5.2, "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, as shown in Figure 3–15. Note that the imported fact includes a validation warning. For information on how to remove this validation warning, see Section 3.5.3, "What You Need to Know About ADF Business Components Circular References".

Figure 3–15 ADF Business Components Facts in Rules Designer



Alias	View Definition	Top Level
DepartmentsView	orderapproval.DepartmentsView	✓
EmployeesView	orderapproval.EmployeesView	✓

3.5.2 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).

3.5.3 What You Need to Know About ADF Business Components Circular References

ADF Business Components Facts can include a circular reference, as shown in Figure 3–15. When this warning is shown in the Business Rule validation log you need to manually resolve the circular reference. To do this you must deselect the **Visible** checkbox for one of the properties that is involved in the circular reference.

3.5.4 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 deselecting the **Visible** checkbox 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*.

3.6 Working with Bucketsets

You can create a bucketset 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 bucketset as a **Global Bucketset** that allows reuse, where a bucketset is named and stored in the data model, or as a **Local Bucketset** that is specified when you define a Decision Table and only applies to one condition expression. For more information on using a local bucketset, see Section 5.2.2, "How to Add Condition Rows to a Decision Table".

You can use Bucketsets for the following:

- You can associate fact type properties with bucketsets. This allows you to limit the acceptable set of values for a property of a fact. For more information, see Section 3.7.1, "How to Associate a Bucketset with a Fact Property".
- In a Decision Table a bucketset defines a list of values or value ranges in the condition expressions that are part of the Decision Table. The bucketset values or ranges determine, for each condition expression in a Decision Table, that it has two or more possibilities. Using a bucketset each possibility in a condition expression is divided into groups or ranges where a cell specifies one **Bucket** of values from the bucketset (or possibly multiple buckets of values per cell). For example, if a bucketset is defined for colors, then the buckets could include a list of strings: "blue", "red", and "orange". A bucketset that includes integers could have three buckets, less than 1, 1 to 10, and greater than 10. For more information, see Section 5.2.2, "How to Add Condition Rows to a Decision Table".
- You can associate globals, functions, and function arguments with bucketsets. Associating a bucketset with a global allows for design-time validation that an assigned value is limited to the values specified in the bucketset. Associating a bucketset with a function argument validates that the function is only called with values in the bucketset. Using bucketsets 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. This type of bucketset validation is "weak" in the sense that only design-time constant values are validated. No runtime checks are applied based on the globals or function arguments associated with bucketsets. Associating a bucketset with a function automatically sets a Decision Table condition row to use that bucketset when the function is used as the expression for that condition row. For more information, see Section 3.7.2, "How to Associate a Bucketset with Functions or Function Arguments".

- In addition to design-time validation you can use an LOV bucketset to provide options that are displayed in lists when entering expressions in IF/THEN rule tests and actions. For more information, see Section 4.11.3, "How to Use Bucketsets to Provide Options for Test Expressions".

There are three forms for bucketsets:

- LOV: Defined by a list of values (see Section 3.6.1, "How to Define a List of Values Global Bucketset").
- Range: Defined by a list of value ranges, defined by the range endpoints (see Section 3.6.2, "How to Define a List of Ranges Global Bucketset").
- Enum: Defined by a list of enumerated types that is imported from either of:
  - XML types (see Section 3.6.3, "How to Define an Enumerated Type (Enum) Bucketset from XML Types").
  - Java facts (see Section 3.6.4, "How to Define an Enumerated Type (Enum) Bucketset from Java Types").

3.6.1 How to Define a List of Values Global Bucketset

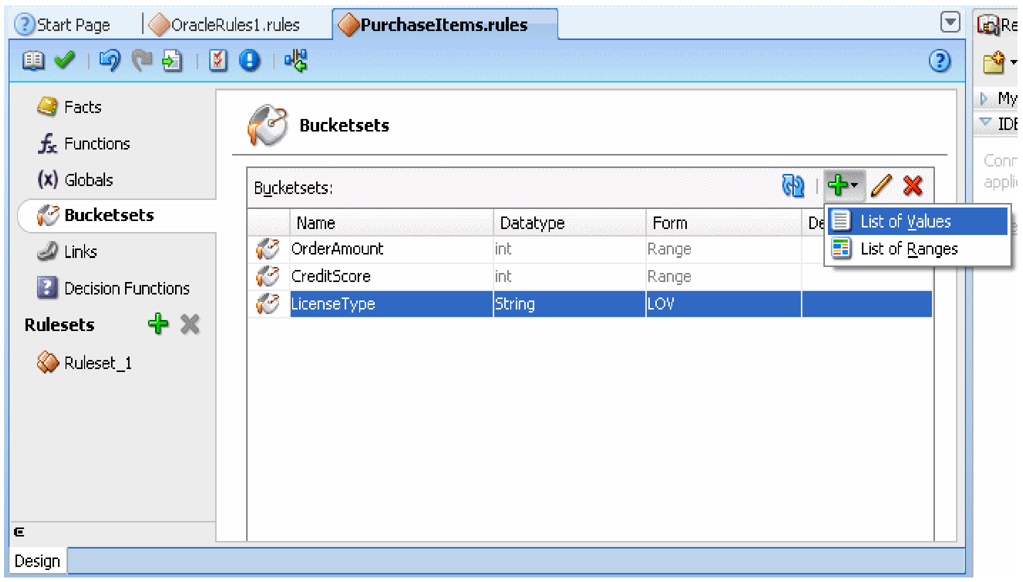
A list of values bucketset lets you specify the type and the list of buckets for the bucketset.

For more information, see Section 3.6.5, "What You Need to Know About List of Values Bucketsets".

To define a list of values (LOV) global bucketset:

1. From Rules Designer select the **Bucketsets** navigation tab.
2. From the dropdown list next to the **Create BucketSet...** icon, select **List of Values**, as shown in Figure 3–16.

Figure 3–16 Adding a List of Values Bucketset



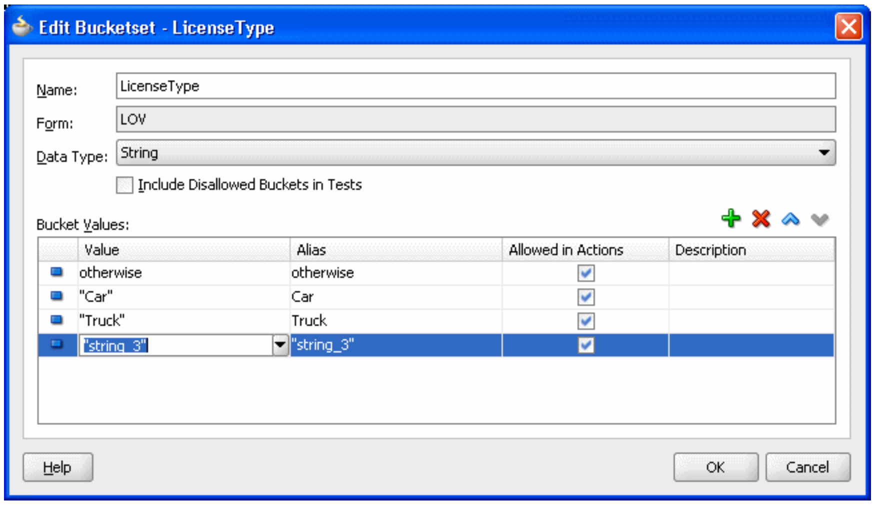
Name	Datatype	Form	De
OrderAmount	int	Range	
CreditScore	int	Range	
LicenseType	String	LOV	

3. Double-click the bucket icon for the bucket. This displays the Edit Bucketset dialog.
4. In the Edit Bucketset dialog, enter the bucketset name in the **Name** column.
5. In the **Data Type** column select a data type from dropdown list.

   For example, select **String** from the dropdown list.
6. Click the **Create** icon to add a value.
7. For each bucket that you add, do the following:
   - In the **Value** field, enter a value. Note that for String type values in an LOV bucket, you can select the entire string with three clicks. This allows you to enter the string and Rules Designer adds the same alias without quotation marks, as shown in Figure 3–17.
   - In the **Alias** field, enter an alias.
   - 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 Buckets in Tests** field, see Section 3.6.7, "What You Need to Know About Bucketset Allowed in Actions Option".
   - In the **Description** field, enter a description.
8. Add additional values by clicking the **Create** icon as needed for the bucketset, as shown in Figure 3–17.

Figure 3–17 Create List of Values Bucketset



Name: LicenseType
Form: LOV
Data Type: String
Include Disallowed Buckets in Tests

Bucket Values:

Value	Alias	Allowed in Actions	Description
otherwise	otherwise	✓	
"Car"	Car	✓	
"Truck"	Truck	✓	
"string_3"	"string_3"	✓	

9. On the Edit Bucketset window, click **OK**.

You can control rule ordering in a Decision Table by changing the relative position of the buckets in an LOV bucketset associated with a condition expression in a Decision Table.

To change the order of buckets in a list of values bucketset:

1. In the Edit Bucketset dialog for the bucketset, select the bucket you want to reorder.
2. Click the **Move Down** icon to reorder the bucket down.
3. Click the **Move Up** icon to reorder the bucket up.
4. Click **OK**.

3.6.2 How to Define a List of Ranges Global Bucketset

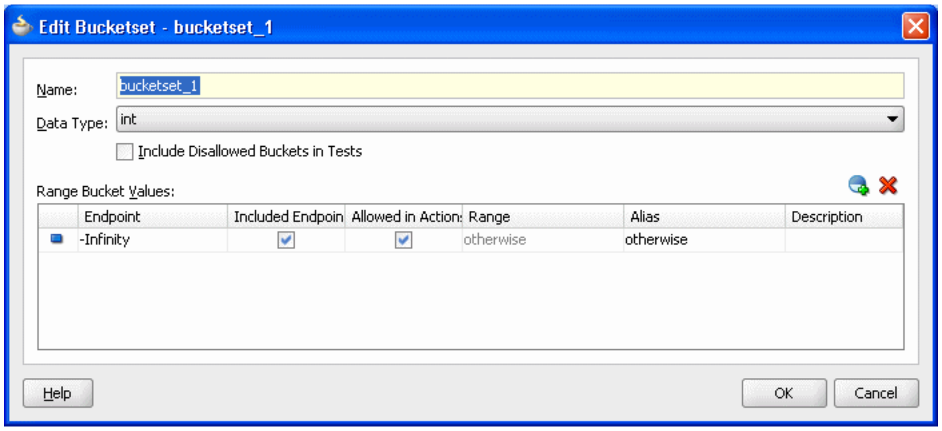
A list of ranges bucketset lets you specify the type and the endpoints for buckets in the bucketset.

For more information, see Section 3.6.6, "What You Need to Know About Range Bucketsets".

To define a list of ranges (range) global bucketset:

1. From Rules Designer select the **Bucketsets** navigation tab.
2. From the dropdown list next to the **Create BucketSet...** icon, select **List of Ranges**.
3. Double-click in the **Data Type** field. This displays the Edit Bucketset dialog, as shown in Figure 3–18.

Figure 3–18 Edit Bucketset: List of Ranges



Name: bucketset_1

Data Type: int

☐ Include Disallowed Buckets in Tests

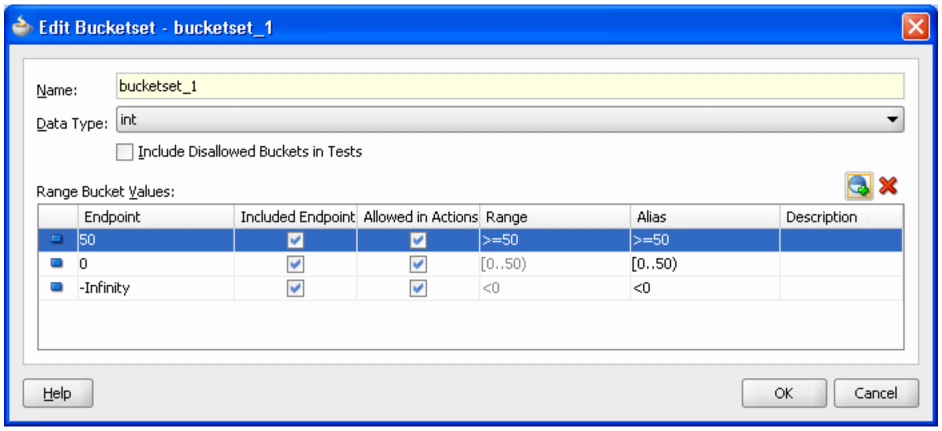
Range Bucket Values:

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
-Infinity	✓	✓	otherwise	otherwise	

4. In the Edit Bucketset dialog, enter the bucketset name in the **Name** field.
5. In the Edit Bucketset dialog, in the **Data Type** field, from the dropdown list, select the appropriate data type for the bucketset.

   In this example, select **int**.
6. Click the **Add Bucket** icon repeatedly to add the number of buckets you need in the bucketset as shown in Figure 3–19.

Figure 3–19 Edit Bucketset: Adding Required Buckets



Name: bucketset_1

Data Type: int

☐ Include Disallowed Buckets in Tests

Range Bucket Values:

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
50	✓	✓	>=50	>=50	
0	✓	✓	[0..50)	[0..50)	
-Infinity	✓	✓	<0	<0	

In these steps you add three buckets. You start with the default values, as shown in Figure 3–19. After changing the default buckets, the buckets have the following values:

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

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

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

   In this example, enter 1000 for the first bucket.

8. In the **Included Endpoint** field, select the checkbox as appropriate to include or exclude the bucket endpoint.

   In this example, you can leave this checkbox checked to include the bucket endpoint.

9. In the **Allowed in Actions** field select the checkbox as appropriate to include the bucket in the bucketset allowable values.

   For more information on the **Allowed in Actions** field and the **Include Disallowed Buckets in Tests** field, see Section 3.6.7, "What You Need to Know About Bucketset Allowed in Actions Option".

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

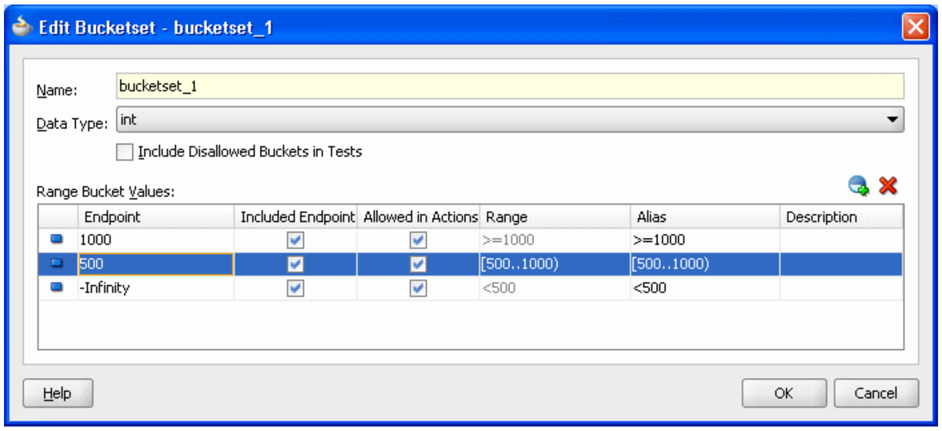
    The alias appears in Decision Tables that use this bucketset. Use an alias to give a more meaningful name to the bucket than the default value (the range-based Range value).

    The Range field is read-only: it clearly identifies the actual range associated with the bucket regardless of the Alias value. For more information, see Section 3.6.6, "What You Need to Know About Range Bucketsets").

11. Moving down the list of buckets, for each subsequent bucket, repeat from Step 7. For the second bucket, enter the endpoint value 500.

    Figure 3–20 shows the completed bucketset.

Figure 3–20 Edit Bucketset: Completed Range Buckets



Name: bucketset_1

Data Type: int

☐ Include Disallowed Buckets in Tests

Range Bucket Values:

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
1000	✓	✓	>=1000	>=1000	
500	✓	✓	[500..1000)	[500..1000)	
-Infinity	✓	✓	<500	<500	

12. In the Edit Bucketset dialog, click **OK**.

3.6.3 How to Define an Enumerated Type (Enum) Bucketset from XML Types

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

For more information, see Section 3.2.4, "What You Need to Know About XML Facts".

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

1. Obtain an XSD with the desired enumerations.

Example 3–2 shows the `order.xsd` schema file which contains the enumeration `Status`.

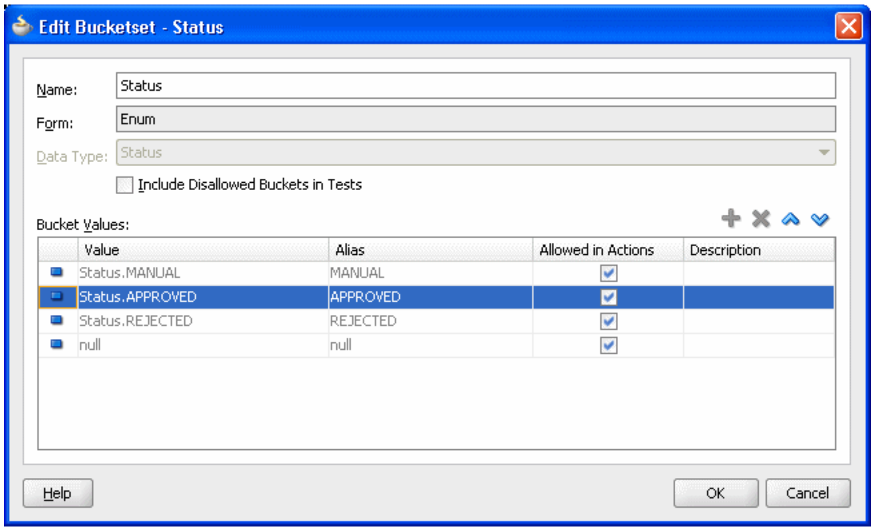
Example 3–2 Order.xsd Schema

```
<?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 Section 3.2, "Working with XML Facts".

3. Click the **Bucketsets** navigation tab and select the Enum bucketset to see the bucketset, as shown in Figure 3–21. In Figure 3–21, notice that the imported `Status` enumeration values shown in Example 3–2 are imported as buckets with the XSD-specified values.

Figure 3–21 Bucketset Showing the Form Enum with Imported Values



Name: Status
Form: Enum
Data Type: Status
Include Disallowed Buckets in Tests
Bucket Values:

	Value	Alias	Allowed in Actions	Description
	Status.MANUAL	MANUAL	✓	
	Status.APPROVED	APPROVED	✓	
	Status.REJECTED	REJECTED	✓	
	null	null	✓	

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

To change the order of buckets in an enum bucketset:

1. In the Edit Bucketset dialog for the bucketset, select the bucket you want to reorder.
2. Click the **Move Down** icon to reorder the bucket down.
3. Click the **Move Up** icon to reorder the bucket up.
4. Click **OK**.

3.6.4 How to Define an Enumerated Type (Enum) Bucketset from Java Types

When you import a Java enum, Rules Designer automatically creates an enumerated type bucketset for each Java enum. Although enumerated type bucketsets are read-only, you can change the order of buckets.

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

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

   Example 3–3 shows the `RejectPurchaseItem.java` class which contains enumeration `OrderSize`.

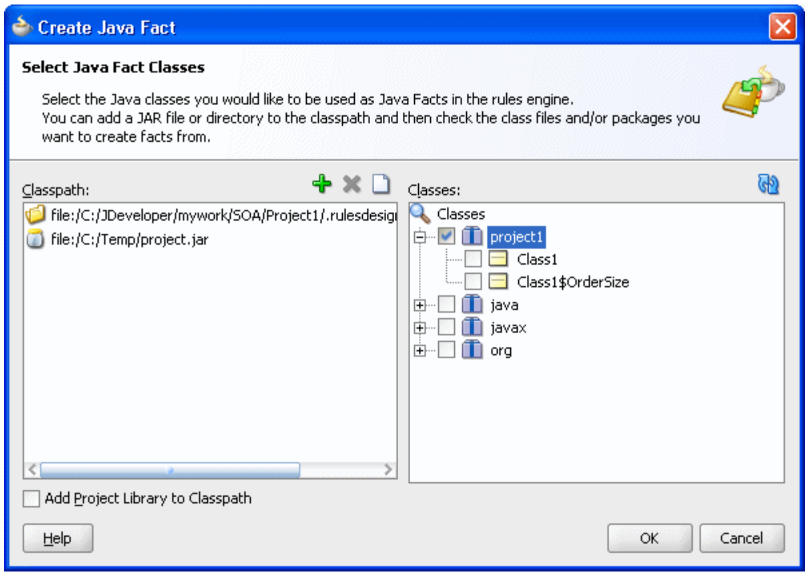
Example 3–3 Java Fact with enum OrderSize

```
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 Section 3.3, "Working with Java Facts".

Figure 3–22 shows a how to create a Java fact for the Java enumeration `Class1$OrderSize`.

Figure 3–22 Creating a Java Fact



Create Java Fact

Select Java Fact Classes

Select the Java classes you would like to be used as Java Facts in the rules engine.
You can add a JAR file or directory to the classpath and then check the class files and/or packages you want to create facts from.

Classpath:

- file:/C:/JDeveloper/mywork/SOA/Project1/.rulesdesig
- file:/C:/Temp/project.jar

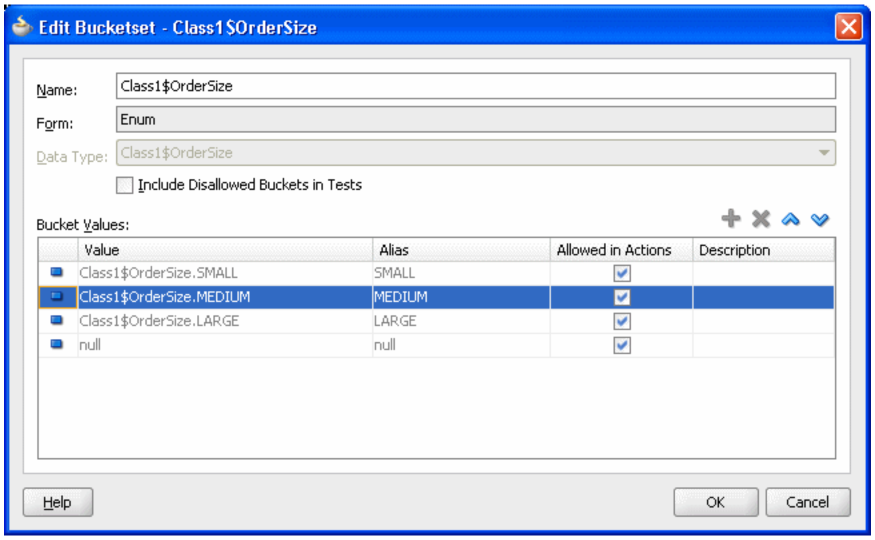
Classes:

- Classes
  - [✓] project1
    - [ ] Class1
    - [ ] Class1$OrderSize
  - [ ] java
  - [ ] javax
  - [ ] org

[ ] Add Project Library to Classpath

3. In Rules Designer click the **Bucketsets** navigation tab and select the Enum bucketset, as shown in Figure 3–23. Note that the `Class1$OrderSize` enumeration from the enumeration in Example 3–3 is now a bucketset with the Java `enum`-specified values.

Figure 3–23 Edit Bucketset Dialog for Java Enum



Edit Bucketset - Class1$OrderSize

Name: Class1$OrderSize

Form: Enum

Data Type: Class1$OrderSize

[ ] Include Disallowed Buckets in Tests

Bucket Values:

Value	Alias	Allowed in Actions	Description
Class1$OrderSize.SMALL	SMALL	✓	
Class1$OrderSize.MEDIUM	MEDIUM	✓	
Class1$OrderSize.LARGE	LARGE	✓	
null	null	✓	

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

To change the order of buckets in an enumerated type (enum) bucketset:

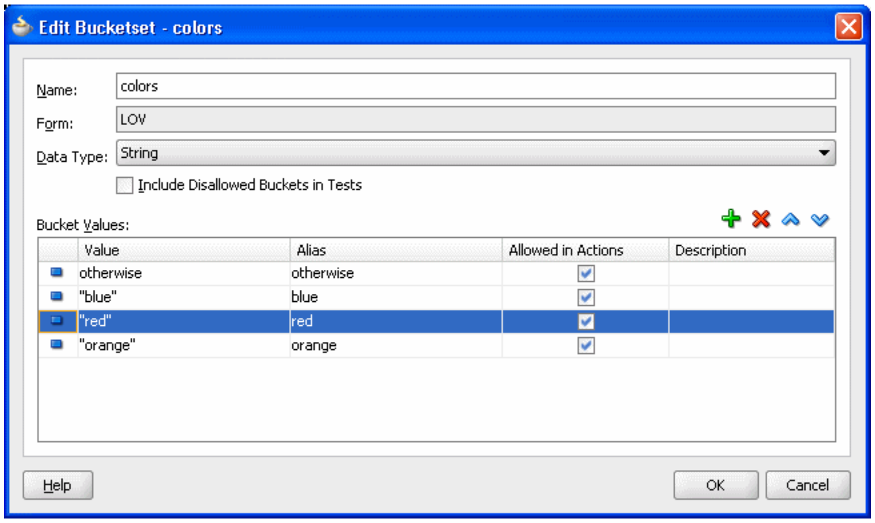
1. In the Edit Bucketset dialog for the bucketset, select the bucket you want to reorder.
2. Click the **Move Down** icon to reorder the bucket down.
3. Click the **Move Up** icon to reorder the bucket up.
4. Click **OK**.

3.6.5 What You Need to Know About List of Values Bucketsets

In a Decision Table the order of the buckets in a bucketset 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 buckets in a list of values bucketset associated with a condition expression; however, you cannot reorder range buckets.

Figure 3–24 shows a bucketset definition in Rules Designer for a bucketset named colors using a list of values.

Figure 3–24 Bucketset Definition Using List of Values



Name: colors

Form: LOV

Data Type: String

☐ Include Disallowed Buckets in Tests

Bucket Values:

Value	Alias	Allowed in Actions	Description
otherwise	otherwise	✓	
"blue"	blue	✓	
"red"	red	✓	
"orange"	orange	✓	

As shown in Figure 3–24, by default with a List of Values bucketset 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 bucket. Thus, with `otherwise` in the list of values a condition expression that uses the bucketset 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` bucket. The `otherwise` value cannot be removed from an LOV bucketset but it can be excluded by clearing the **Allowed in Actions** checkbox (when `otherwise` is excluded an attempt to assign any value that is not in the list of buckets in the bucketset causes a validation warning).

Table 3–3 shows the bucketset values that Rules Designer supports for LOV bucketsets.

Table 3–3 Supported Types for LOV Bucketsets

Type	Description
Java primitive types	This includes `int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, and `float`
`String`	Contains `String` types
`Calendar`	Contains `Calendar` types in the current locale

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

3.6.6 What You Need to Know About Range Bucketsets

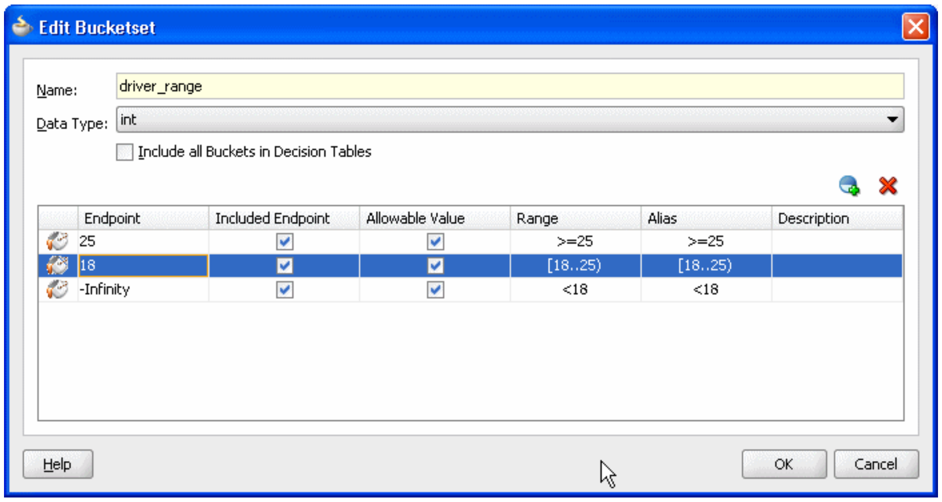
When you add a bucket to a List of Ranges bucketset, the value is calculated based on the currently selected bucket value and the next highest bucket value. When you change the endpoint value the value is automatically sorted in the bucketset; thus, it does not matter where a bucket is added. However,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

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

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

Figure 3–25 shows the Edit Bucketset window for a bucketset with an integer, `int`, range.

Figure 3–25 Bucketset Definition Using List of Ranges and Three Endpoints



Name: driver_range
Data Type: int
☐ Include all Buckets in Decision Tables

Endpoint	Included Endpoint	Allowable Value	Range	Alias	Description
25	✓	✓	>=25	>=25	
18	✓	✓	[18..25)	[18..25)	
-Infinity	✓	✓	<18	<18	

Table 3–4 shows the types Rules Designer supports for Range buckets.

Table 3–4 Supported Types for Range Buckets

Type	Description
Selected primitive types	This includes: `int`, `double`, `short`, `long`, and `float`

Table 3–4 (Cont.) Supported Types for Range Buckets

Type	Description
`Calendar`	Contains Calendar 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".

3.6.7 What You Need to Know About Bucketset Allowed in Actions Option

When you define buckets in a bucketset you might define some buckets corresponding to non-permissible values. For example, in a bucketset for driver ages you would typically not allow a bucket that contains values less than 0. Thus, when a fact with driver data includes an age property associated with a driver ages bucketset, 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 bucketset you select **Allowed in Actions** for valid buckets and deselect this option for invalid buckets.

The bucketset option **Include Disallowed Buckets in Tests** allows you to include all the buckets, whether **Allowed in Actions** is selected or not, in Decision Table conditions and in rule tests. By including all buckets you can explicitly test for illegal values. Using the option **Include Disallowed Buckets 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 deselect the **Include Disallowed Buckets 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 Section 4.11, "Using Bucketsets as Constraints for Options Values in Rules".
2. You want to consider excluded buckets in rule tests and in Decision Tables. In this case, you should select **Include Disallowed Buckets 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 bucketsets could include cases for excluded buckets 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 bucket in the bucketset, or, leave the **Allowed in Actions** field configured as is and select the **Include Disallowed Buckets in Tests** field. Using the **Include Disallowed Buckets in Tests** field is not only convenient, you do not need to reconfigure every bucket, it also preserves the configuration of **Allowed in Actions** so that you can easily reuse this bucketset to handle the first case (when you deselect **Include Disallowed Buckets in Tests**).

3.6.8 What You Need to Know About Bucket Values

When you enter a bucket value in a bucketset, the value you supply must be valid for the type specified for the bucketset. If the value you enter is not valid for the bucketset 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 bucketset, and then supply a value 2.2 to a bucket, Rules Designer shows a warning such as the following:

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

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

For an additional example, when you enter a value for a bucket, for example if you enter a bucket value with bucketset with data type short and add a bucket 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 bucket value, similar to the previous example, because a String is not a valid bucket value for a bucketset with data type short. The solution to this is to enter appropriate values for all buckets (in this example, enter a value less than or equal to 32767).

3.7 Associating a Bucketset with Facts and Functions

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

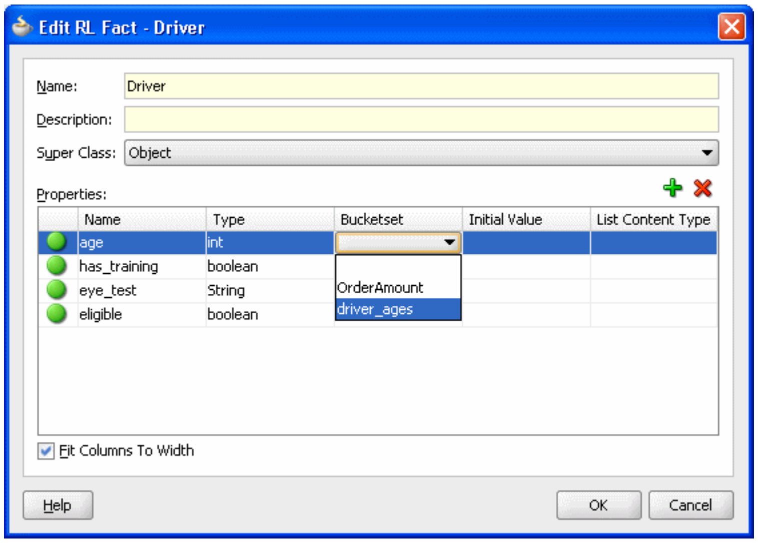
3.7.1 How to Associate a Bucketset with a Fact Property

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

To associate a bucketset with a fact property:

1. From Rules Designer select the **Facts** navigation tab.
2. Select the fact type you are interested. This displays the appropriate Edit Fact dialog for the fact type you select.
3. In the Properties table, under **Bucketset**, select the cell for the appropriate fact property and from the dropdown list select the bucketset you want to associate with the property. For example, see Figure 3–26.

Figure 3–26 Defining a Bucketset for a Property



Name: Driver

Description:

Super Class: Object

Properties:

Name	Type	Bucketset	Initial Value	List Content Type
age	int			
has_training	boolean			
eye_test	String			
eligible	boolean			

Bucketset dropdown: OrderAmount, driver_ages

☑ Fit Columns To Width

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

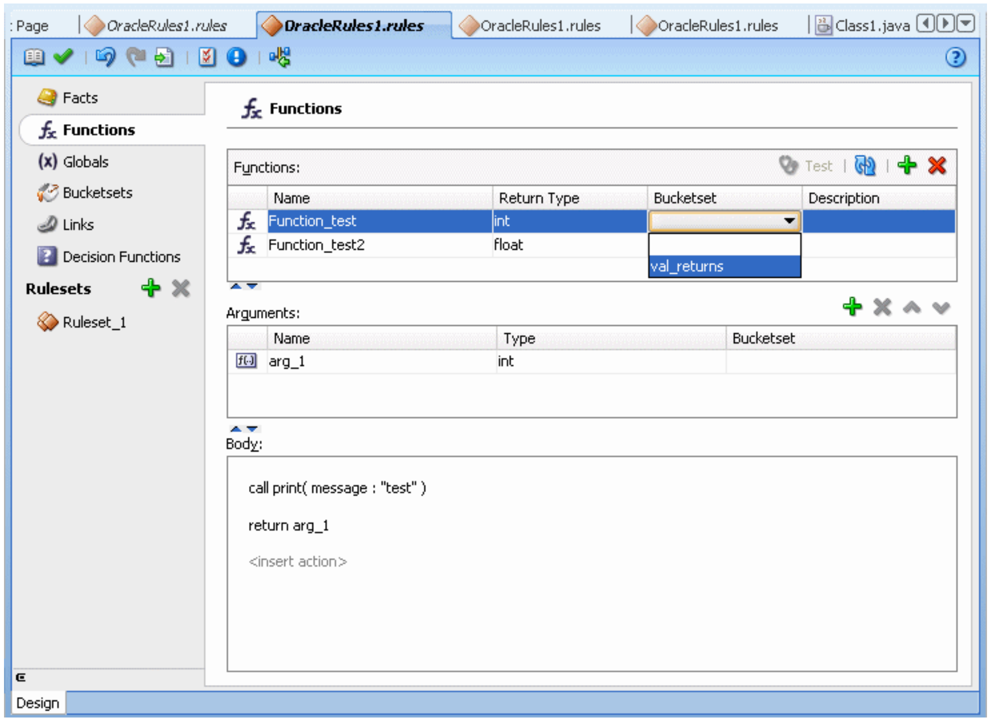
3.7.2 How to Associate a Bucketset with Functions or Function Arguments

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

To associate a bucketset 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 **Bucketset**, select the cell and from the dropdown list select the bucketset you want to use. For example, see Figure 3–27.

Figure 3–27 Defining a Bucketset for a Function Return Value



	Name	Return Type	Bucketset	Description
	Function_test	int		
	Function_test2	float	val_returns	

	Name	Type	Bucketset
	arg_1	int	

```
call print( message : "test" )

return arg_1

<insert action>
```

To associate a bucketset 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 **Bucketset**, select the cell and from the dropdown list select the bucketset you want to use.

4 Working with Rulesets and Rules

A ruleset is an Oracle Business Rules object that you use to group one or more rules and Decision Tables.

This chapter includes the following sections:

For more information, see Section 1.1.5, "What Are Rulesets?".

4.1 Introduction to Working with Rulesets and Rules

You can use business rules to define key decisions and policies for a business, including:

- 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 two ways to work with rules:

- Using IF/THEN rules
- Using rules in a Decision Table

This chapter describes working with IF/THEN rules. For information on Decision Tables, see Chapter 5, "Working with Decision Tables".

4.2 Working with Rulesets

A ruleset provides a unit of execution for rules and for 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 the stack. In rulesets, the priority of rules applies to specify the order of firing of the rules in the ruleset. Rulesets also provide an effective date specification that identifies that the ruleset is always active, or that the ruleset is restricted based on a time and date range, or a starting or ending time and date.

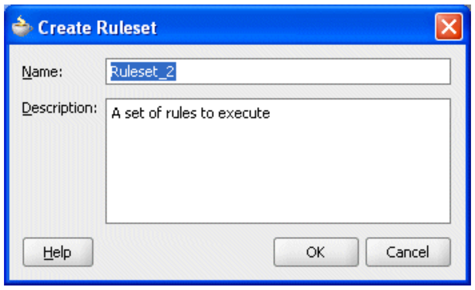
4.2.1 How to Create a Ruleset

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 ruleset:

1. In Rules Designer, go to the **Rulesets** navigation tab.
2. Click the **Create Ruleset...** icon. This displays the Create Ruleset dialog.
3. Enter a name in the **Name** field.
4. Enter a description in the **Description** field, as shown in Figure 4–1.

Figure 4–1 Adding a Ruleset



Name: Ruleset_2
Description: A set of rules to execute

5. Click **OK**.

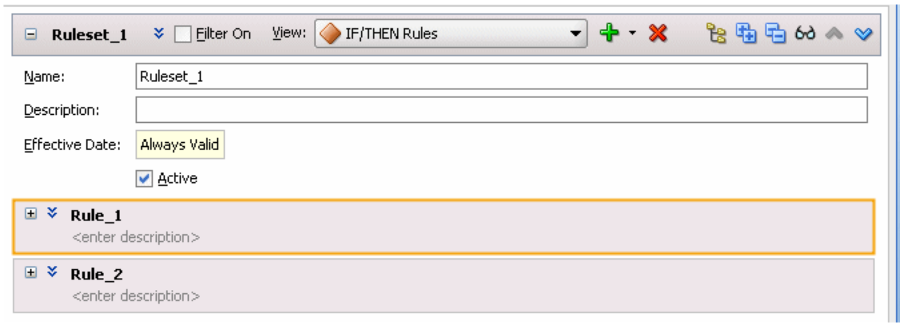
4.2.2 How to Set the Effective Date for a Ruleset

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 Section 4.9, "Using Date Facts, Date Functions, and Specifying Effective Dates".

To set the effective date for a ruleset:

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

Figure 4–2 Ruleset Showing Effective Date Field



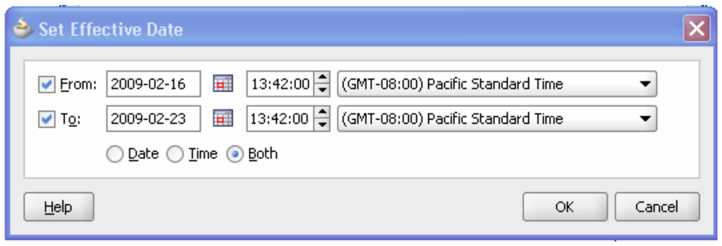
Name: Ruleset_1
Description:
Effective Date: Always Valid
☑ Active

Rule_1
<enter description>

Rule_2
<enter description>

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

Figure 4–3 Using the Set Effective Date Dialog



☑ From: 2009-02-16 13:42:00 (GMT-08:00) Pacific Standard Time
☑ To: 2009-02-23 13:42:00 (GMT-08:00) Pacific Standard Time
○ Date ○ Time ● Both

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

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

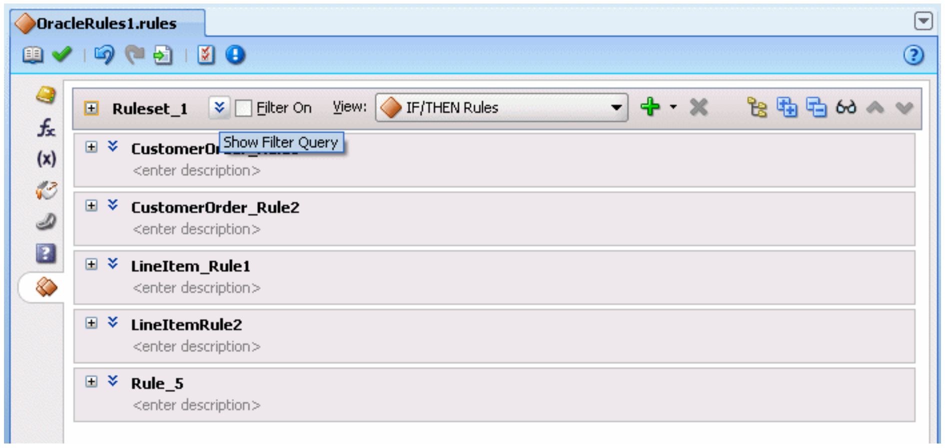
As the number of rules in a ruleset increases, it can be difficult to navigate the list of rules. You can instruct Rules Designer to filter the list of rules, to display only rules of interest. For example, you can display only active rules or only rules that have validation warnings.

For more information on creating rules, see Section 4.3, "Working with Rules".

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

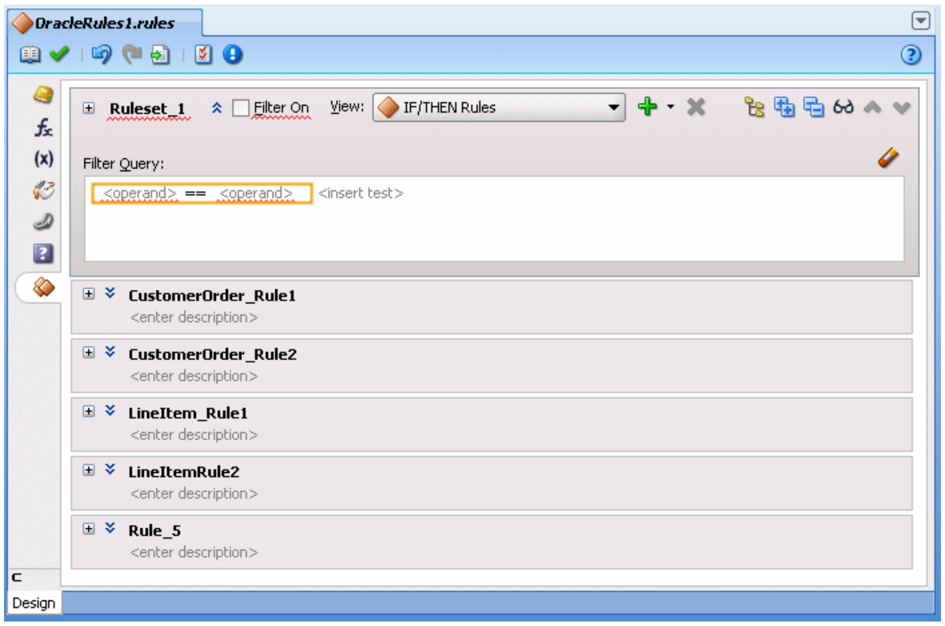
1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. To show the rule filter settings, next to the ruleset name, click **Show Filter Query** as Figure 4–4 shows.

Figure 4–4 Showing a Filter Query in a Ruleset



3. In the **Filter Query** field, click **<insert test>** to insert a default test as Figure 4–5 shows.

Figure 4–5 Inserting a Default Filter Query Test



4. Configure the default test.

   In this case, as shown in Figure 4–6, when you click an **<operand>** you can choose from the rule-specific options shown in Table 4–1.

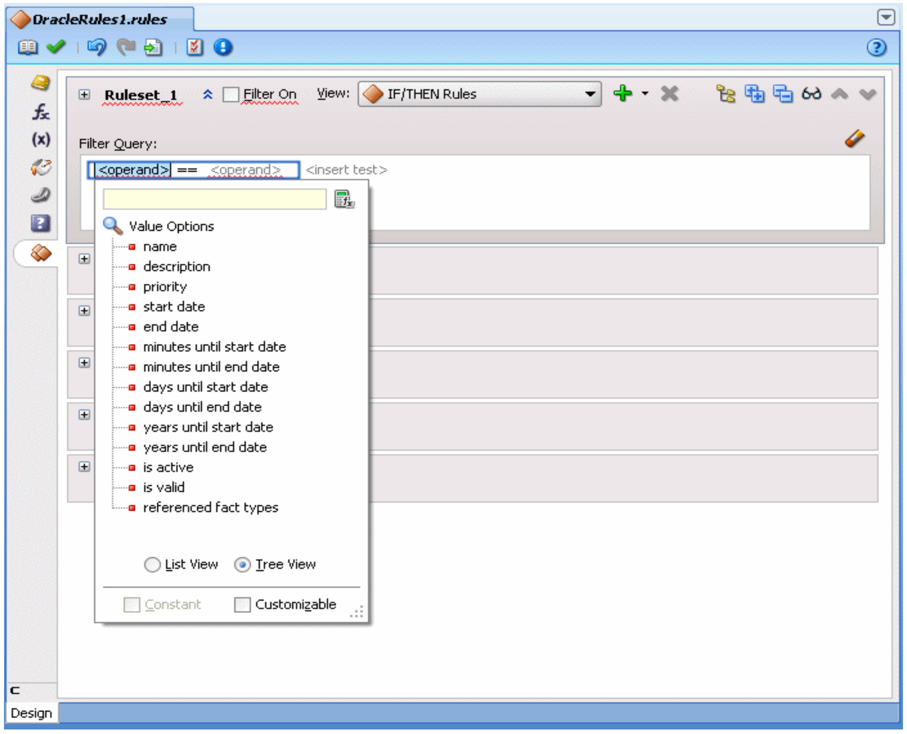
Table 4–1 Rule Filter Query Operands

Operand	Description
`name`	Matches against the rule name.
`description`	Matches against the rule description.
`priority`	Matches against the rule priority. For more information, see Section 4.5.5, "How to Set a Priority for a Rule".
`start date`	Matches against the rule start date. For more information, see Section 4.9.2, "How to Set the Effective Date for a Rule".

Table 4–1 (Cont.) Rule Filter Query Operands

Operand	Description
`end date`	Matches against the rule end date. For more information, see Section 4.9.2, "How to Set the Effective Date for a Rule".
`minutes until start date`	Matches against a specified number of minutes until the rule start date. For more information, see Section 4.9.2, "How to Set the Effective Date for a Rule".
`minutes until end date`	Matches against a specified number of minutes until the rule end date. For more information, see Section 4.9.2, "How to Set the Effective Date for a Rule".
`days until start date`	Matches against a specified number of days until the rule start date. For more information, see Section 4.9.2, "How to Set the Effective Date for a Rule"
`days until end date`	Matches against a specified number of days until the rule end date. For more information, see Section 4.9.2, "How to Set the Effective Date for a Rule"
`years until start date`	Matches against a specified number of years until the rule start date. For more information, see Section 4.9.2, "How to Set the Effective Date for a Rule"
`years until end date`	Matches against a specified number of years until the rule end date. For more information, see Section 4.9.2, "How to Set the Effective Date for a Rule"
`is active`	Matches against whether the rule is active. For more information, see Section 4.5.3, "How to Select the Active Option".
`is valid`	Matches against whether the rule has validation warnings. For more information, see Section 4.4.2, "Understanding Rule Validation".
`referenced fact types`	Matches against one or more fact types.

Figure 4–6 Filter Query Operands

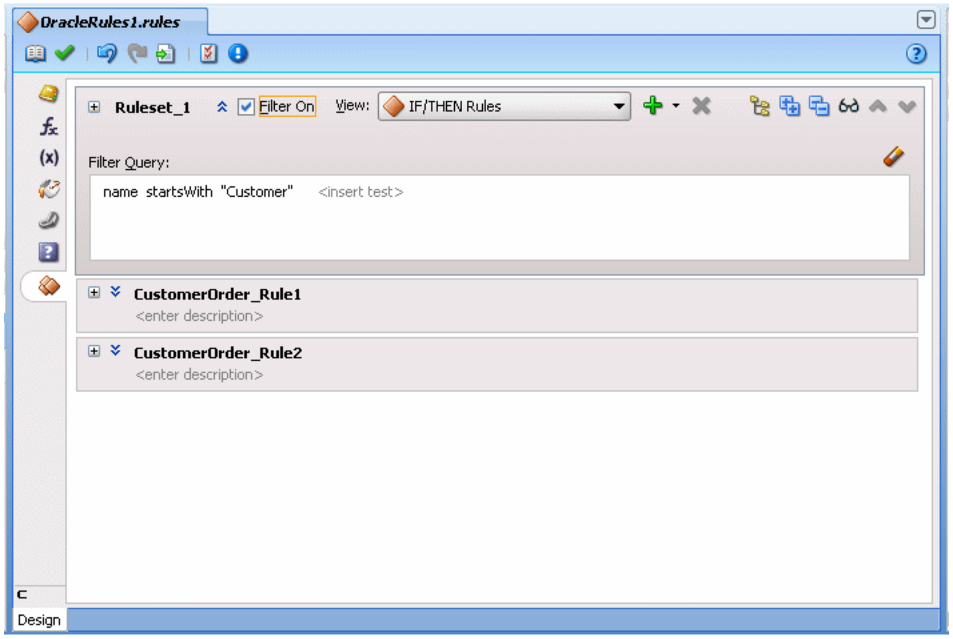


For more information, see Section 4.3.2, "How to Define a Test in a Rule".

5. Select the operator to choose an operator for the comparison. For example, for the name you can select **startsWith** from the operand list.
6. Enter a comparison operand for the right-hand-side of the filter test. For example, enter the string `Customer`.
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** checkbox.

      Rules Designer displays only the rules that match the filter query as Figure 4–7 shows.

Figure 4–7 Enable Filter Query in a Ruleset with Filter On Option



   b. To disable the filter query, deselect the **Filter On** checkbox.

      Rules Designer displays all the rules in the ruleset.

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

4.3 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 need to create a ruleset (or use the default ruleset). For more information on creating a ruleset, see Section 4.2, "Working with Rulesets".

You can easily test your rules as you are designing them without having to deploy your application. For more information, see Section 8.1.1, "How to Test Rules Using a Test Function in Rules Designer".

Rules Designer rule validation can assist you when you work with rules. To show the validation log window, click the **Validate** icon 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. For more information on rule validation, see Section 4.4.2, "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 Section 4.2.3, "How to Use a Filter to Display Matching Rules in a Ruleset".

4.3.1 How to Add Rules

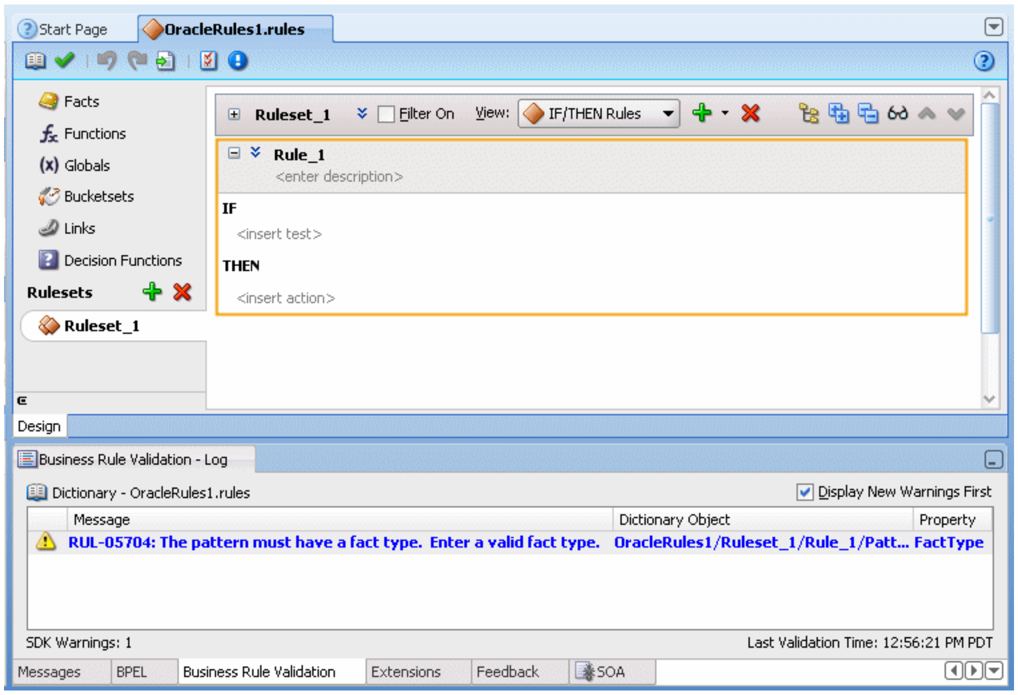
To create a rule you first add the rule to a ruleset, and then you insert tests and actions. The actions are associated with pattern matches. At runtime when a test in the **IF** area of a rule matches, the Rules Engine activates the **THEN** action and prepares to run the actions associated with the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

Rules Designer lets you create a rule where by default the rule fires for each matching fact. To enable other options, where the same fact type matches more than once, or never, you select **Advanced Mode**. For more information on advanced mode and showing advanced settings, see Section 4.5, "Using Advanced Settings with Rules and Decision Tables".

To add rules in a ruleset:

1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. In the **View** field, select **IF/THEN Rules**.
3. Click **Add** to add a rule. For example, click **Add** to add a rule named `Rule_1`, as shown in Figure 4–8.

Figure 4–8 Adding a Rule in a Ruleset



Rule_1
<enter description>
IF
<insert test>
THEN
<insert action>

Message	Dictionary Object	Property
RUL-05704: The pattern must have a fact type. Enter a valid fact type.	OracleRules1/Ruleset_1/Rule_1/Patt...	FactType

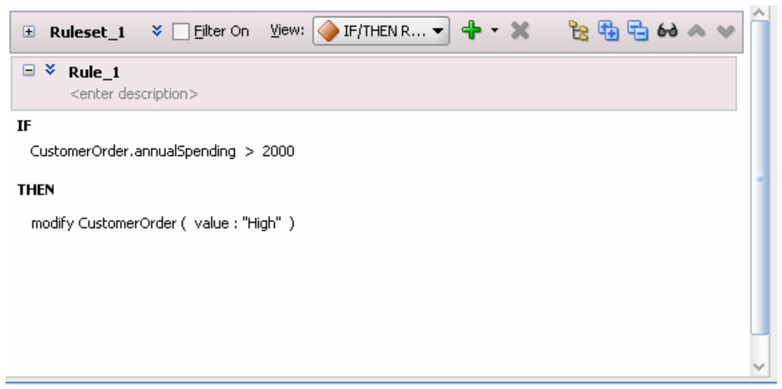
SDK Warnings: 1
Last Validation Time: 12:56:21 PM PDT

4.3.2 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 bucketsets to limit the values for tests and actions in rules. For more information, see Section 4.11, "Using Bucketsets as Constraints for Options Values in Rules".

Figure 4–9 shows this sample rule.

Figure 4–9 Adding a Test to a Rule



IF

CustomerOrder.annualSpending > 2000

THEN

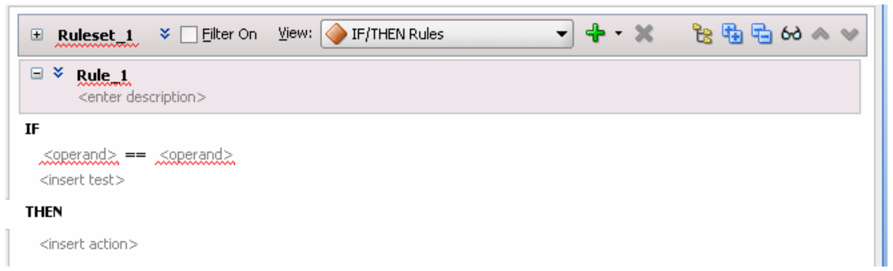
modify CustomerOrder (value : "High")

At runtime, when this rule is processed the Rules Engine checks the facts against rule pattern tests that you define to find matching facts. 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, 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. For a test, the **IF** area of a rule includes a left-hand-side `<operand>` and a right-hand-side `<operand>`, as shown in Figure 4–10.

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



IF

<operand> == <operand>

<insert test>

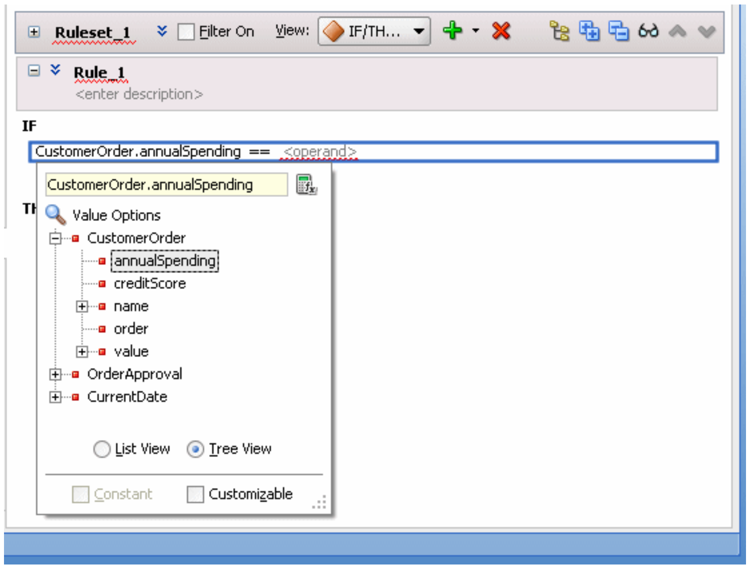
THEN

<insert action>

6. 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 dropdown list, as shown in Figure 4–11:

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



CustomerOrder.annualSpending == <operand>

CustomerOrder.annualSpending

Value Options
- CustomerOrder
  - annualSpending
  - creditScore
  - name
  - order
  - value
- OrderApproval
- CurrentDate

a. To enter a value use the dropdown 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**.

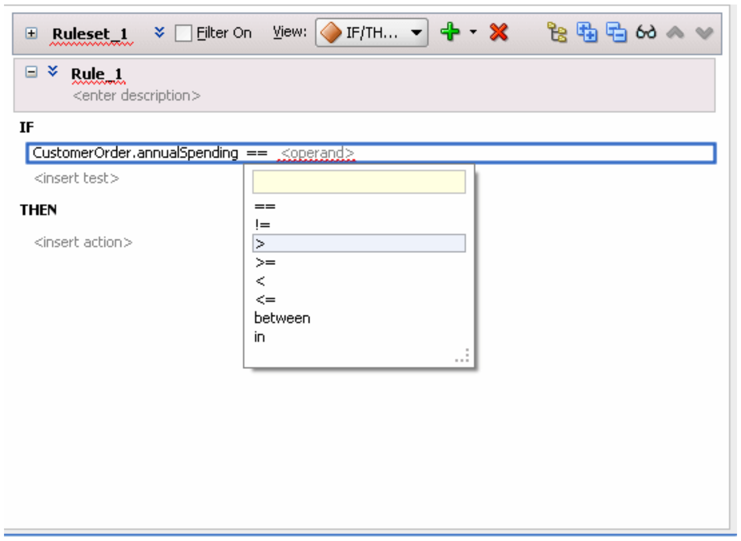
b. To enter a literal value, type the value into the text entry area and press **Enter**.

   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 test, you replace the operator with the desired logical operator or accept the default (==). To do this, select the default **==** operator. This displays a text entry area and a dropdown list, as shown in Figure 4–12.

   To test a logical condition between the left-hand and right-hand operands, select one of the logical operators as shown in Figure 4–12: == (equality), != (not equal), > (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to).

Figure 4–12 Configuring the Operator of a Test in a Rule

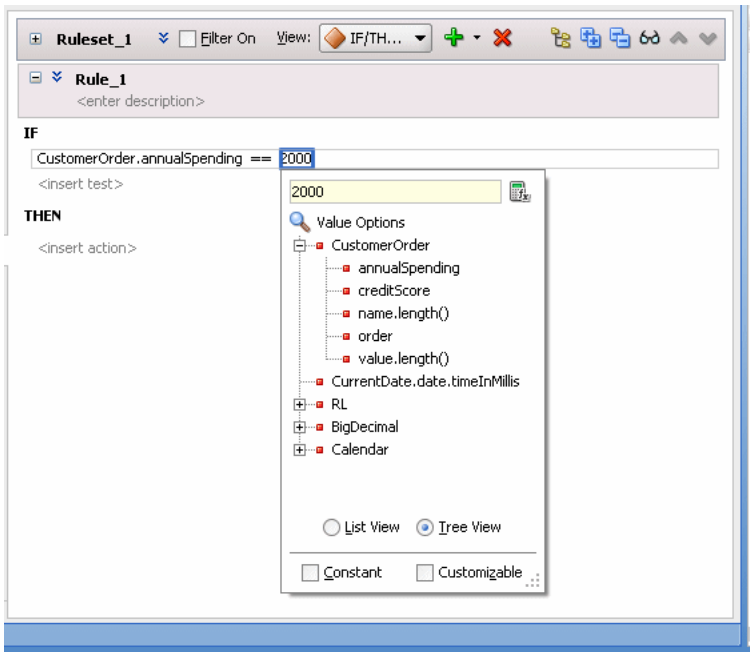


8. In a test, you replace the right-hand-side operand with a value.

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

   For example, enter `2000` 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



4.3.3 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:

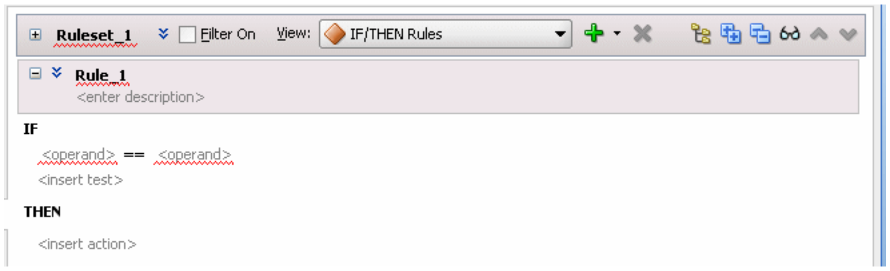
```
IF
     CustomerOrder.annualSpending between 100 and 2000
THEN
     Modify CustomerOrder.value = "Normal"
```

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>**, as shown in Figure 4–14.

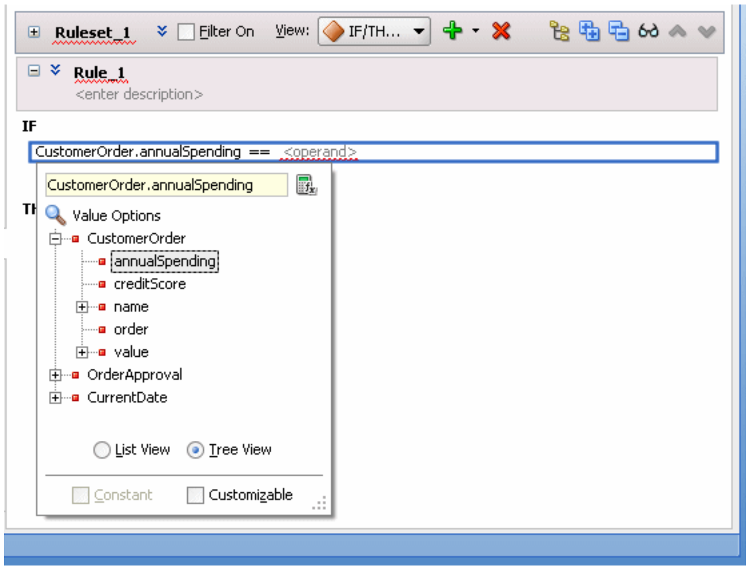
Figure 4–14 Rule Test with Left-hand-side operand and Right-hand-side operand



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 dropdown list, as shown in Figure 4–15:

Figure 4–15 Adding a Test Left-hand-side Operand to a Rule



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

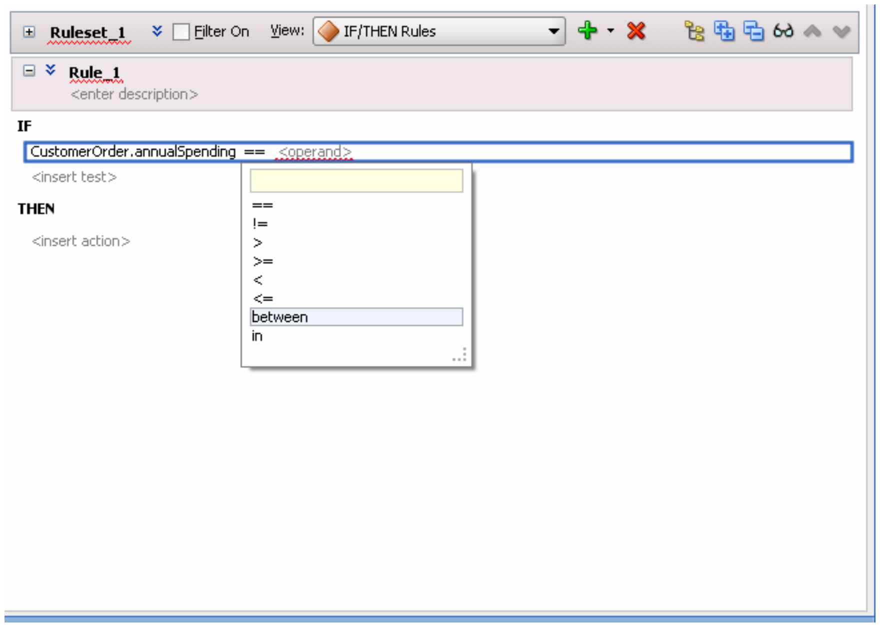
   You can view the options using a single list, by selecting **List View**, or using a navigator by selecting **Tree View**.

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`.

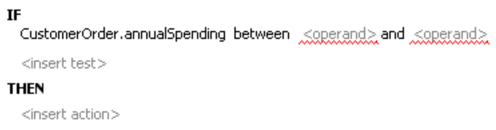
7. In a range test, you choose the `between` operator. To do this, select the default **==** operator. This displays a text entry area and a dropdown list. Select **between** as shown in Figure 4–16.

Figure 4–16 Configuring the Operator of a Range Test in a Rule



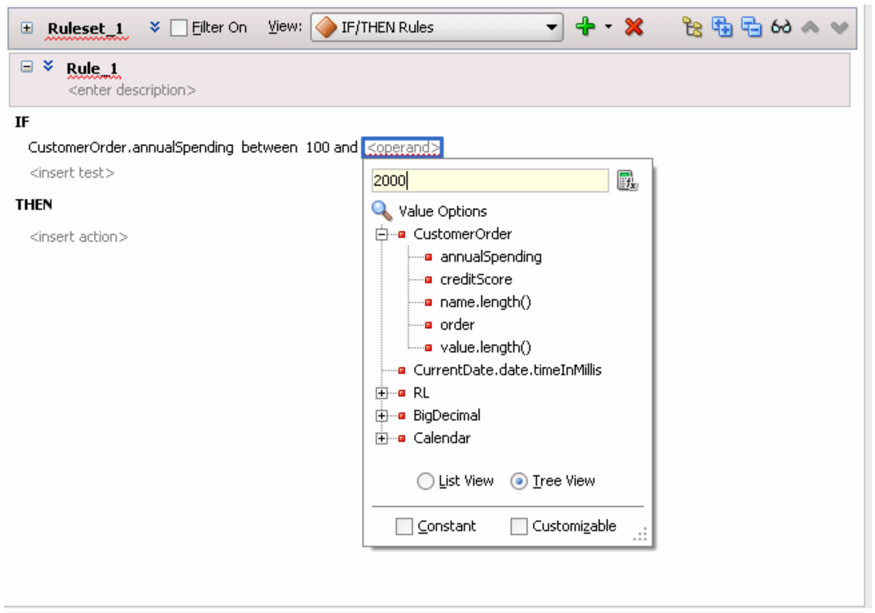
This adds two more **<operand>** placeholders as shown in Figure 4–17.

Figure 4–17 Between Operator in a Range Test


IF
CustomerOrder.annualSpending between <operand> and <operand>
<insert test>
THEN
<insert action>

8. Configure the **<operand>** placeholders as you would for any operand as shown in Figure 4–18.

Figure 4–18 Configuring the Operand of a Range Test in a Rule



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

4.3.4 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:

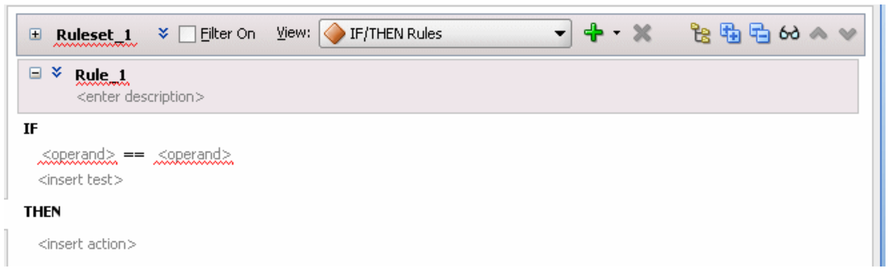
```
IF
     CustomerOrder.lineItem.sku in 12345, 43255, 76348
THEN
     Modify CustomerOrder.value = "High"
```

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>**, as shown in Figure 4–10.

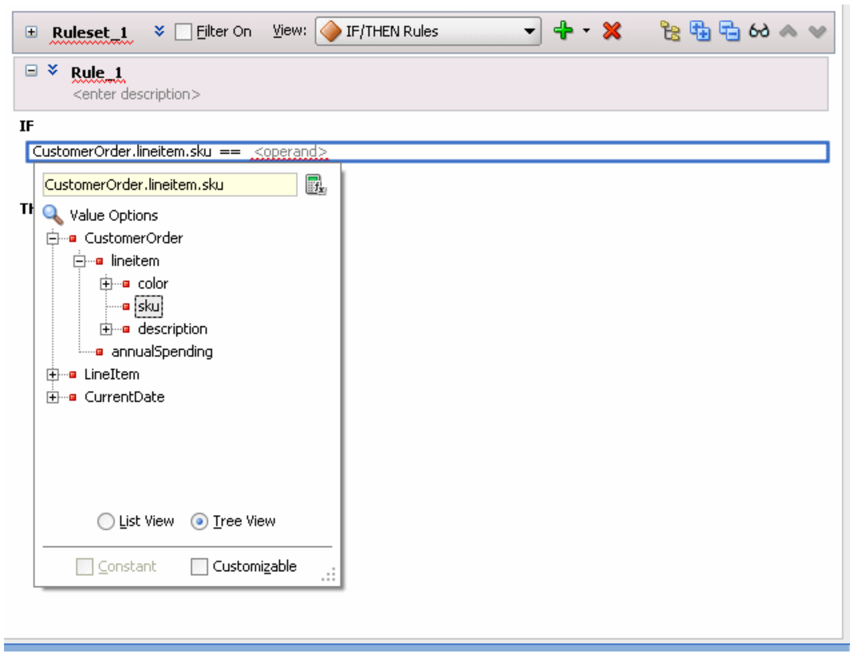
Figure 4–19 Rule Test with Left-hand-side operand and Right-hand-side operand



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 dropdown list as shown in Figure 4–20:

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



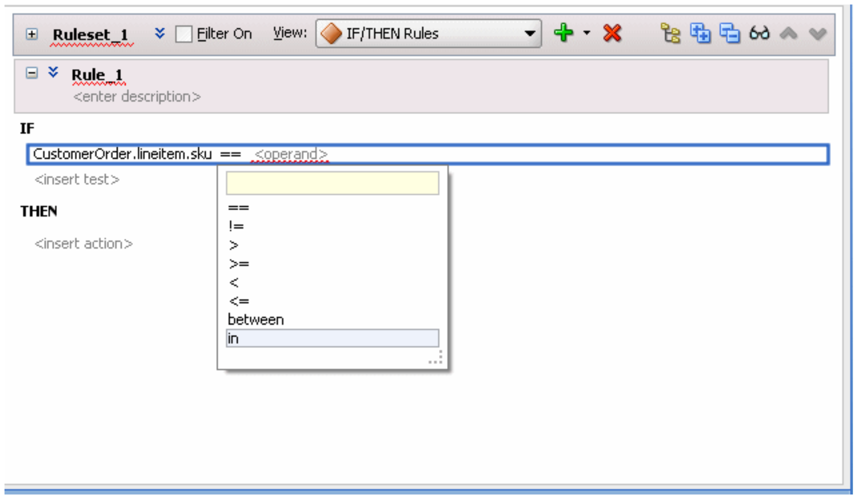
   a. To enter a value use the dropdown list to select an item from the value options.

      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**.

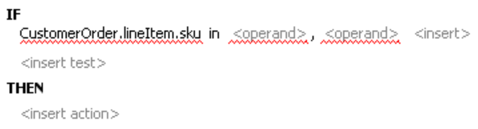
7. In a set test, you use the `in` operator. To do this, select the default == operator. This displays a text entry area and a dropdown list. Select **in** as shown in Figure 4–21.

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

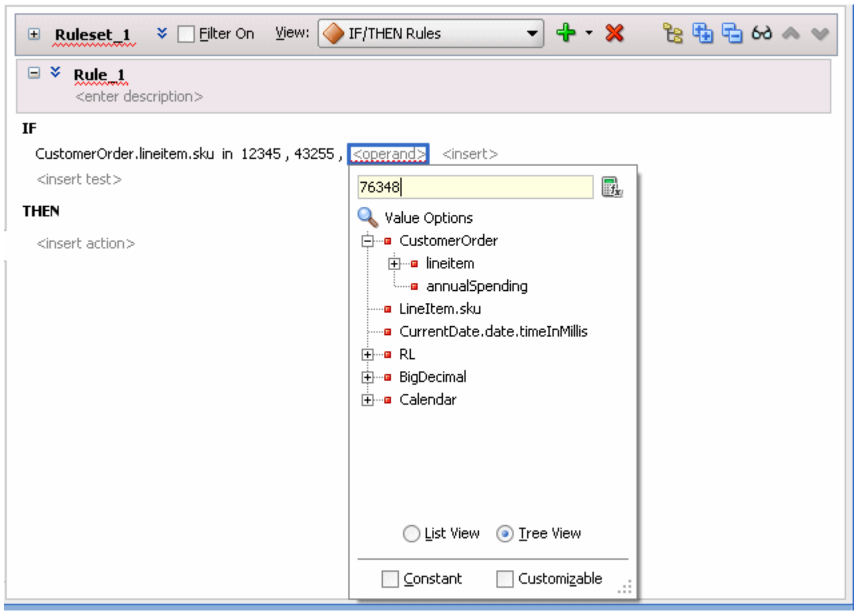

IF
CustomerOrder.lineItem.sku in <operand> , <operand> <insert>
<insert test>
THEN
<insert action>

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



IF
CustomerOrder.lineitem.sku in 12345 , 43255 , <operand> <insert>
<insert test>
THEN
<insert action>

76348

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

4.3.5 How to Define Actions in Rules

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 Section 4.5, "Using Advanced Settings with Rules and Decision Tables".

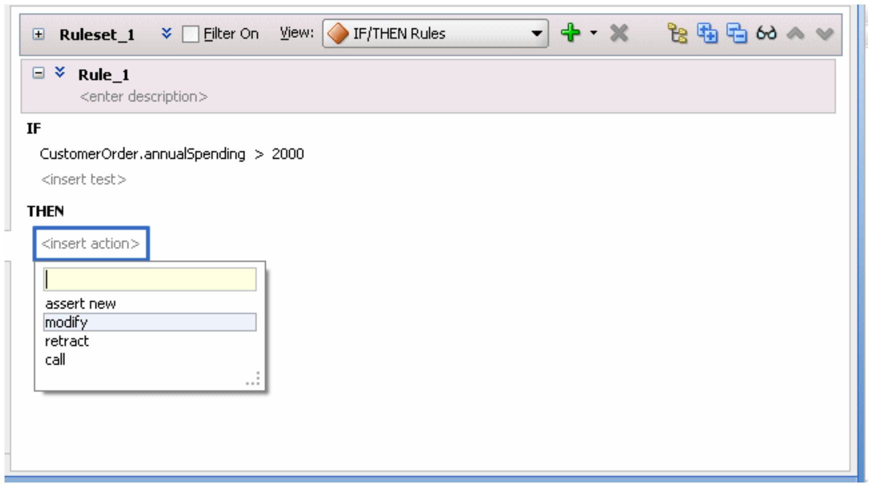
Table 4–2 Rule Action Choices

Action Form	Description
`Assert New`	Assert a new fact
`Modify`	Modify a data value associated with a matched fact
`Retract`	Retract a fact
`Call`	Call a function

To define actions in rules:

1. In Rules Designer, select a ruleset from the **Rulesets** navigation tab.
2. In a rule, in the **THEN** area, select **<insert action>**. This displays the add action list as shown in Figure 4–24.

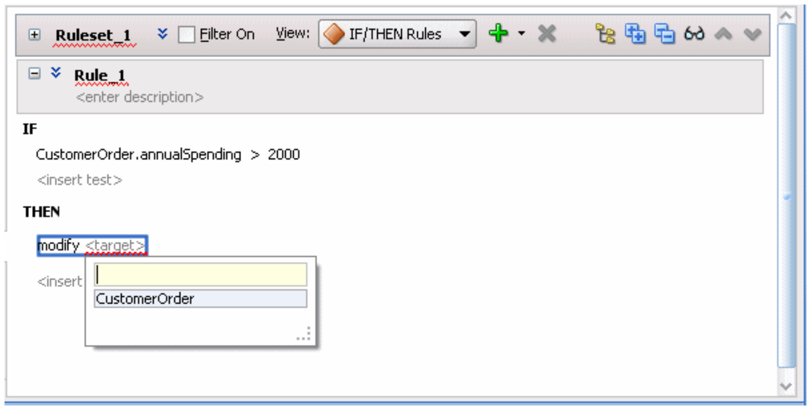
Figure 4–24 Adding a Modify Action to a Rule



3. In the add action list, select the type of action you want to add. For example, select **modify**.

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

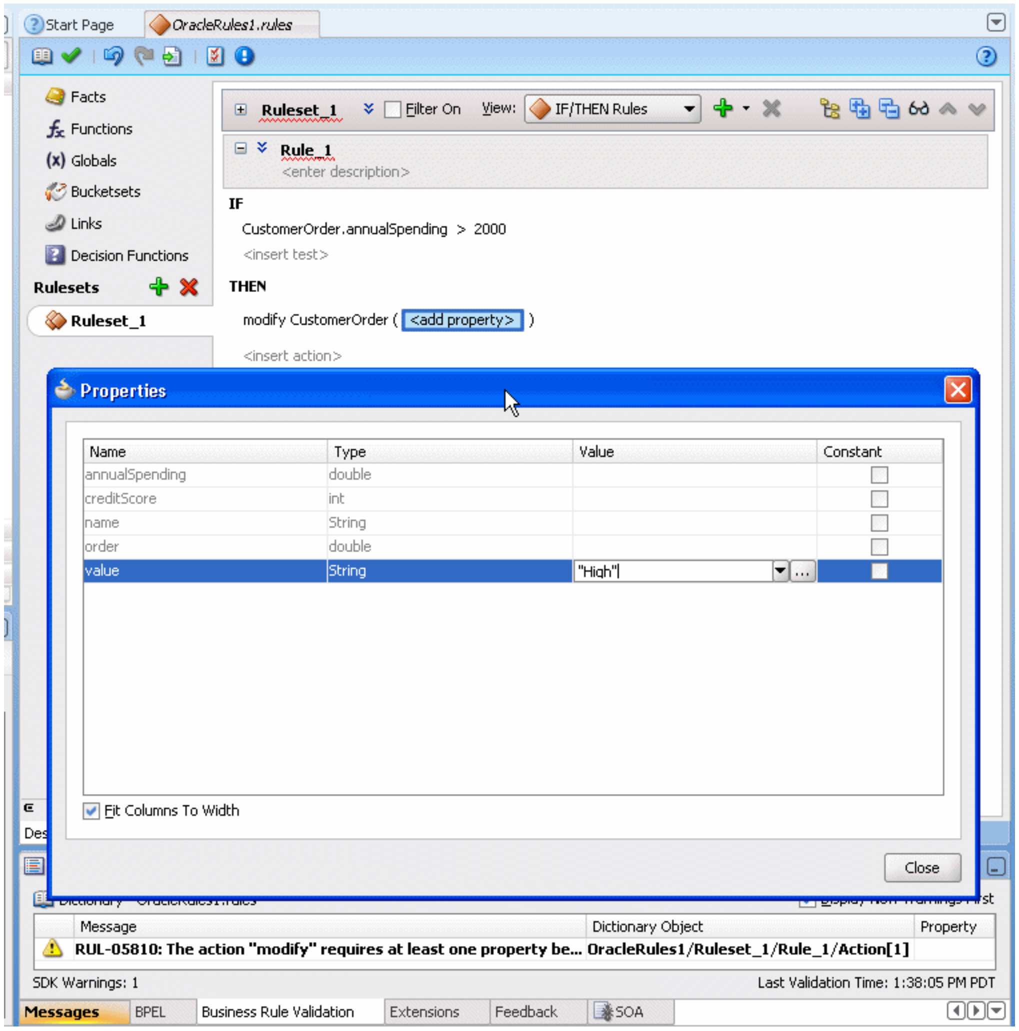
Figure 4–25 Adding Modify Action to a Rule and Selecting the Target



5. Select **<add 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



```
Ruleset_1
Rule_1
<enter description>
IF
CustomerOrder.annualSpending > 2000
<insert test>
THEN
modify CustomerOrder ( <add property> )
<insert action>
```

Name	Type	Value	Constant
annualSpending	double		☐
creditScore	int		☐
name	String		☐
order	double		☐
value	String	"High"	☐

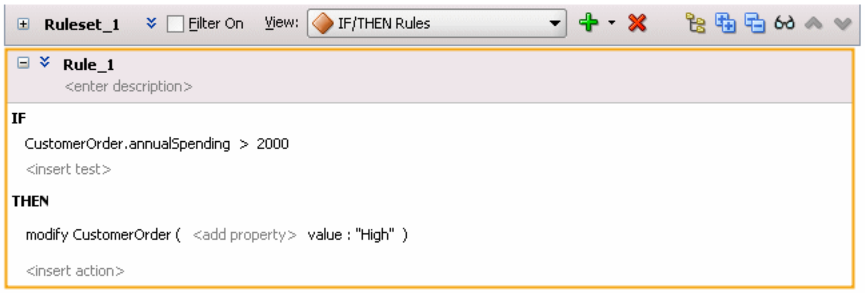
☑ Fit Columns To Width

	Message	Dictionary Object	Property
⚠	RUL-05810: The action "modify" requires at least one property be...	OracleRules1/Ruleset_1/Rule_1/Action[1]	

SDK Warnings: 1   Last Validation Time: 1:38:05 PM PDT

7. In the Properties dialog, click **Close**. This displays the rule as shown in Figure 4–27.

Figure 4–27 Rule with Test and Action Added



```
Ruleset_1
Rule_1
<enter description>
IF
CustomerOrder.annualSpending > 2000
<insert test>
THEN
modify CustomerOrder ( <add property> value : "High" )
<insert action>
```

4.3.6 What You Need to Know About Rule Actions

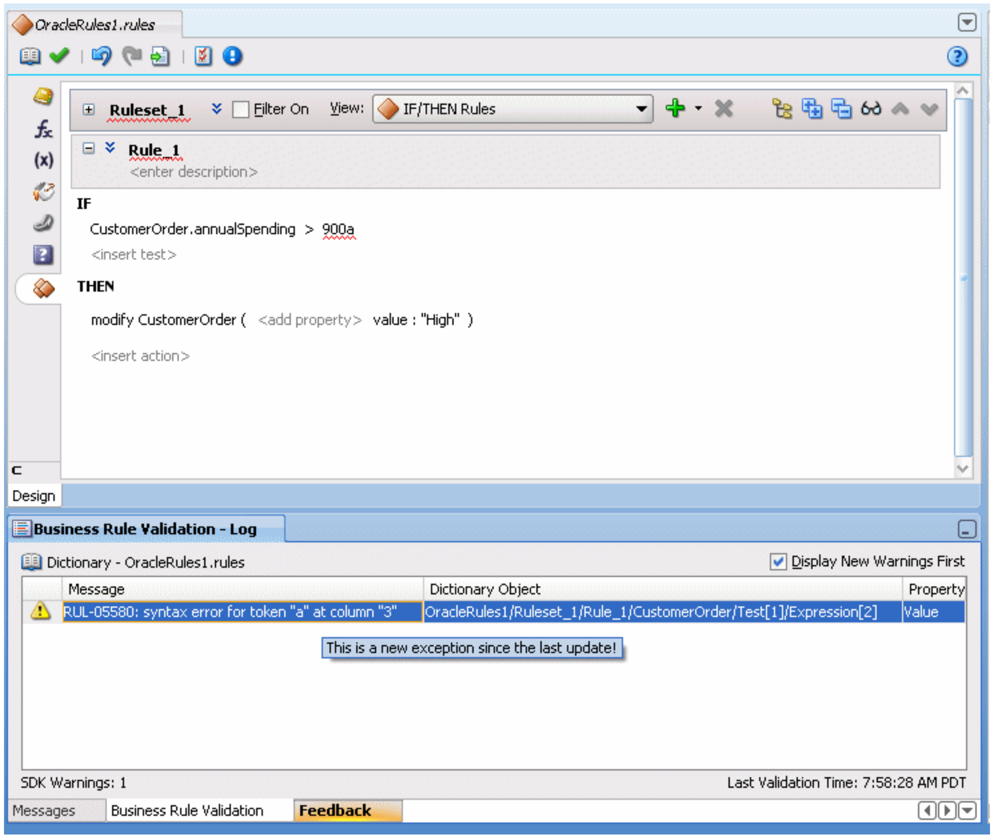
A rule loop occurs when the value for a condition is changed by an action. Loops can occur across rules in a single rule, spread over several Decision Tables, or spread over rules and Decision Tables in the same ruleset. You need to avoid creating rule actions that modify fact properties that are used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

4.4 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** icon 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 4–28.

Figure 4–28 Validation Warnings Shown in Log and On Screen with Wavy Underline


OracleRules1.rules
Ruleset_1
Filter On
View:
IF/THEN Rules
Design
Business Rule Validation - Log
Dictionary - OracleRules1.rules
Display New Warnings First
SDK Warnings: 1
Last Validation Time: 7:58:28 AM PDT
Messages
Business Rule Validation
Feedback

```
Rule_1
<enter description>
IF
  CustomerOrder.annualSpending > 900a
  <insert test>
THEN
  modify CustomerOrder ( <add property> value : "High" )
  <insert action>
```

Message	Dictionary Object	Property
RUL-05580: syntax error for token "a" at column "3"	OracleRules1/Ruleset_1/Rule_1/CustomerOrder/Test[1]/Expression[2]	Value

This is a new exception since the last update!

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 dropdown 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.

4.4.1 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
`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.

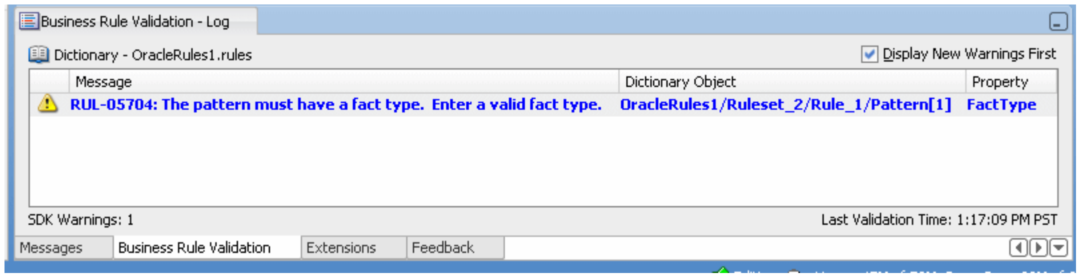
For more information, see:

- Section 4.4.2, "Understanding Rule Validation"
- Section 4.4.3, "Understanding Decision Table Validation"
- Section 4.4.4, "How to Validate a Dictionary"

4.4.2 Understanding Rule Validation

When you click the **Validate** icon Rules Designer displays the validation log. When you first add a rule you see validation warnings similar to those shown in Figure 4–29.

Figure 4–29 Rules Validation Messages



Business Rule Validation - Log

Dictionary - OracleRules1.rules ☑ Display New Warnings First

Message	Dictionary Object	Property
RUL-05704: The pattern must have a fact type. Enter a valid fact type.	OracleRules1/Ruleset_2/Rule_1/Pattern[1]	FactType

SDK Warnings: 1 Last Validation Time: 1:17:09 PM PST

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]
```

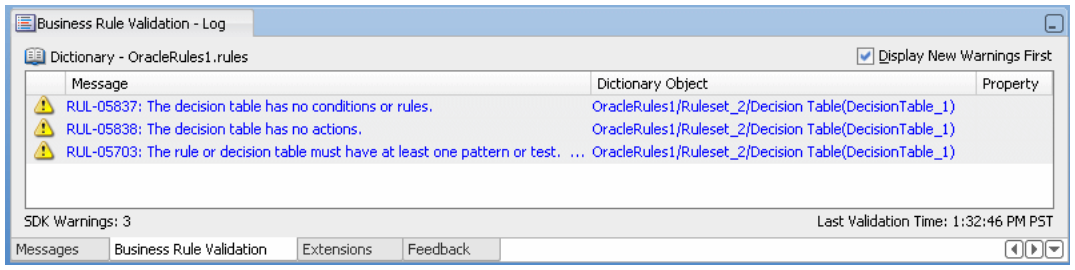
In validation messages, the dictionary object name for a rule uses indexes that start at 1. Thus, the first pattern is `Pattern[1]`.

In addition to validating rules, you can also test them in Rules Designer as you are designing them. For more information, see Section 8.1.1, "How to Test Rules Using a Test Function in Rules Designer".

4.4.3 Understanding Decision Table Validation

When you click the **Validate** icon Rules Designer displays the validation log. When you first add a Decision Table you see validation warnings similar to those shown in Figure 4–30.

Figure 4–30 Decision Table Validation Messages



Business Rule Validation - Log

Dictionary - OracleRules1.rules ☑ Display New Warnings First

Message	Dictionary Object	Property
RUL-05837: The decision table has no conditions or rules.	OracleRules1/Ruleset_2/Decision Table(DecisionTable_1)	
RUL-05838: The decision table has no actions.	OracleRules1/Ruleset_2/Decision Table(DecisionTable_1)	
RUL-05703: The rule or decision table must have at least one pattern or test. …	OracleRules1/Ruleset_2/Decision Table(DecisionTable_1)	

SDK Warnings: 3 Last Validation Time: 1:32:46 PM PST

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:

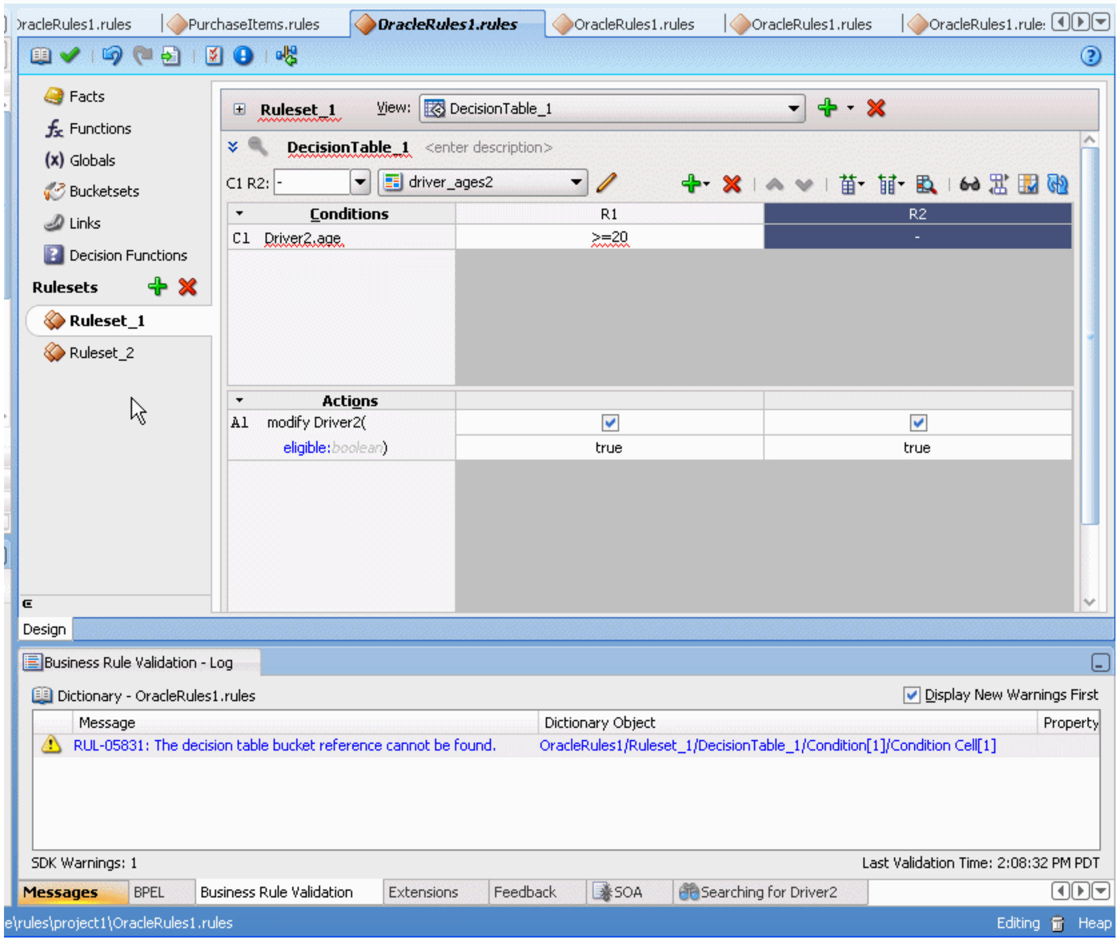
```
OR1/Ruleset_1/DecisionTable_1/Action[1]/Action Cell[1]
```

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 as shown in Figure 4–31.

Figure 4–31 Decision Table with Warning on a Condition Cell



Conditions	R1	R2
C1 Driver2.age	>=20	-

Actions		
A1 modify Driver2(	✓	✓
eligible:boolean)	true	true

Business Rule Validation - Log

Dictionary - OracleRules1.rules

Display New Warnings First

Message	Dictionary Object	Property
RUL-05831: The decision table bucket reference cannot be found.	OracleRules1/Ruleset_1/DecisionTable_1/Condition[1]/Condition Cell[1]	

SDK Warnings: 1

Last Validation Time: 2:08:32 PM PDT

4.4.4 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** icon (a checkmark).
2. Select the Business Rule Validation 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 dropdown 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.

4.5 Using Advanced Settings with Rules and Decision Tables

Advanced settings for rules and Decision Tables let you work with features that provide advanced options that not all Oracle Business Rules users need. These features include:

- **Advanced Mode**: allows additional pattern matching options and nested tests in rules.

  For more information, see:

  - Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table"
  - Section 4.5.2, "How to Select the Advanced Mode Option"
  - Section 4.7.5, "What You Need to Know About Advanced Mode Rules"

- **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 Section 4.8.2, "How to Create Simple Tree Mode Rules".

- **Auto Conflict Resolution**: (available only with Decision Table advanced settings). Specifies that Decision Table conflicts are automatically resolved using an Override conflict resolution, when this is possible using the automatic conflict resolution policies.

  For more information, see Section 5.3.1.4, "Understanding Decision Table Conflict Analysis".

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

  For more information, see Section 4.5.3, "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 Section 4.5.4, "How to Select the Logical Option".

- **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 Section 4.5.5, "How to Set a Priority for a Rule".

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

  For more information, see, Section 4.5.6, "How to Specify Effective Dates".

- **Allow Gaps** (available only with Decision Table advanced settings). This checkbox 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 Section 5.3.1.3, "Understanding Decision Table Gap Analysis" and Section 5.3.5, "How to Perform Decision Table Gap Analysis".

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

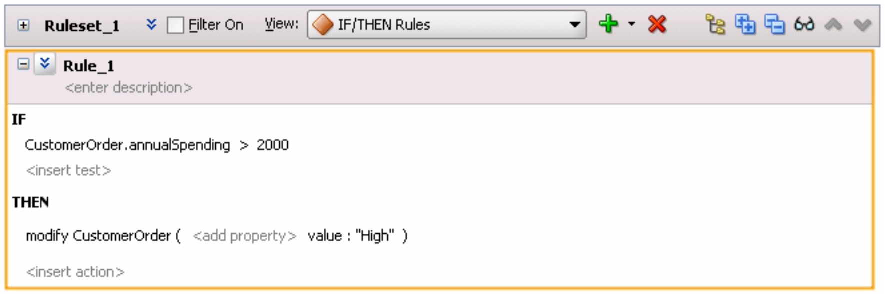
In Rules Designer, next to each rule name and Decision Table name, the show or hide advanced settings icon lets you show and hide advanced settings.

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 dropdown list, select either **IF/THEN Rules** or select a Decision Table.

a. To show the advanced settings, next to the rule name click **Show Advanced Settings**, as shown in Figure 4–32 (there is a highlighted icon shown next to the rule name, **Rule_1**).

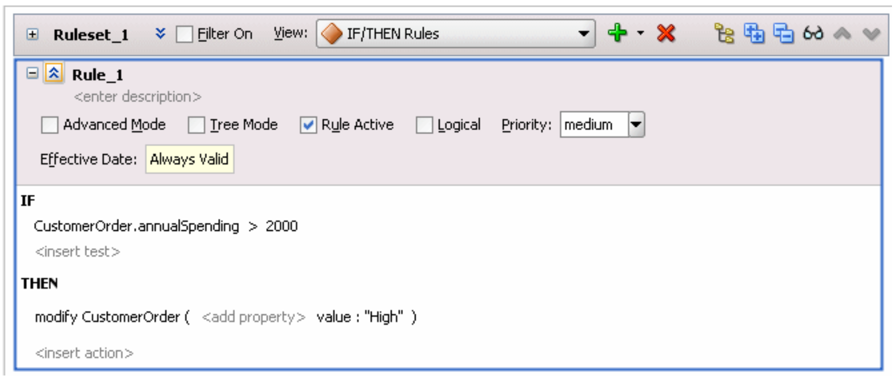
Figure 4–32 Showing Rules Advanced Settings



IF
CustomerOrder.annualSpending > 2000
<insert test>
THEN
modify CustomerOrder (<add property> value : "High")
<insert action>

b. To hide the advanced settings, next to the rule name click **Hide Advanced Settings**, as shown in Figure 4–33 (there is a highlighted icon shown next to the rule name, **Rule_1**).

Figure 4–33 Hiding Advanced Settings in a Rule



☐ Advanced Mode ☐ Tree Mode ☑ Rule Active ☐ Logical Priority: medium
Effective Date: Always Valid

IF
CustomerOrder.annualSpending > 2000
<insert test>
THEN
modify CustomerOrder (<add property> value : "High")
<insert action>

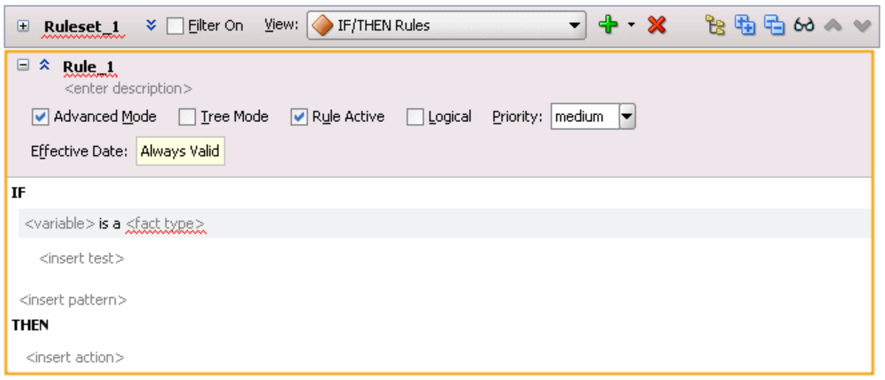
4.5.2 How to Select the Advanced Mode Option

Select Advanced Mode to use Rule or Decision Table features that provide additional pattern matching options and additional actions. For more information, see Section 4.7, "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** icon next to the rule or Decision Table name (see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table").
3. Select **Advanced Mode**, as shown in Figure 4–34.

Figure 4–34 Setting Advanced Mode Option



4.5.3 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 unselected, 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** icon next to the rule or Decision Table name (see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table").
3. Select **Rule Active**.

4.5.4 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 on the logical property, see *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*.

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** icon next to the rule or Decision Table name (see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table").
3. Select **Logical**.

4.5.5 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).

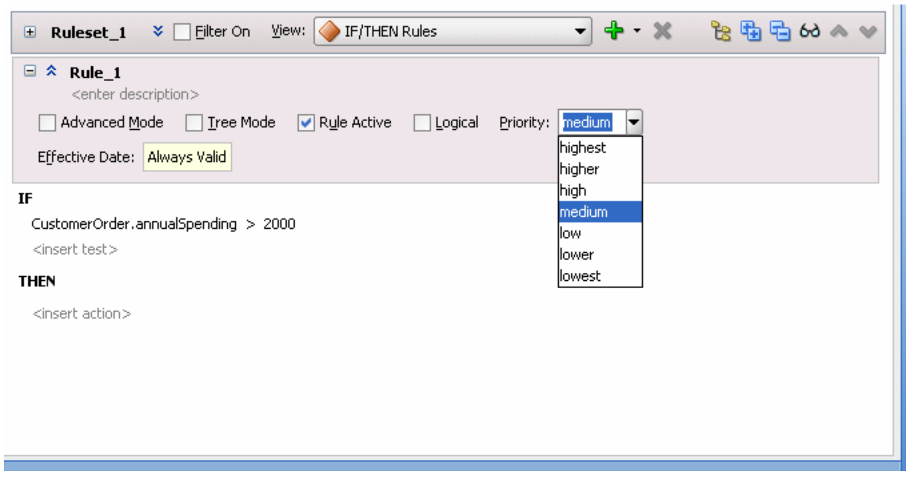
Table 4–4 Priority String Value Mapping

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:

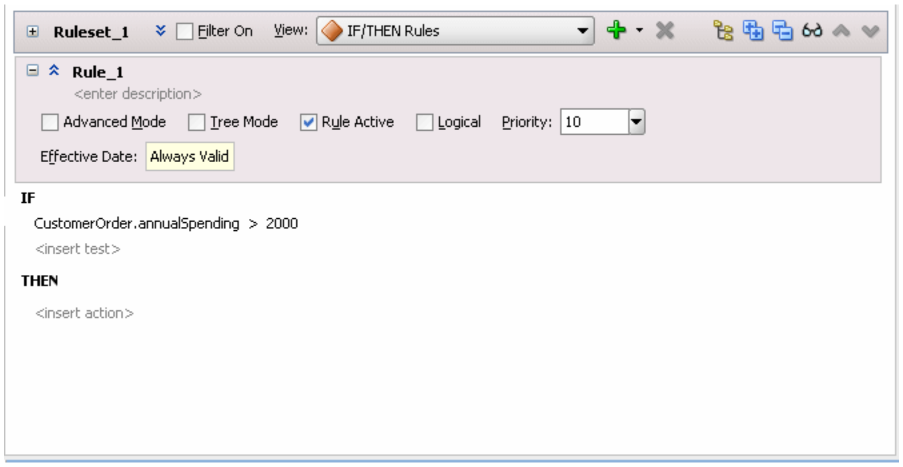
1. Select the rule or Decision Table where you want to set the priority.
2. Click the **Show Advanced Settings** icon next to the rule or Decision Table name (see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table").
3. In the **Priority** field, specify the priority value:
   a. To specify a named priority, select a named priority from the **Priority** dropdown list as Figure 4–35 shows.

Figure 4–35 Choosing a Predefined Named Priority



   b. To specify an integer priority, click in the Priority field and enter a positive or negative integer value and press **Enter**, as Figure 4–36 shows.

Figure 4–36 Choosing a User Defined Numeric Priority



Advanced Mode [ ] · Tree Mode [ ] · Rule Active [✓] · Logical [ ] · Priority: 10

Effective Date: Always Valid

```
IF
  CustomerOrder.annualSpending > 2000
  <insert test>
THEN
  <insert action>
```

4.5.6 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** icon next to the rule or Decision Table name (see Section 4.5.1, "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 Section 4.9, "Using Date Facts, Date Functions, and Specifying Effective Dates" and Section 4.2.2, "How to Set the Effective Date for a Ruleset".

4.6 Working with Nested Tests

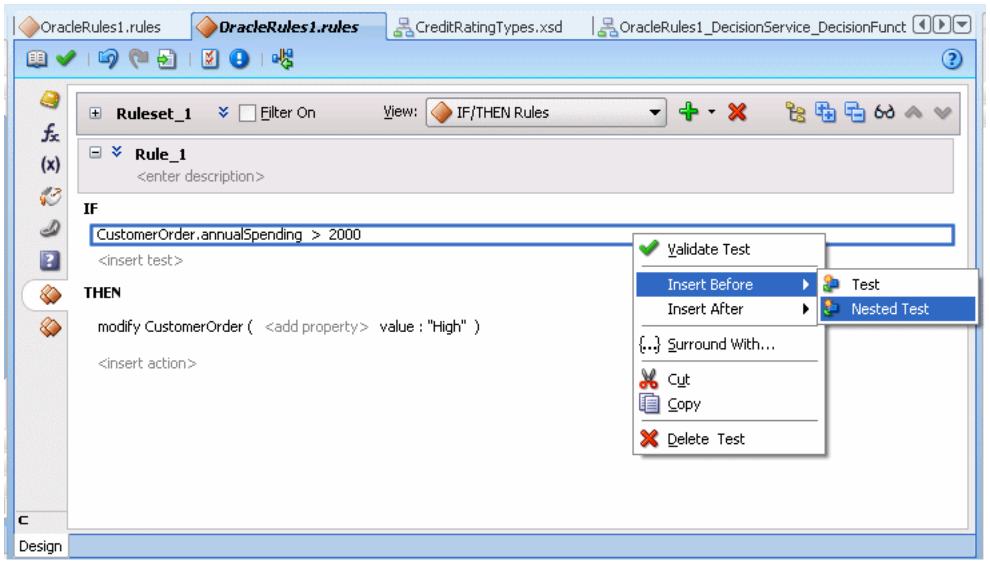
In a rule or a Decision Table you can create more complicated tests using the nested tests feature.

4.6.1 How to Use Nested Tests

To use nested tests:

1. Select the rule where you want to use a nested test.
2. In the **IF** area, select a test. This surrounds the test with a highlighted box.
3. With a test selected right-click to display the dropdown list, as shown in Figure 4–37.

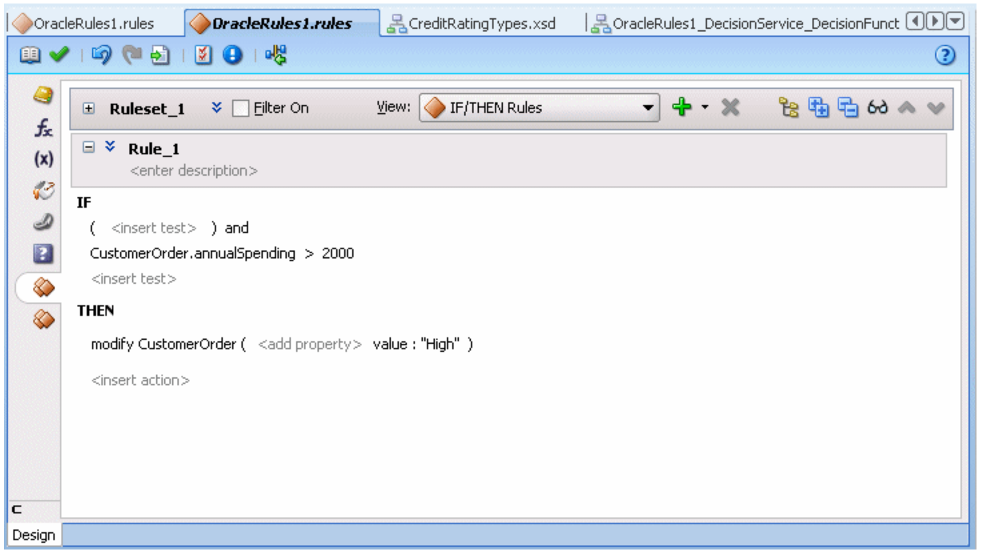
Figure 4–37 Adding a Nested Test to a Rule



```
Rule_1
<enter description>
IF
CustomerOrder.annualSpending  >  2000
<insert test>
THEN
modify CustomerOrder (  <add property>  value : "High"  )
<insert action>
```

4. To add the nested test, from the dropdown list select either **Insert Before** or **Insert After** and then select **Nested Test**. A nested test is shown in Figure 4–38.

Figure 4–38 A Nested Test Added to a Rule



```
Rule_1
<enter description>
IF
(  <insert test>  ) and
CustomerOrder.annualSpending  >  2000
<insert test>
THEN
modify CustomerOrder (  <add property>  value : "High"  )
<insert action>
```

4.7 Working with Advanced Mode Rules

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 Section 4.7.1, "How to Use Advanced Mode Pattern Matching Options")
- Additional Matched Fact Naming options (see Section 4.7.2, "How to Use Advanced Mode Matched Fact Naming")
- Additional Supported Action forms (see Section 4.7.3, "How to Use Advanced Mode Action Forms")

- Pattern Match Aggregate Function options (see Section 4.7.4, "How to Use Advanced Mode Aggregate Conditions")

For more information, see Section 4.7.5, "What You Need to Know About Advanced Mode Rules".

4.7.1 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.

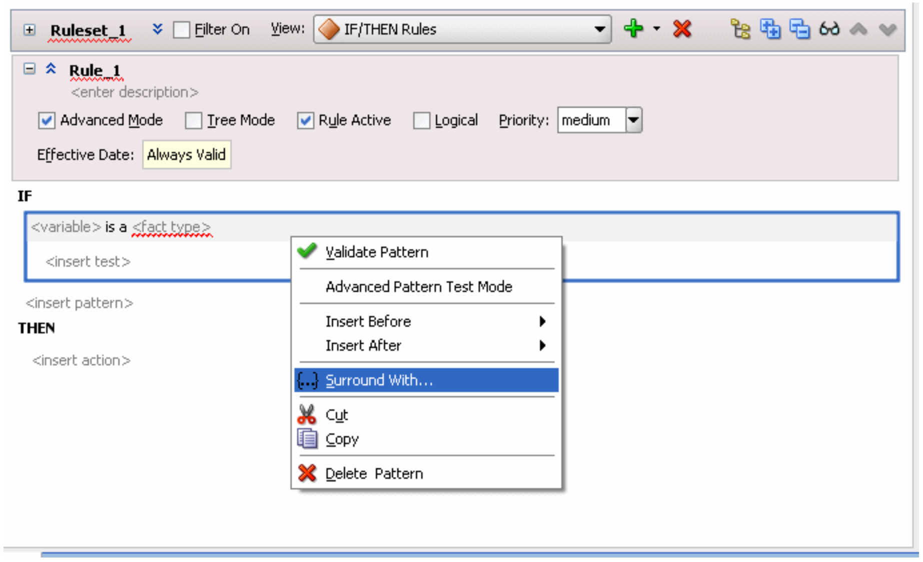
Table 4–5 Advanced Mode Pattern Matching Options

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 Section 4.7.4, "How to Use Advanced Mode Aggregate Conditions".

To use advanced mode pattern matching options:

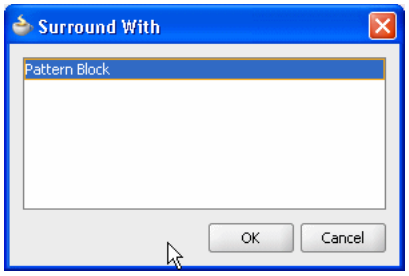
1. Select the rule or Decision Table where you want to use pattern matching options.
2. Click the **Show Advanced Settings** icon next to the rule or Decision Table name (see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table").
3. Select **Advanced Mode**.
4. Right-click a test pattern and select **Surround With...** as shown in Figure 4–39.

Figure 4–39 Surrounding With Option



The Surround With dialog appears as shown in Figure 4–40.

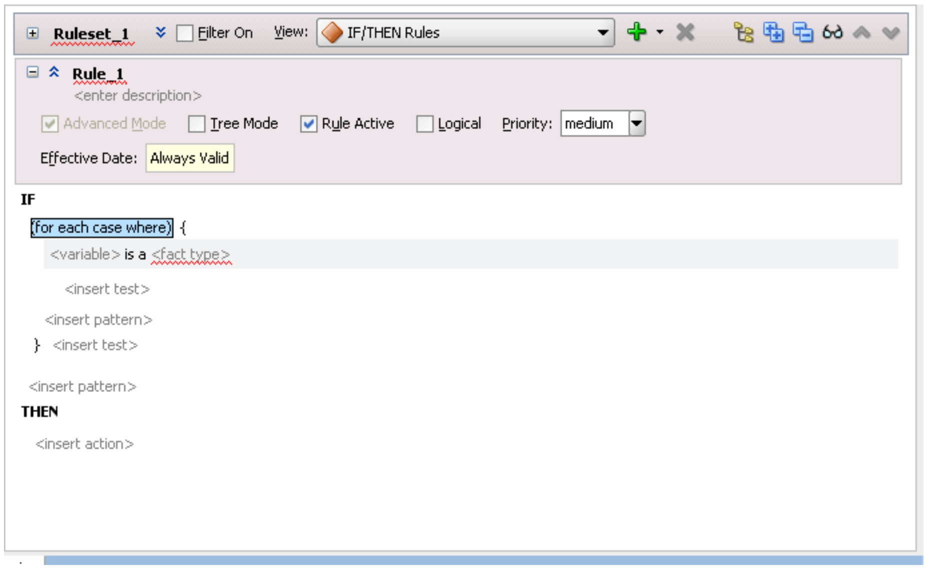
Figure 4–40 Surround With Dialog



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

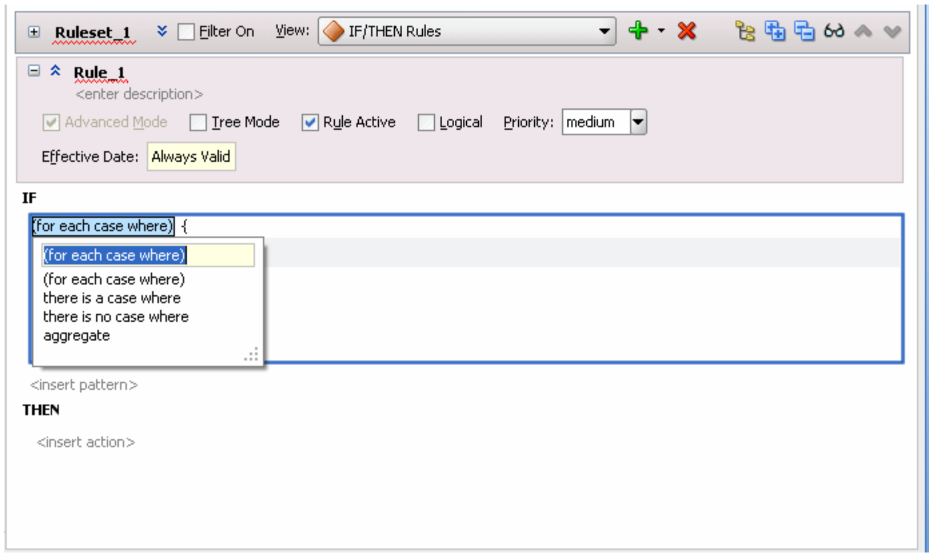
   The pattern is surrounded by a nested pattern with the default **(for each case where)** as shown in Figure 4–41.

Figure 4–41 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 4–42.

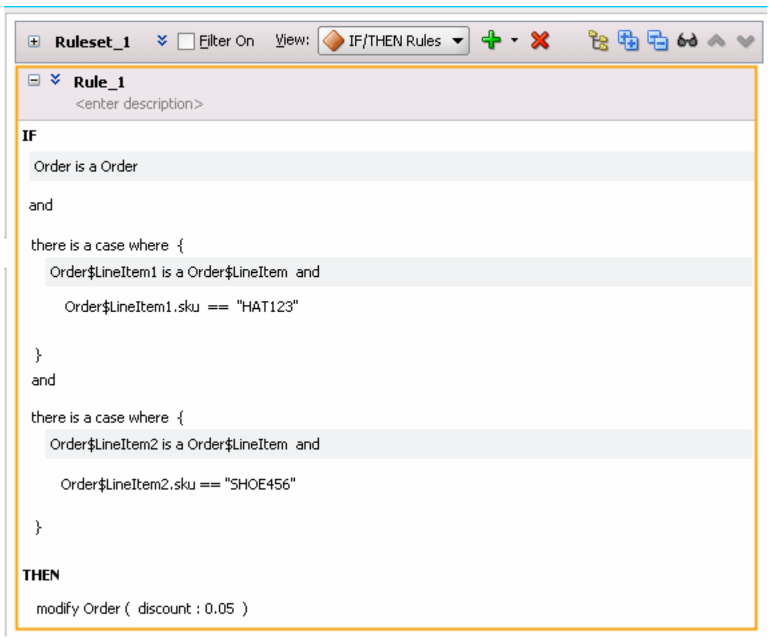
Figure 4–42 Adding an Advanced Pattern Match Option



4.7.2 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 4–43 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.

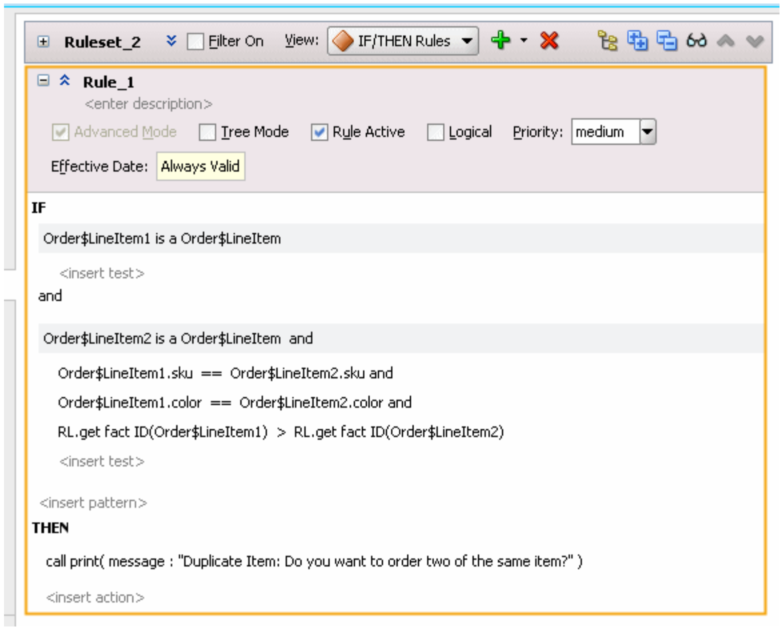
Figure 4–43 Rule Using a Pattern Binding Variable



```
IF
 Order is a Order

 and

 there is a case where {
    Order$LineItem1 is a Order$LineItem and

      Order$LineItem1.sku == "HAT123"

 }
 and

 there is a case where {
    Order$LineItem2 is a Order$LineItem and

      Order$LineItem2.sku == "SHOE456"

 }

THEN
  modify Order ( discount : 0.05 )
```

For example, you can create the rule, as shown in Figure 4–44 to find duplicate items in a customer order. This example shows the use of matched in a rule.

Figure 4–44 Rule to Find Duplicate Items in an Order



IF

Order$LineItem1 is a Order$LineItem

<insert test>

and

Order$LineItem2 is a Order$LineItem and

Order$LineItem1.sku == Order$LineItem2.sku and

Order$LineItem1.color == Order$LineItem2.color and

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

<insert test>

<insert pattern>

THEN

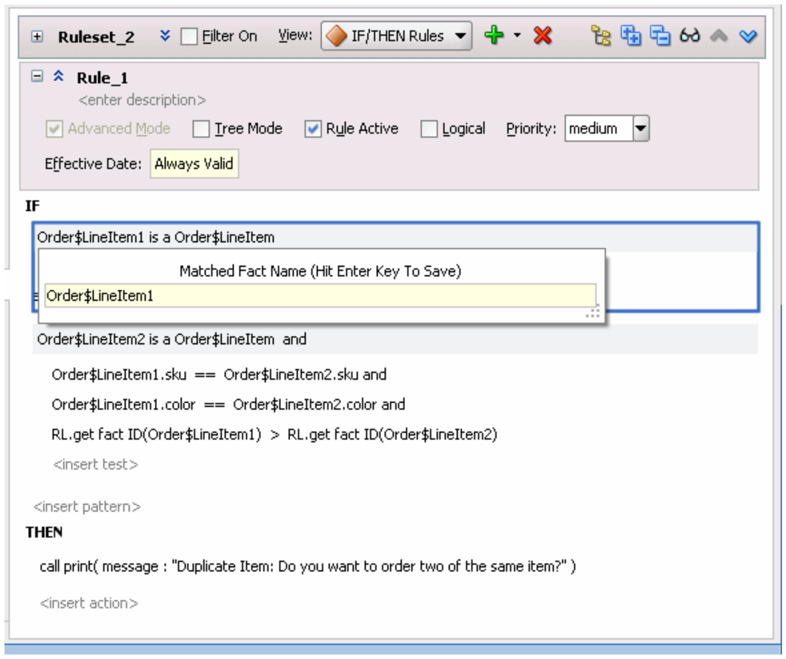
call print(message : "Duplicate Item: Do you want to order two of the same item?")

<insert action>

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** icon next to the rule name (see Section 4.5.1, "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 dropdown list.
5. Select the supplied matched fact name and modify it as needed, as shown in Figure 4–45. For example, enter the matched fact name `Order$LineItem1` and then press **Enter**.

Figure 4–45 Adding a Matched Fact Variable Name

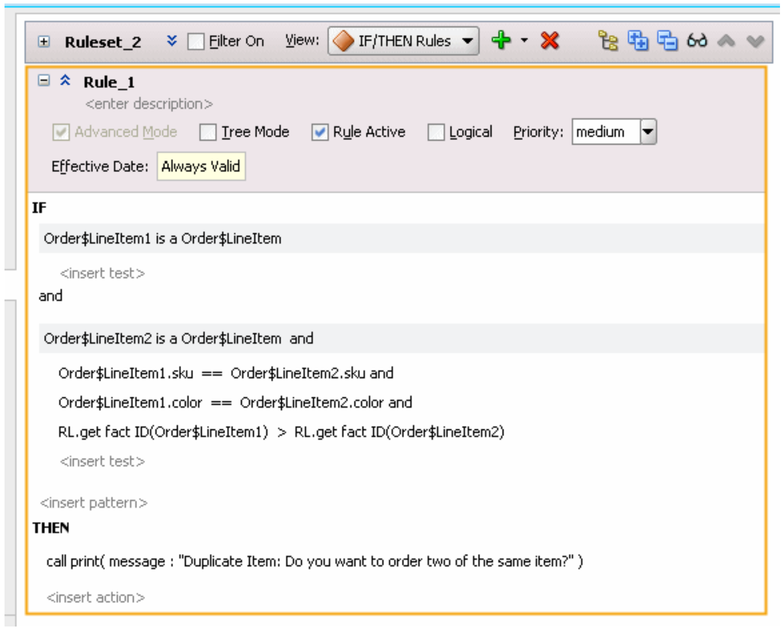

Ruleset_2 Filter On View: IF/THEN Rules

Rule_1
<enter description>
[✓ Advanced Mode] Tree Mode [✓ Rule Active] Logical Priority: medium
Effective Date: Always Valid

```
IF
Order$LineItem1 is a Order$LineItem
    Matched Fact Name (Hit Enter Key To Save)
    Order$LineItem1
Order$LineItem2 is a Order$LineItem  and
  Order$LineItem1.sku == Order$LineItem2.sku and
  Order$LineItem1.color == Order$LineItem2.color and
  RL.get fact ID(Order$LineItem1) > RL.get fact ID(Order$LineItem2)
  <insert test>
<insert pattern>
THEN
 call print( message : "Duplicate Item: Do you want to order two of the same item?" )
 <insert action>
```

6. Create the rule as Figure 4–46 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 4–46 Choosing a Matched Fact Variable Name as an Operand


Ruleset_2 Filter On View: IF/THEN Rules

Rule_1
<enter description>
[✓ Advanced Mode] Tree Mode [✓ Rule Active] Logical Priority: medium
Effective Date: Always Valid

```
IF
Order$LineItem1 is a Order$LineItem
  <insert test>
and
Order$LineItem2 is a Order$LineItem  and
  Order$LineItem1.sku == Order$LineItem2.sku and
  Order$LineItem1.color == Order$LineItem2.color and
  RL.get fact ID(Order$LineItem1) > RL.get fact ID(Order$LineItem2)
  <insert test>
<insert pattern>
THEN
 call print( message : "Duplicate Item: Do you want to order two of the same item?" )
 <insert action>
```

Note in Figure 4–46 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 Appendix C.4, "How Do I Correctly Express a Self-Join?".

4.7.3 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.

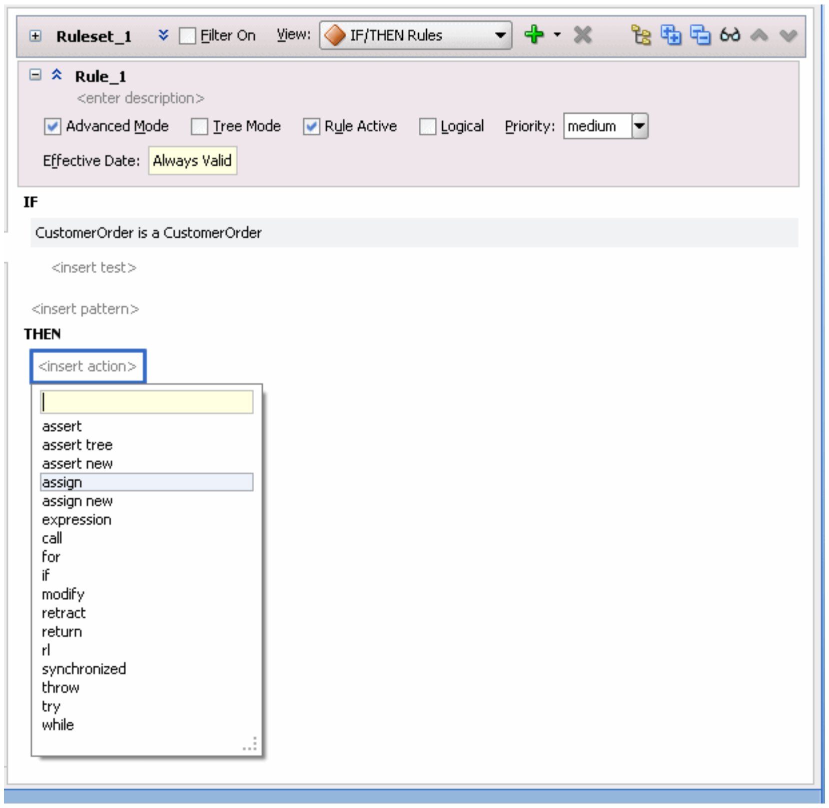
Table 4–6 Advanced Mode Action Options

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. A for loop can be used 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.

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** icon next to the rule or Decision Table name (see Section 4.5.1, "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 4–47.

Figure 4–47 Adding an Action to a Rule in Advanced Mode



6. In the dropdown 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.

4.7.4 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.

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
```

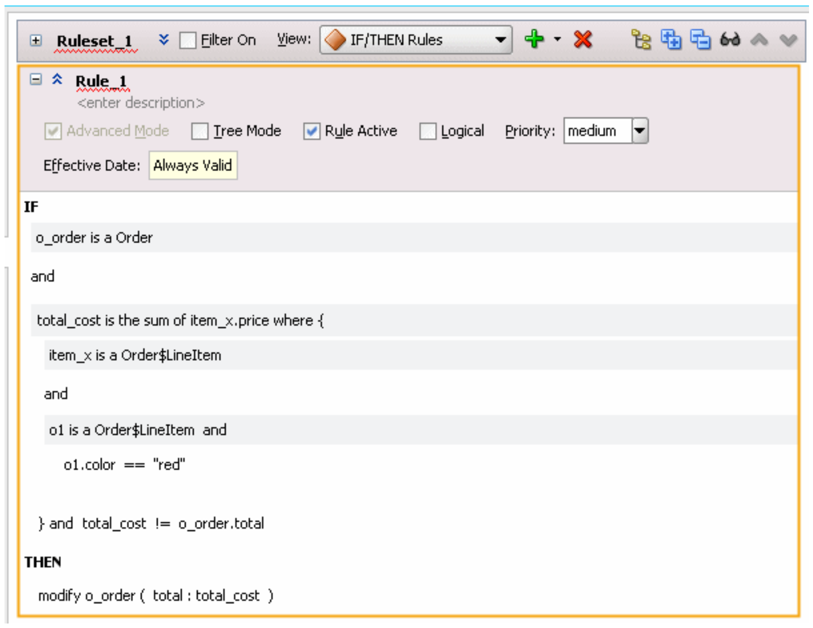
The five line items may span multiple facts. Thus, you can use the `count` aggregate function to write this sample special order rule.

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 4–48.

Figure 4–48 Using Aggregate Functions with Rules Red Color Total Cost Rule

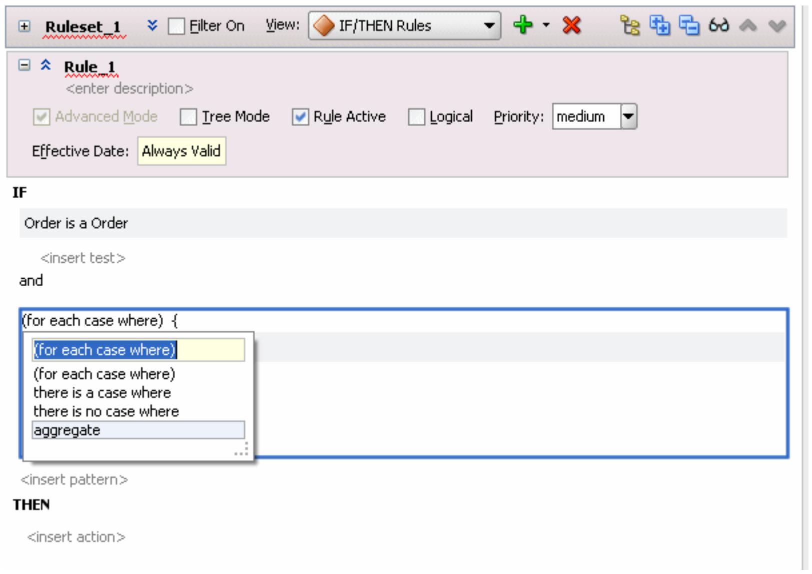


```
IF
 o_order is a Order
 and
 total_cost is the sum of item_x.price where {
   item_x is a Order$LineItem
   and
   o1 is a Order$LineItem and
     o1.color == "red"

 } and total_cost != o_order.total
THEN
 modify o_order ( total : total_cost )
```

To use advanced mode aggregates:

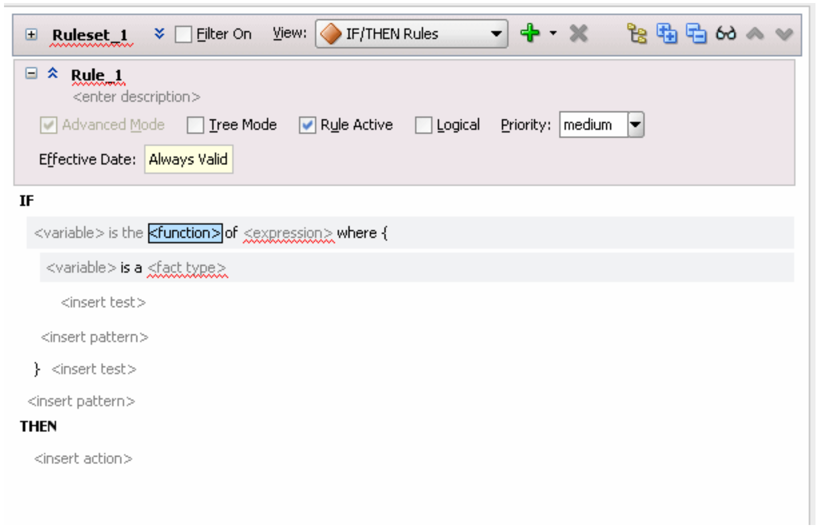
1. Select or create the rule or Decision Table where you want to use an aggregate function.
2. Click the **Show Advanced Settings** icon next to the rule or Decision Table name (see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table").
3. Select **Advanced Mode**.
4. Enter the fact type you want to work with.
5. Select **<insert pattern>** to add a pattern.
6. Select the new pattern.
7. Right-click the pattern and select **Surround With...**. This displays the Surround With dialog.
8. In the Surround With dialog select **Pattern Block**. For more information, see Section 4.7.1, "How to Use Advanced Mode Pattern Matching Options".
9. Click **OK**.
10. In the pattern select the first field. By default this field contains **(for each case where)**, as shown in Figure 4–49.

Figure 4–49 Adding an Advanced Pattern Match Option



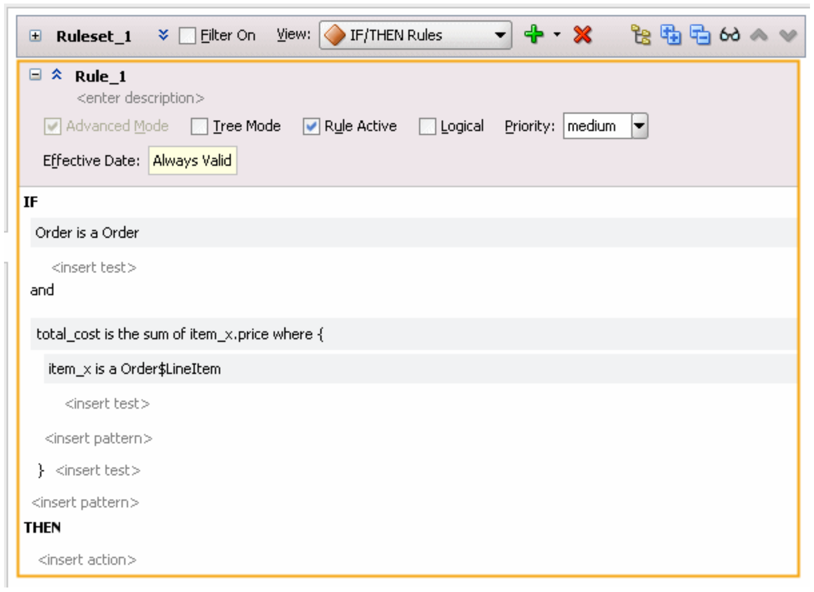
11. Select the **aggregate** option. This adds the context sensitive fields for an aggregate, as shown in Figure 4–50.

Figure 4–50 Using Aggregate Functions in a Rule



12. Click **<function>** and select a function from the dropdown list.
13. In the condition, click **<fact type>** and select a fact type from the dropdown list.
14. Click **<expression>** and select an expression from the dropdown list.
15. 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 4–51 shows a completed rule using aggregate. In this example, for clarity the rule shows the variable names `total_cost` and `item_x`.

Figure 4–51 Completed Aggregate Function in a Rule



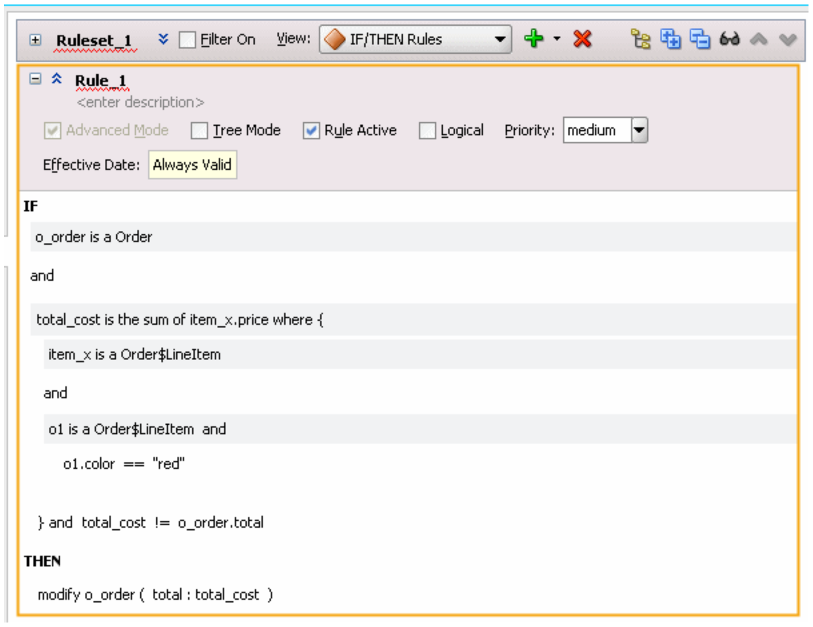
Ruleset_1 Filter On View: IF/THEN Rules

Rule_1
<enter description>
[✓ Advanced Mode] Tree Mode [✓ Rule Active] Logical Priority: medium
Effective Date: Always Valid

```
IF
  Order is a Order
    <insert test>
  and
  total_cost is the sum of item_x.price where {
    item_x is a Order$LineItem
      <insert test>
    <insert pattern>
  } <insert test>
  <insert pattern>
THEN
  <insert action>
```

16. Enter additional tests as required. For this example you enter the test for items with color "red", as Figure 4–52 shows.

Figure 4–52 Using Aggregate Functions with Rules Red Color Total Cost Rule



Ruleset_1 Filter On View: IF/THEN Rules

Rule_1
<enter description>
[✓ Advanced Mode] Tree Mode [✓ Rule Active] Logical Priority: medium
Effective Date: Always Valid

```
IF
  o_order is a Order
  and
  total_cost is the sum of item_x.price where {
    item_x is a Order$LineItem
    and
    o1 is a Order$LineItem and
      o1.color == "red"
  } and total_cost != o_order.total
THEN
  modify o_order ( total : total_cost )
```

4.7.5 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 deselect **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).

To deselect the advanced mode option:

1. Select the rule or Decision Table where you want to deselect **Advanced Mode**.
2. Click the **Show Advanced Settings** icon next to the rule or Decision Table name (see Section 4.5.1, "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** icon changes from gray to enabled). Then simplify the rule and when **Advanced Mode** is enabled, deselect **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.

4.8 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.

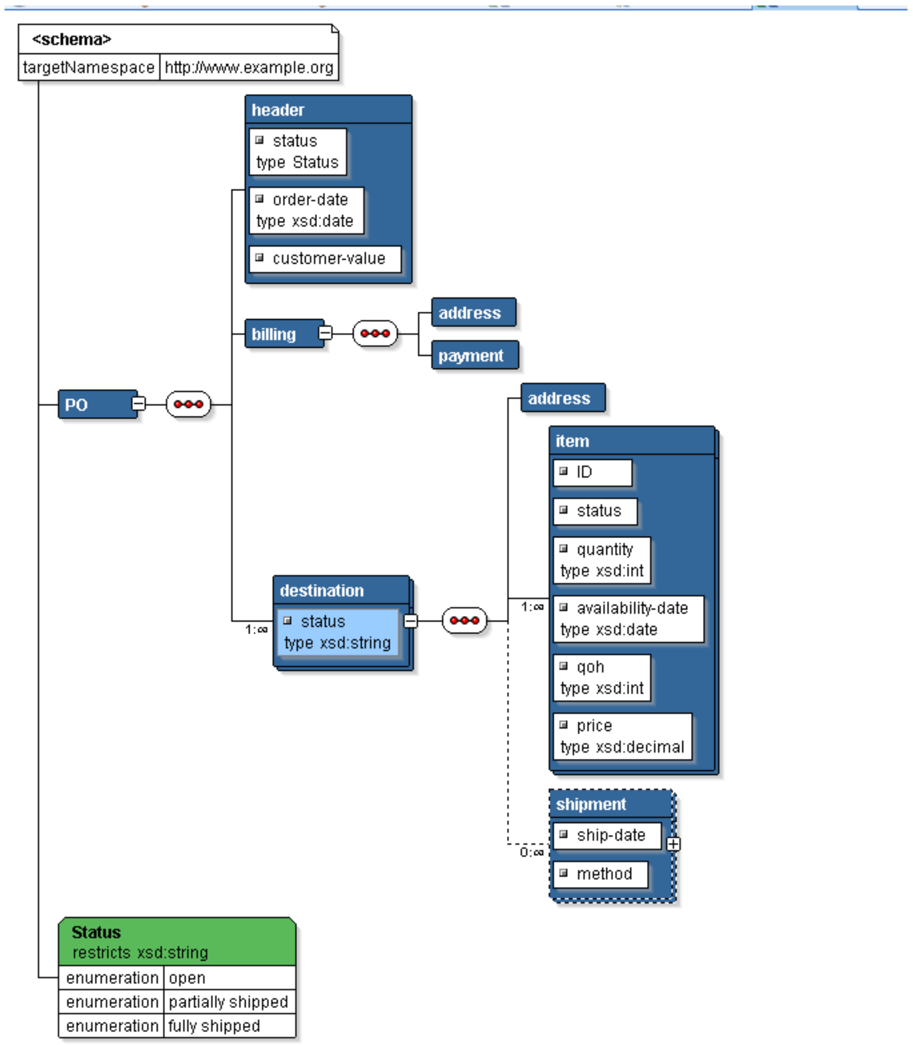
4.8.1 Introduction to Tree Mode Rules

To introduce tree mode rules, it is instructive to work with an example. 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 4–53 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 4–53 PO Schema for Tree Mode Rules Sample



Example 4–1 shows the sample purchase order XML schema as represented in Figure 4–53.

Example 4–1 Sample Purchase Order (PO) Schema

```
<?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>
```

Example 4–2 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
```

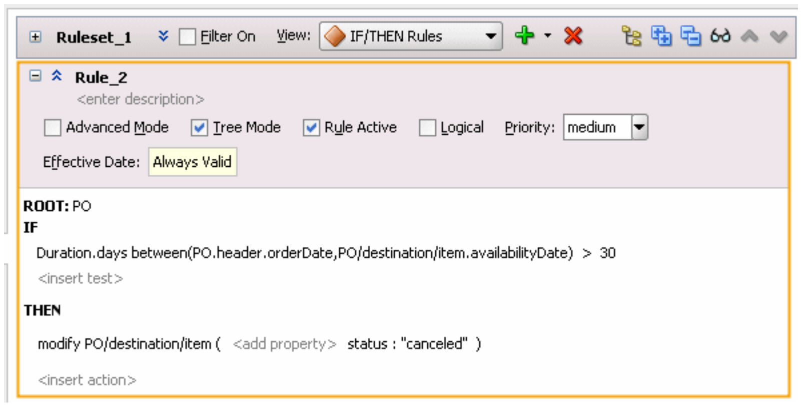
Example 4–2 Sample Abbreviated PO XML Instance

```
<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>
```

4.8.1.1 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, as shown in Figure 4–54.

Figure 4–54 Simple Tree Mode Rule with Tree Mode Selected



ROOT: PO
IF
Duration.days between(PO.header.orderDate,PO/destination/item.availabilityDate) > 30
<insert test>
THEN
modify PO/destination/item (<add property> status : "canceled")
<insert action>

When you create rules with **Tree Mode** selected and **Advanced Mode** unselected 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.

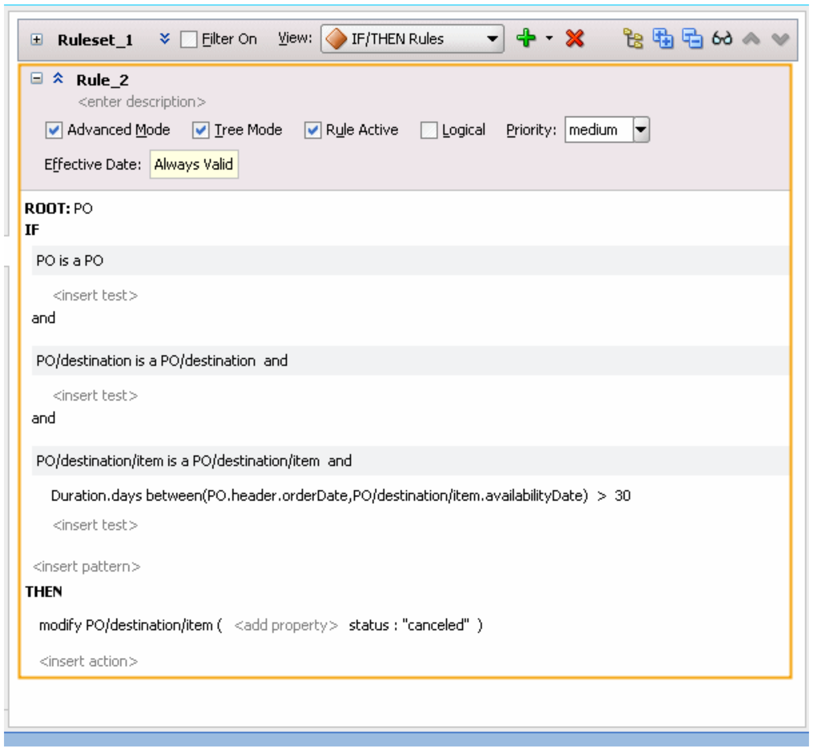
With Simple Tree Mode you can only choose terms that do not require many-to-many joins or aggregation.

For more information, see Section 4.8.2, "How to Create Simple Tree Mode Rules".

4.8.1.2 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 4–55. Rules Designer shows patterns for the tree structured facts but the simple tests that join the parent and child facts are hidden.

Figure 4–55 Advanced Tree Mode



Rule_2
<enter description>
Advanced Mode ✓ Tree Mode ✓ Rule Active ✓ Logical ☐ Priority: medium
Effective Date: Always Valid

```
ROOT: PO
IF
 PO is a PO
   <insert test>
and
 PO/destination is a PO/destination  and
   <insert test>
and
 PO/destination/item is a PO/destination/item  and
   Duration.days between(PO.header.orderDate,PO/destination/item.availabilityDate) > 30
   <insert test>
 <insert pattern>
THEN
  modify PO/destination/item (  <add property>  status : "canceled"  )
  <insert action>
```

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**. 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 Section 4.8.3, "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.

4.8.2 How to Create Simple Tree Mode Rules

Given the XML schema shown in Example 4–1 and the schema instance shown in Example 4–2, 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.

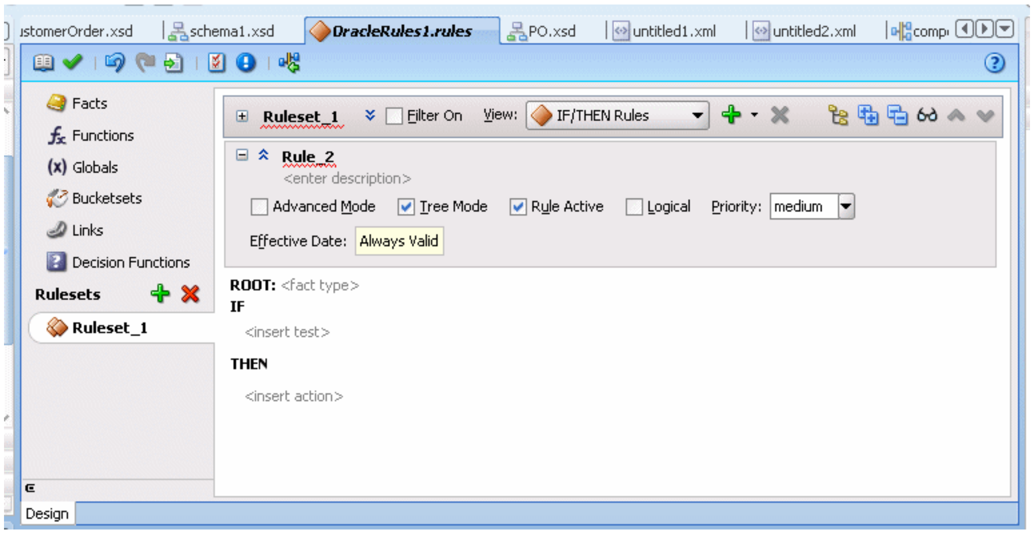
   For more information, see Section 4.3.1, "How to Add Rules".

2. View advanced settings.

   For more information, see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table".

3. Select **Tree Mode** as Figure 4–56 shows.

Figure 4–56 Simple Tree Mode Advanced Settings



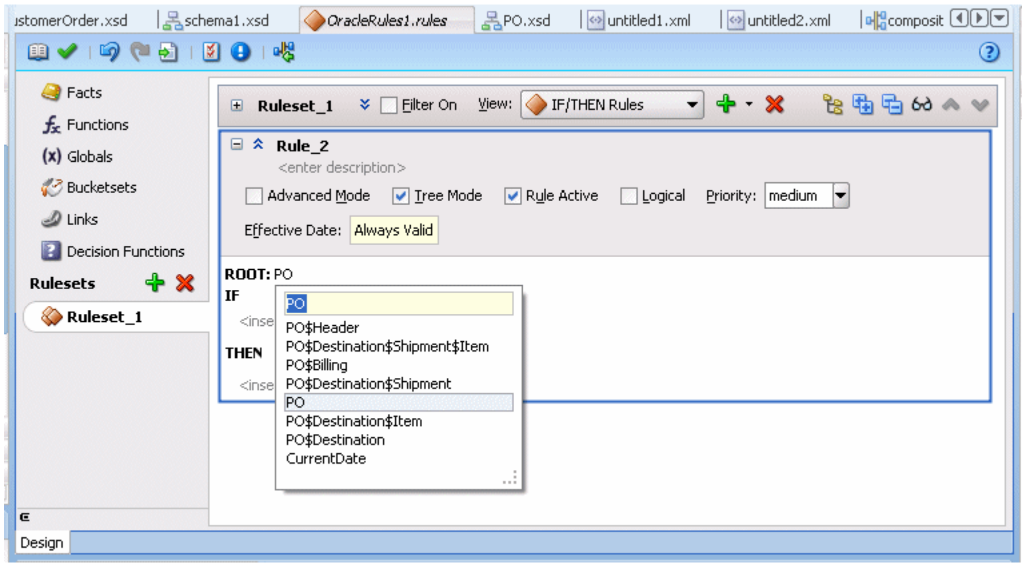
Ruleset_1 ☐ Filter On View: IF/THEN Rules

Rule_2
<enter description>
☐ Advanced Mode ☑ Tree Mode ☑ Rule Active ☐ Logical Priority: medium
Effective Date: Always Valid

ROOT: <fact type>
IF
<insert test>
THEN
<insert action>

4. Next to **ROOT:**, click the **<fact type>** place holder and select **PO** from the dropdown list as Figure 4–57 shows.

Figure 4–57 Simple Tree Mode: Configuring the Root



Ruleset_1 Filter On View: IF/THEN Rules

Rule_2
<enter description>
Advanced Mode [✓ Tree Mode] [✓ Rule Active] Logical Priority: medium
Effective Date: Always Valid

ROOT: PO
IF
<inse
THEN
<inse

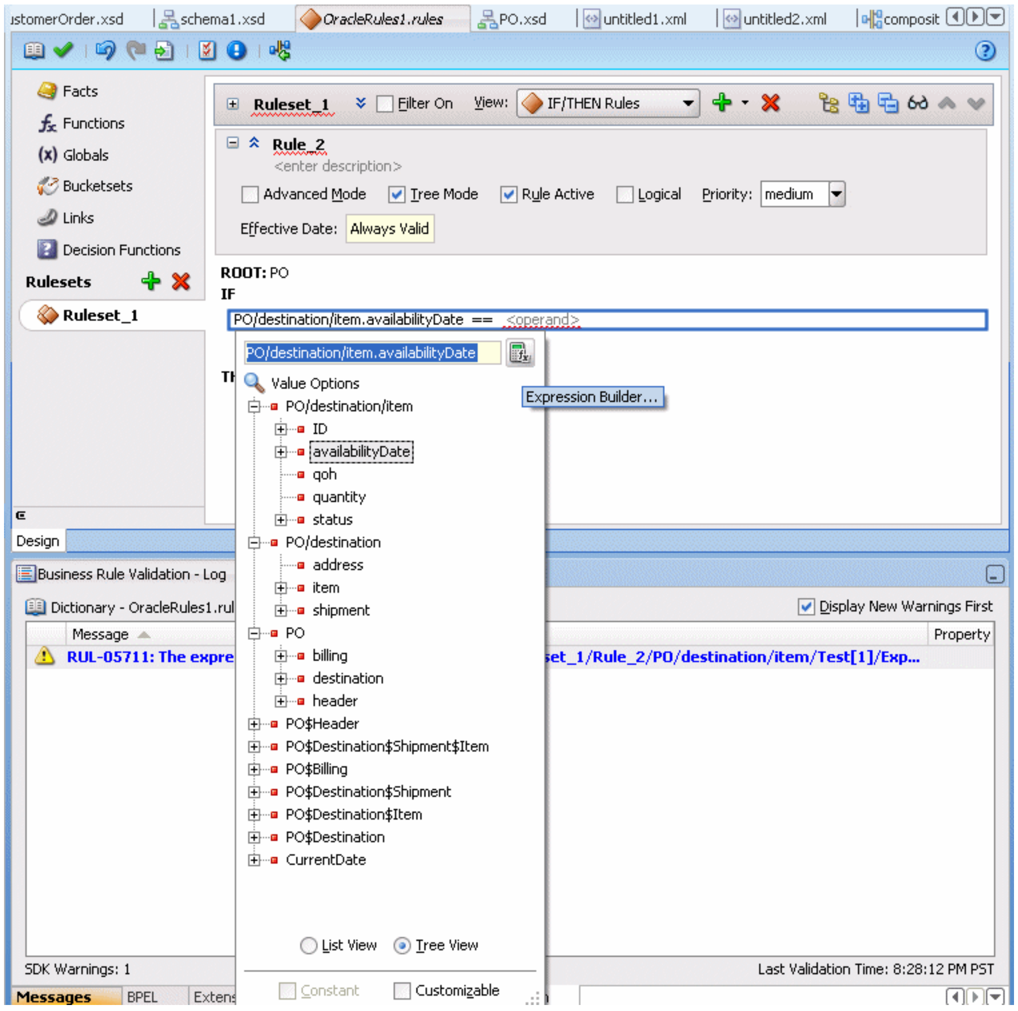
PO
PO$Header
PO$Destination$Shipment$Item
PO$Billing
PO$Destination$Shipment
PO
PO$Destination$Item
PO$Destination
CurrentDate

5. Select **<insert test>**.

   The IF statement now reads `IF <operand> == <operand>`.

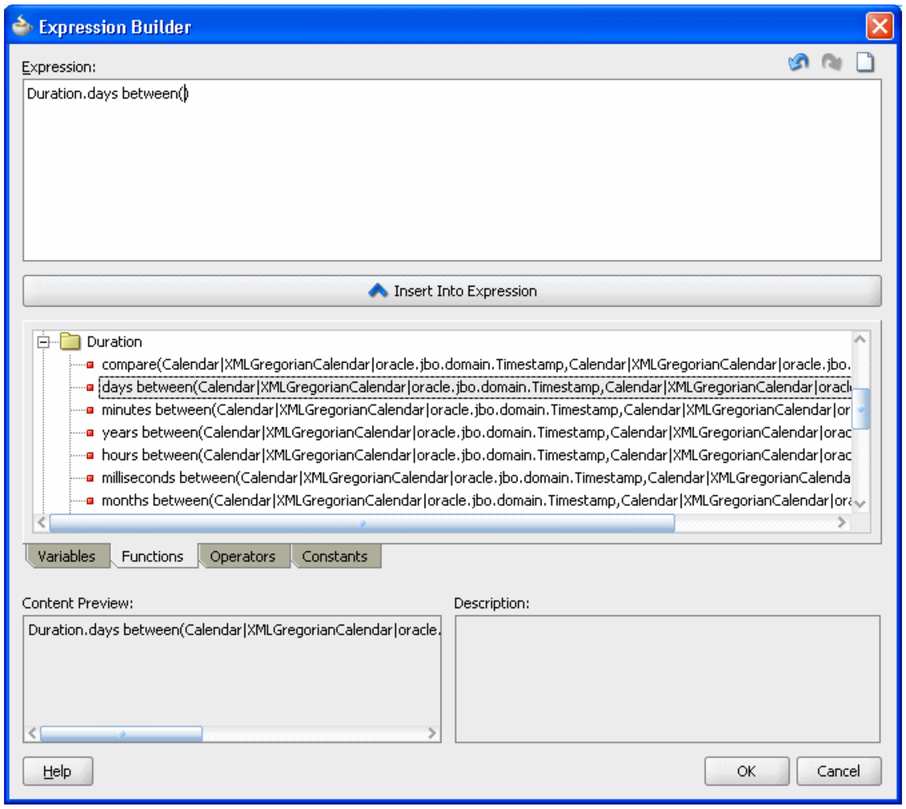
6. Select the left-hand **<operand>**.
7. In the dropdown list, select `PO/destination/item.availabilityDate`.
8. Select **Expression Builder** icon, as shown in Figure 4–58.

Figure 4–58 Adding Simple Tree Mode Expression



9. In the Expression Builder dialog, copy and delete the item shown in the **Expression** area.
10. In the Expression Builder, select the **Functions** tab.
11. In the navigator, expand **Duration** and double-click the **daysbetween** function.
12. Remove the **daysbetween** argument templates, as shown in Figure 4–59.

Figure 4–59 Using Expression Builder to Add a Simple Tree Mode Rule



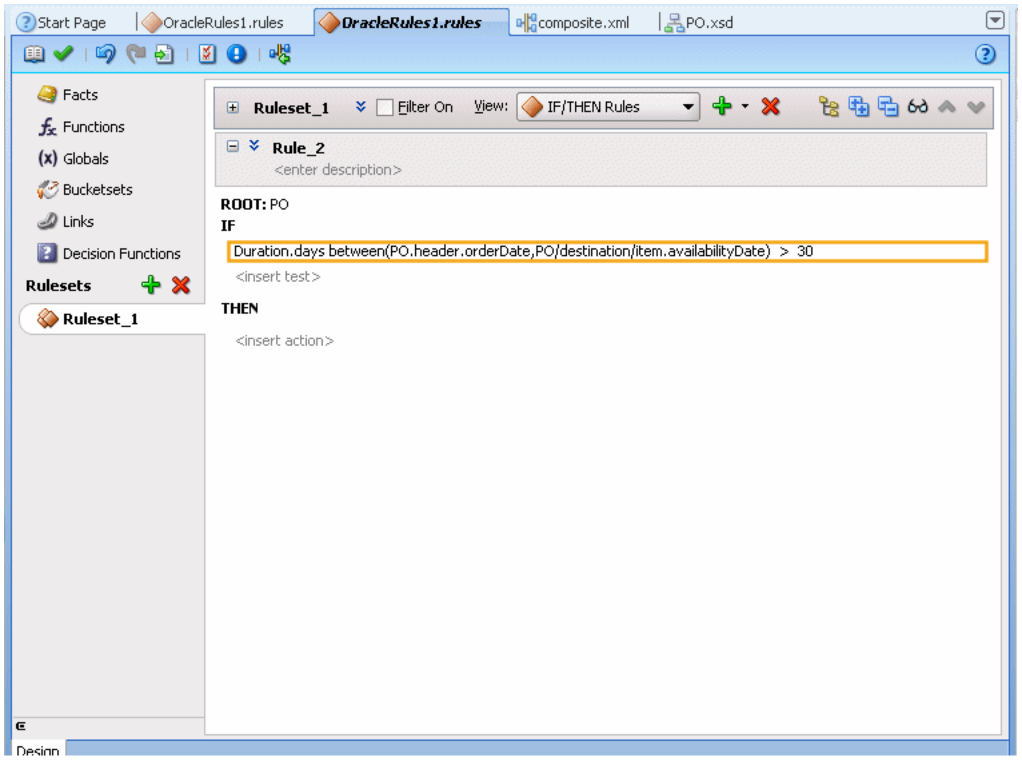
Duration.days between()

Duration
compare(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracle.jbo.
days between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracl
minutes between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|or
years between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|orac
hours between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|orac
milliseconds between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalenda
months between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|ora

Duration.days between(Calendar|XMLGregorianCalendar|oracle.

13. In the **daysbetween** function, paste the value you previously cut as the second argument.
14. In the Expression Builder dialog, select the **Variables** tab.
15. For the **daysbetween** function first argument, use the navigator to expand **PO** and expand **header**, and double-click **orderDate**.
16. In the Expression Builder dialog, click **OK**.
17. In the dropdown list, in the expression area and press **Enter**.
18. Select the operator and enter **>**.
19. Select the right-hand **<operand>** and enter the value 30 and press **Enter**, as shown in Figure 4–60.

Figure 4–60 Simple Tree Mode: Right-Hand Operand with Value 30



```
Rule_2
<enter description>
ROOT: PO
IF
Duration.days between(PO.header.orderDate,PO/destination/item.availabilityDate)  >  30
<insert test>
THEN
<insert action>
```

20. Click **<insert action>** and from the dropdown list select **modify**.

    The THEN statement now reads: `THEN modify <target>`.

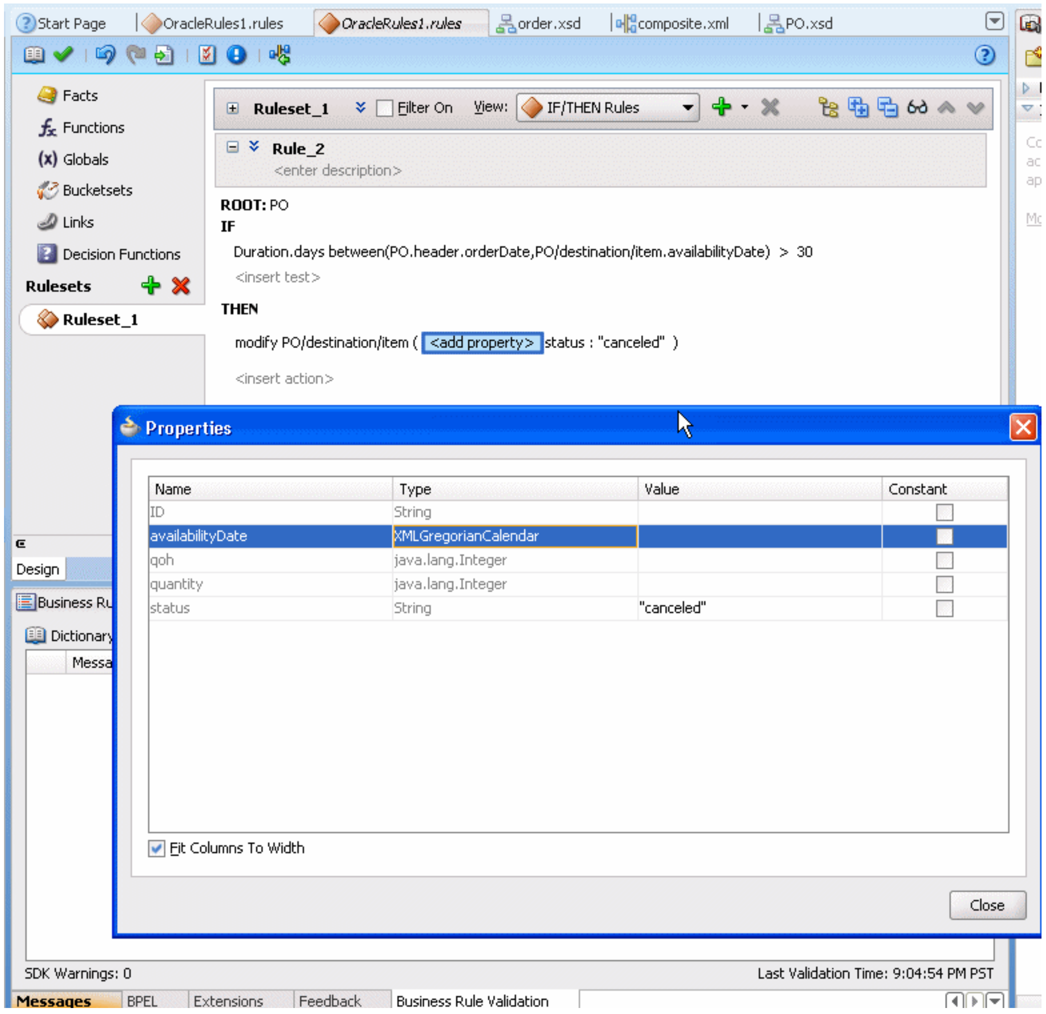
21. Click **<target>** and from the dropdown list select PO/destination/item. The THEN statement now reads:

    ```
    THEN modify PO/destination/item ( <add property> )
    ```

22. Click **<add property>**. This displays the properties dialog.

23. In the properties dialog for the status name, enter the value "canceled", as Figure 4–61 shows.

Figure 4–61 Simple Tree Mode: Action



Ruleset_1

Rule_2

<enter description>

ROOT: PO

IF

Duration.days between(PO.header.orderDate,PO/destination/item.availabilityDate) > 30

<insert test>

THEN

modify PO/destination/item (<add property> status : "canceled")

<insert action>

Properties

Name	Type	Value	Constant
ID	String		☐
availabilityDate	XMLGregorianCalendar		☐
qoh	java.lang.Integer		☐
quantity	java.lang.Integer		☐
status	String	"canceled"	☐

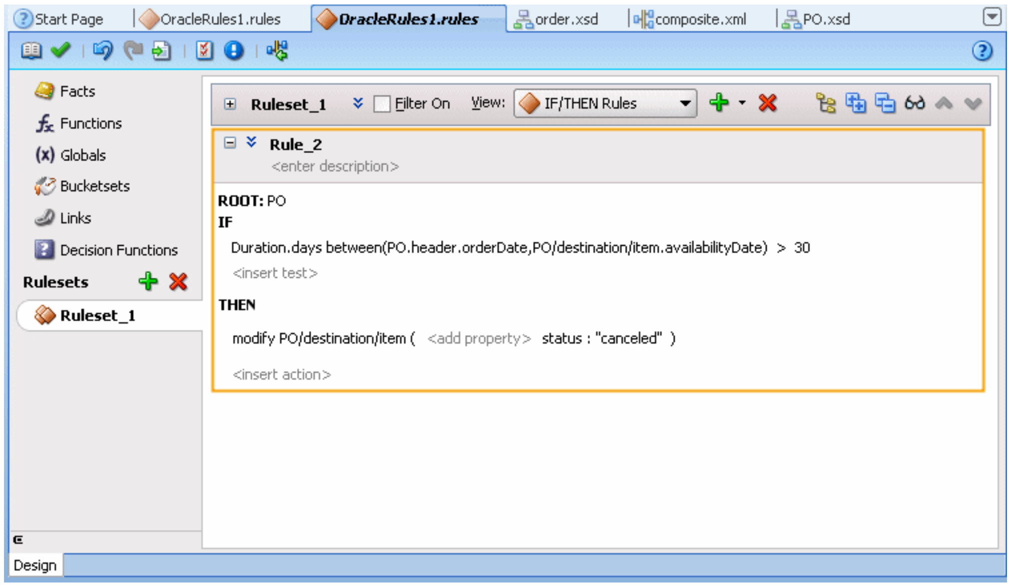
☑ Fit Columns To Width

SDK Warnings: 0

Last Validation Time: 9:04:54 PM PST

24. In the Properties dialog, click **Close**.
25. This displays the finished rule, as shown in Figure 4–62.

Figure 4–62 Simple Tree Mode Rule Final Rule



```
Rule_2
<enter description>
ROOT: PO
IF
  Duration.days between(PO.header.orderDate,PO/destination/item.availabilityDate) > 30
  <insert test>
THEN
  modify PO/destination/item ( <add property> status : "canceled" )
  <insert action>
```

Note that in the `modify` statement, `PO/destination/item` refers to the particular `item` instance member.

4.8.3 How to Create Advanced Tree Mode Rules

Given the XML schema shown in Example 4–1 and the instance of these facts shown in Example 4–2, 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.

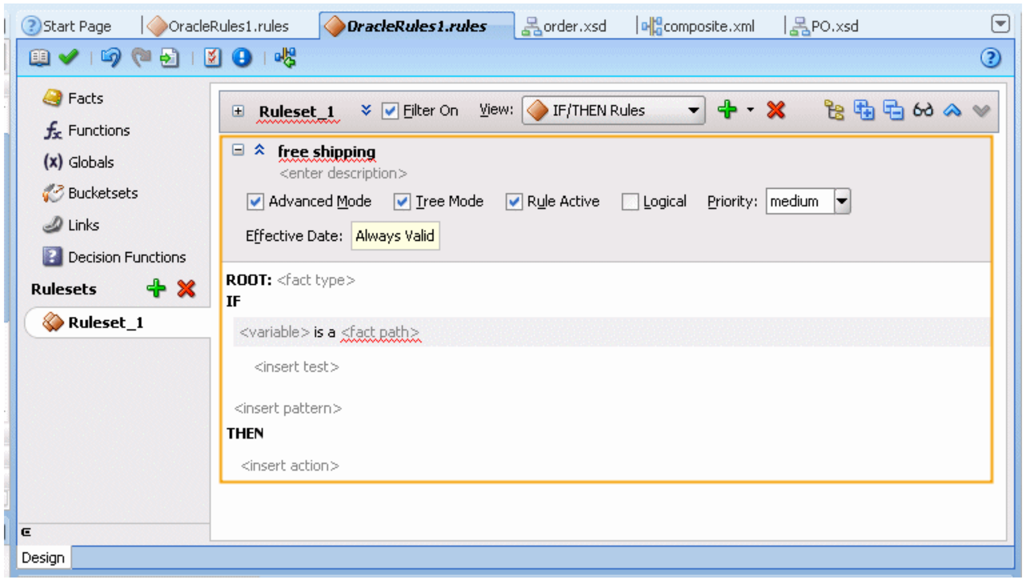
   For more information, see Section 4.3.1, "How to Add Rules".

2. View advanced settings.

   For more information, see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table".

3. Select **Advanced Mode** and select **Tree Mode** as Figure 4–63 shows.

Figure 4–63 Advanced Tree Mode Rule for Free Shipping



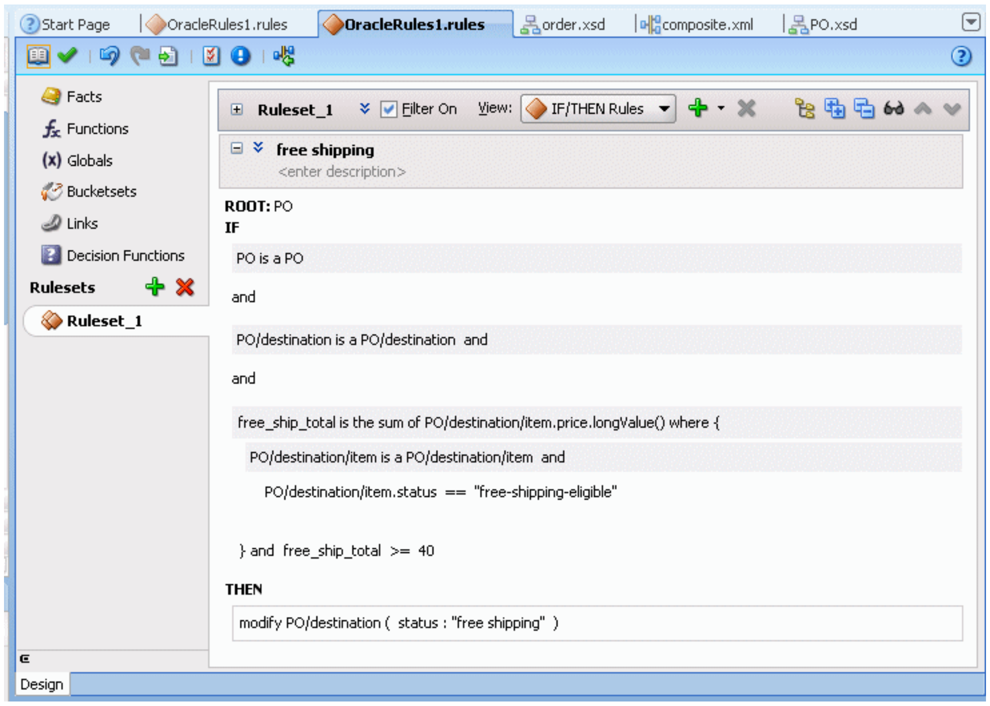
Ruleset_1 ☑ Filter On View: IF/THEN Rules

free shipping
<enter description>
☑ Advanced Mode ☑ Tree Mode ☑ Rule Active ☐ Logical Priority: medium
Effective Date: Always Valid

ROOT: <fact type>
IF
<variable> is a <fact path>
<insert test>
<insert pattern>
THEN
<insert action>

4. Select the **<fact type>** place holder and from the dropdown list, select **PO**.
5. Complete the free shipping rule, as shown in Figure 4–64.

Figure 4–64 Advanced Tree Mode Free Shipping Rule



Ruleset_1 ☑ Filter On View: IF/THEN Rules

free shipping
<enter description>

ROOT: PO
IF
PO is a PO
and
PO/destination is a PO/destination and
and
free_ship_total is the sum of PO/destination/item.price.longValue() where {
PO/destination/item is a PO/destination/item and
PO/destination/item.status == "free-shipping-eligible"
} and free_ship_total >= 40
THEN
modify PO/destination (status : "free shipping")

4.8.4 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.

4.9 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 4–8 describes the available **Effective Date** options.

Table 4–8 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.

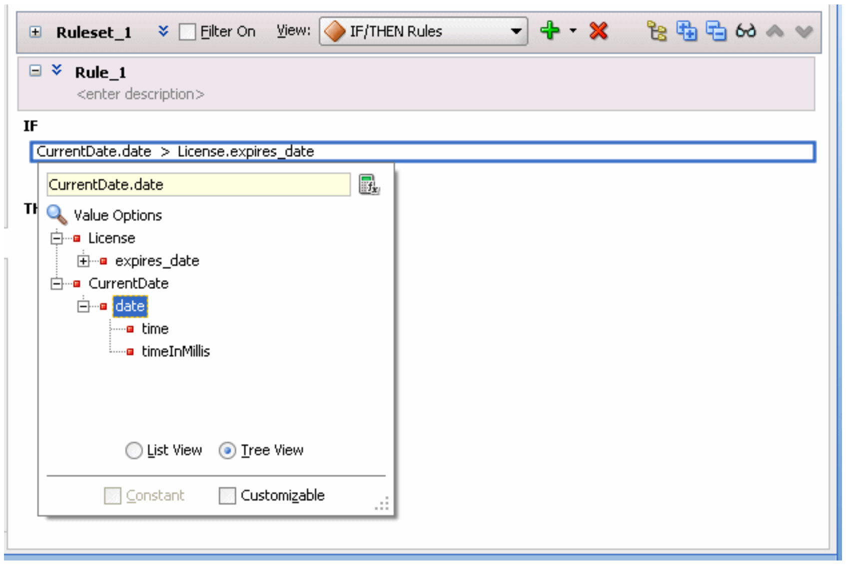
4.9.1 How to Use the Current Date Fact

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 4–65.

Figure 4–65 Rule with Condition Using CurrentDate Fact



CurrentDate.date > License.expires_date

CurrentDate.date

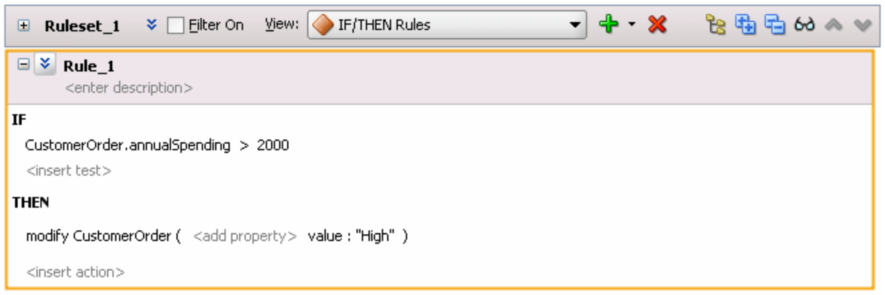
4.9.2 How to Set the Effective Date for a Rule

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 Section 4.2.2, "How to Set the Effective Date for a Ruleset".

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**, as shown highlighted in Figure 4–66.

Figure 4–66 Showing Advanced Settings in a Rule

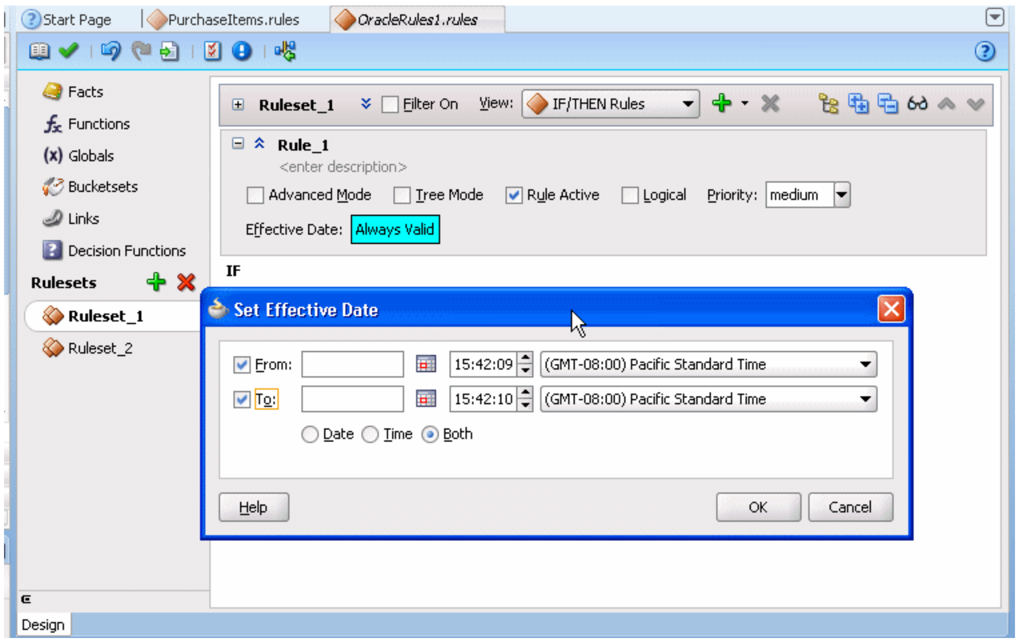


CustomerOrder.annualSpending > 2000

modify CustomerOrder (<add property> value : "High")

4. Select the **Effective Date** field. This displays the Set Effective Date dialog, as shown in Figure 4–67.

Figure 4–67 Setting the Effective Date for a Rule



Ruleset_1
Rule_1
<enter description>
☐ Advanced Mode ☐ Tree Mode ☑ Rule Active ☐ Logical Priority: medium
Effective Date: Always Valid
IF

Set Effective Date
☑ From: 15:42:09 (GMT-08:00) Pacific Standard Time
☑ To: 15:42:10 (GMT-08:00) Pacific Standard Time
Date Time [✓ Both]

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

6. In the Set Effective Date dialog, click **OK**.

4.9.3 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 *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*.

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 Section 4.5.3, "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 unselected, 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** checkbox, the **To** checkbox, or both checkboxes 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** unselected, 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.
   ```

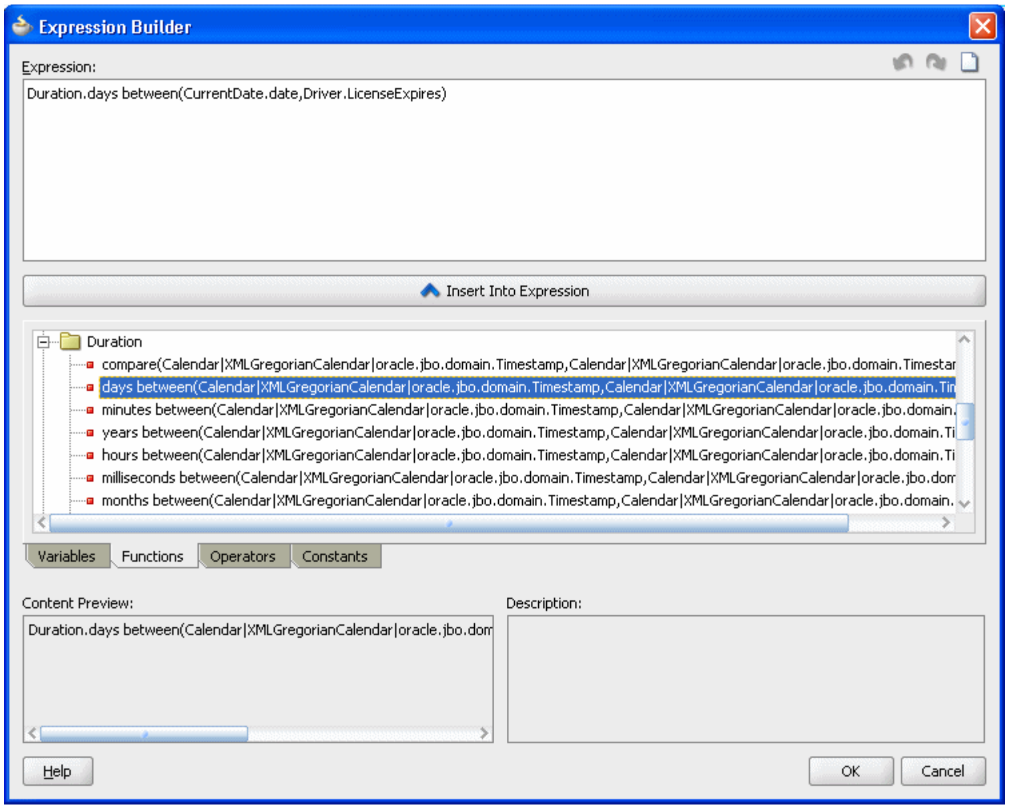
4.9.4 How to Use Duration, JavaDate, OracleDate, and XMLDate Methods

You can use the Duration, JavaDate, and XMLDate, OracleDate, and OracleDuration extension methods in a rule or a Decision Table. For more information, see Appendix B, "Rules Extension Methods".

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 dropdown list, select **Expression Builder...**. For more information, see Section 4.10, "Working with 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 4–68.
10. This allows you to create a rule such as that shown in Figure 4–69.

Figure 4–68 Using Duration Methods in a Rule



Expression:

```
Duration.days between(CurrentDate.date,Driver.LicenseExpires)
```

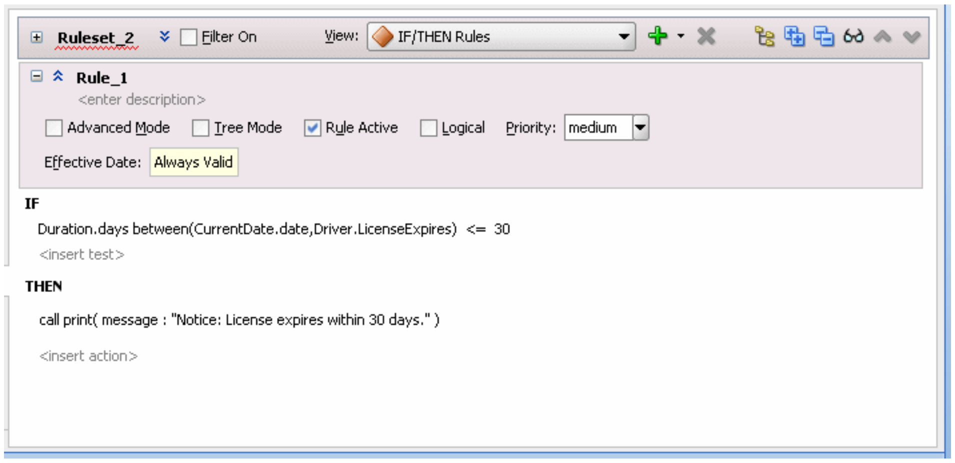
```
Duration
compare(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestar
days between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracle.jbo.domain.Tin
minutes between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracle.jbo.domain.
years between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracle.jbo.domain.Ti
hours between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracle.jbo.domain.Ti
milliseconds between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracle.jbo.dor
months between(Calendar|XMLGregorianCalendar|oracle.jbo.domain.Timestamp,Calendar|XMLGregorianCalendar|oracle.jbo.domain.
```

Content Preview:

```
Duration.days between(Calendar|XMLGregorianCalendar|oracle.jbo.dor
```

Description:

Figure 4–69 Adding a Rule Using Duration Function



Rule_1
<enter description>
Advanced Mode [ ] Tree Mode [ ] Rule Active [✓] Logical [ ] Priority: medium
Effective Date: Always Valid

```
IF
  Duration.days between(CurrentDate.date,Driver.LicenseExpires)  <=  30
  <insert test>
THEN
  call print( message : "Notice: License expires within 30 days." )
  <insert action>
```

4.10 Working with Expression Builder

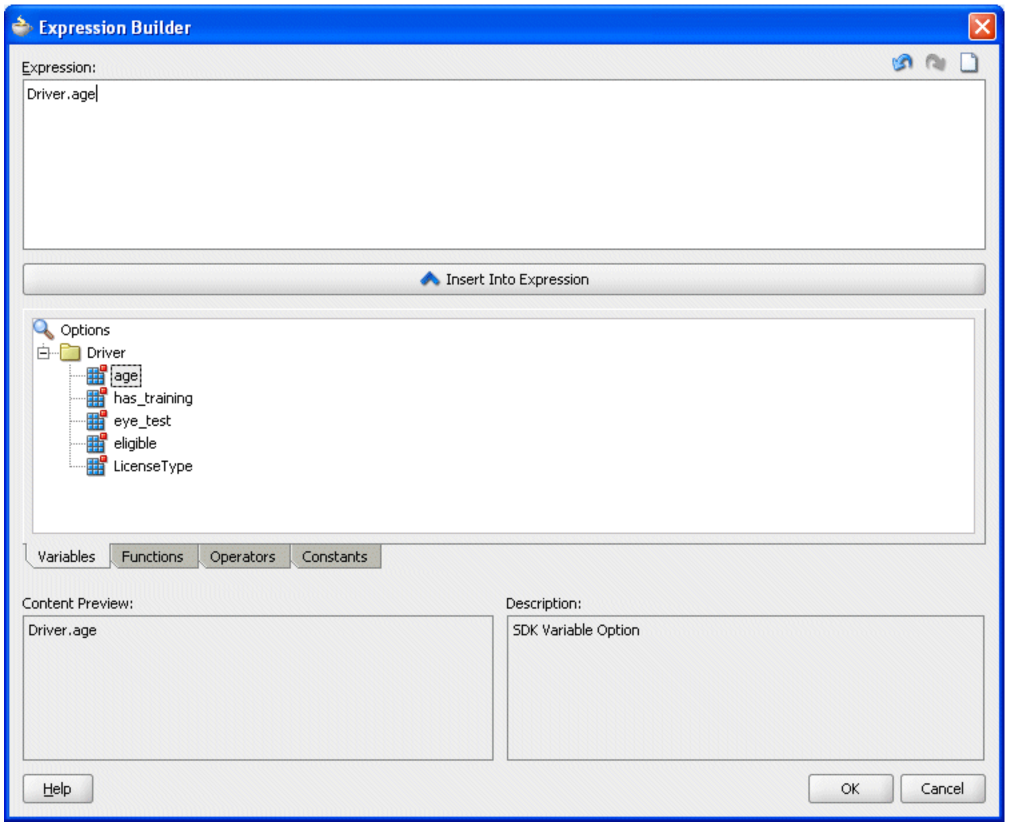
Use the expression builder to create and edit expressions for Oracle Business Rules.

4.10.1 Introduction to the 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 selected.

Figure 4–70 shows the Rules Designer expression builder.

Figure 4–70 Rules Designer Expression Builder



Driver.age

Options
Driver
age
has_training
eye_test
eligible
LicenseType

Driver.age

SDK Variable Option

4.10.2 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.

4.11 Using Bucketsets as Constraints for Options Values in Rules

You can use List of Values Bucketsets and List of Ranges Bucketsets to specify constraints for fact properties in rules. This allows 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 bucketset. Oracle Business Rules also lets you use bucketsets to specify constraints for global initial values, function return values, or function argument values. For more information, see

Section 2.3, "Working with Oracle Business Rules Globals" and Section 3.7, "Associating a Bucketset with Facts and Functions".

4.11.1 How to Use a List of Ranges Bucketset as a Constraint for a Fact Property

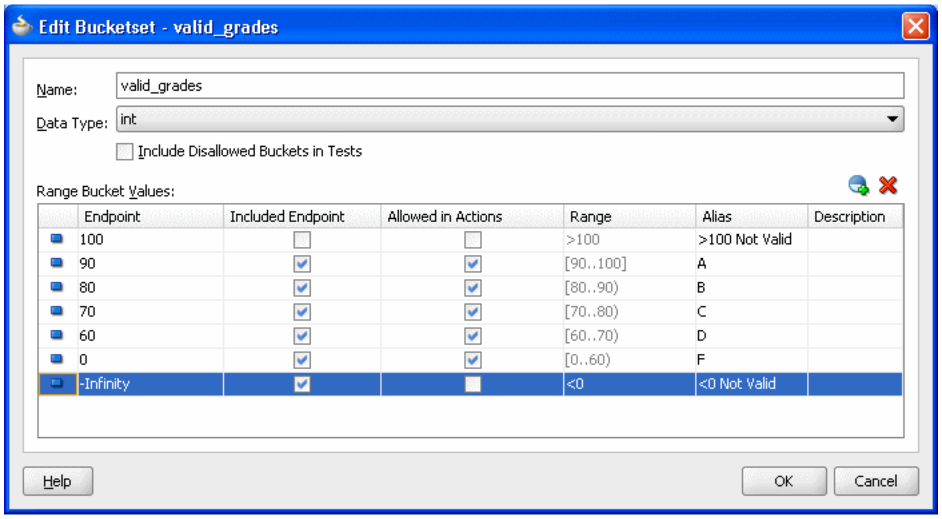
You can use a list of ranges bucketset as a constraint for a fact property.

For more information on using a list of values bucket set as a constraint, see Section 4.11.2, "How to Use a List of Values Bucketset as a Constraint for a Fact Property".

To use a List of Ranges bucketset as a constraint for a fact property:

1. Specify a bucketset that includes the ranges you want to include and select **Allowed in Actions** for all valid ranges. To include a range, deselect **Allowed in Actions** for the top and bottom endpoints.
2. Select **Included Endpoint** as needed for the application.
3. Deselect **Include Disallowed Buckets in Tests**. For example, for a bucketset that defines valid grades and that does not allow values greater than 100, or less than 0, define the bucketset endpoints as shown in Figure 4–71.

Figure 4–71 Valid Grades Bucketset for Fact Property



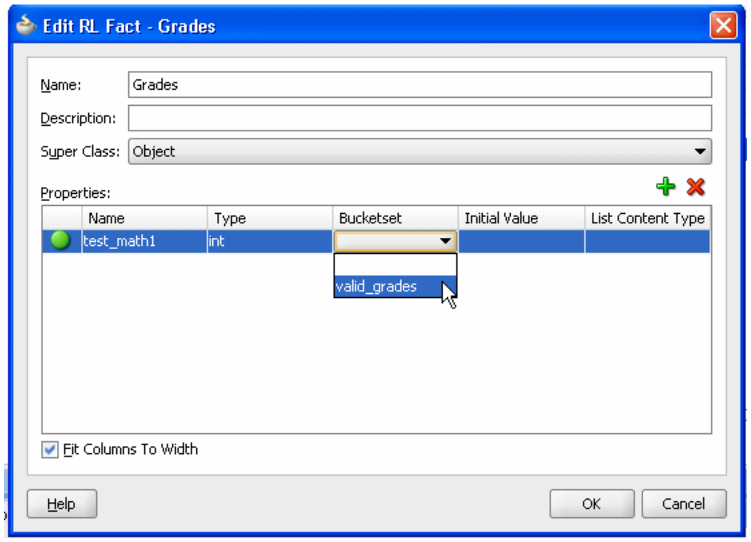
Name: valid_grades
Data Type: int
☐ Include Disallowed Buckets in Tests

Range Bucket Values:

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
100	☐	☐	>100	>100 Not Valid	
90	✓	✓	[90..100]	A	
80	✓	✓	[80..90)	B	
70	✓	✓	[70..80)	C	
60	✓	✓	[60..70)	D	
0	✓	✓	[0..60)	F	
-Infinity	✓	☐	<0	<0 Not Valid	

4. Associate this bucketset with a fact property. For example, associate the bucketset with test_math1 as shown in Figure 4–72.

Figure 4–72 Associating a Bucketset with a Fact Property



Name: Grades

Description:

Super Class: Object

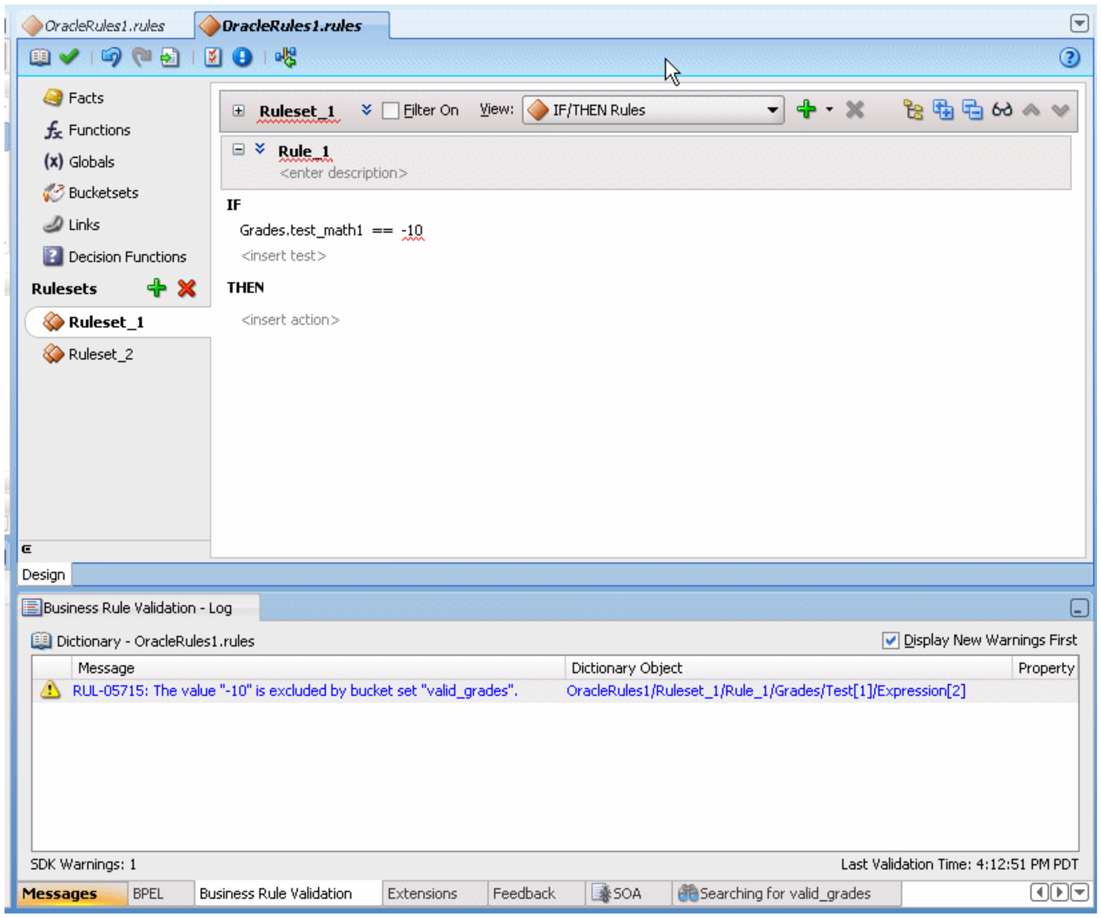
Properties:

	Name	Type	Bucketset	Initial Value	List Content Type
●	test_math1	int	valid_grades		

☑ Fit Columns To Width

Now, if you define a rule with a test that uses the fact property you 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 shows a validation warning as shown in Figure 4–73.

Figure 4–73 Using a Fact Property Value that is not in the Allowed in Actions for Associated Bucketset



Rule_1
<enter description>

IF
Grades.test_math1 == -10
<insert test>

THEN
<insert action>

Business Rule Validation - Log

Dictionary - OracleRules1.rules ☑ Display New Warnings First

Message	Dictionary Object	Property
RUL-05715: The value "-10" is excluded by bucket set "valid_grades".	OracleRules1/Ruleset_1/Rule_1/Grades/Test[1]/Expression[2]	

SDK Warnings: 1 Last Validation Time: 4:12:51 PM PDT

4.11.2 How to Use a List of Values Bucketset as a Constraint for a Fact Property

You can use a list of values bucketset as a constraint for a fact property.

For more information on using a list of ranges bucket set as a constraint, see Section 4.11.1, "How to Use a List of Ranges Bucketset as a Constraint for a Fact Property".

To use a List of Values bucketset as a constraint for a fact property:

1. Specify an LOV bucketset that includes the values you want to include, and select **Allowed in Actions** for all valid values. For more information, see Section 3.6.1, "How to Define a List of Values Global Bucketset".
2. Deselect **Allowed in Actions** for the otherwise bucket.
3. Deselect **Include Disallowed Buckets in Tests**.
4. Associate this bucketset with a fact property.

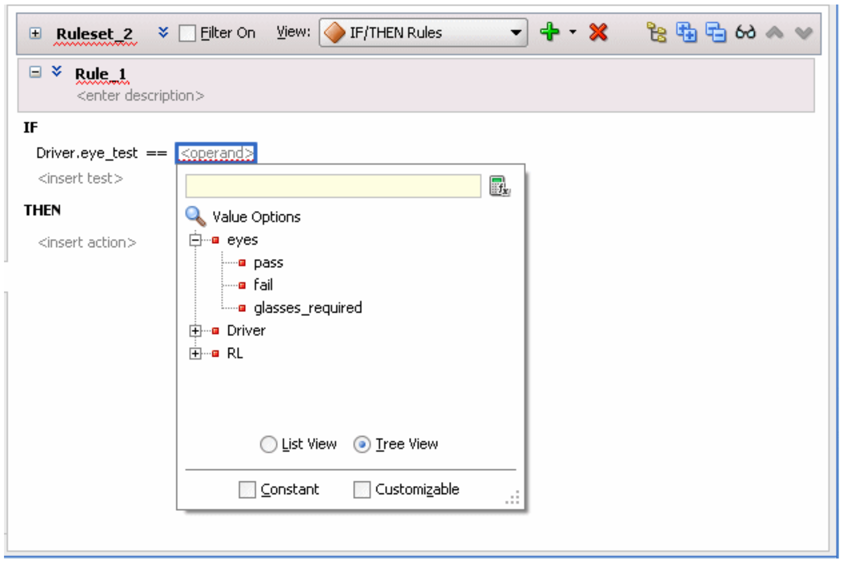
4.11.3 How to Use Bucketsets to Provide Options for Test Expressions

You can use LOV bucketsets to provide options for expressions and actions.

How to use bucketsets to provide options for rule expressions and actions:

1. In Rules Designer, define an LOV bucketset of a type corresponding to a fact property. For more information, see Section 3.6.1, "How to Define a List of Values Global Bucketset".
2. Associate the bucketset with a fact property. For more information, see Section 3.7.1, "How to Associate a Bucketset with a Fact Property".
3. When you enter expressions, Rules Designer shows the bucket values in the values options. For example, when you associate a fact property `Driver.eye_test` with an LOV bucketset named `eyes`, with values: `pass`, `fail`, and `glasses_required`, and then you use `Driver.eye_test` in a test expression, the bucket values are limited as shown in Figure 4–74.

Figure 4–74 Using a Bucketset to Provide Options for a Rule Test Expression



5 Working with Decision Tables

Using a Decision Table you can create and use business rules in an easy to understand format that provides an alternative to the IF/THEN rule format. 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.

This chapter includes the following sections:

5.1 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.

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, incompleteness, or gaps, help speed development of valid, consistent business rules.

Ease of verification and visualization are the major reasons for using Decision Tables.

For information, see Chapter 4, "Working with Rulesets and Rules".

5.1.1 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 icons and selecting values in Rules Designer to help identify and correct conflicting or missing cases.

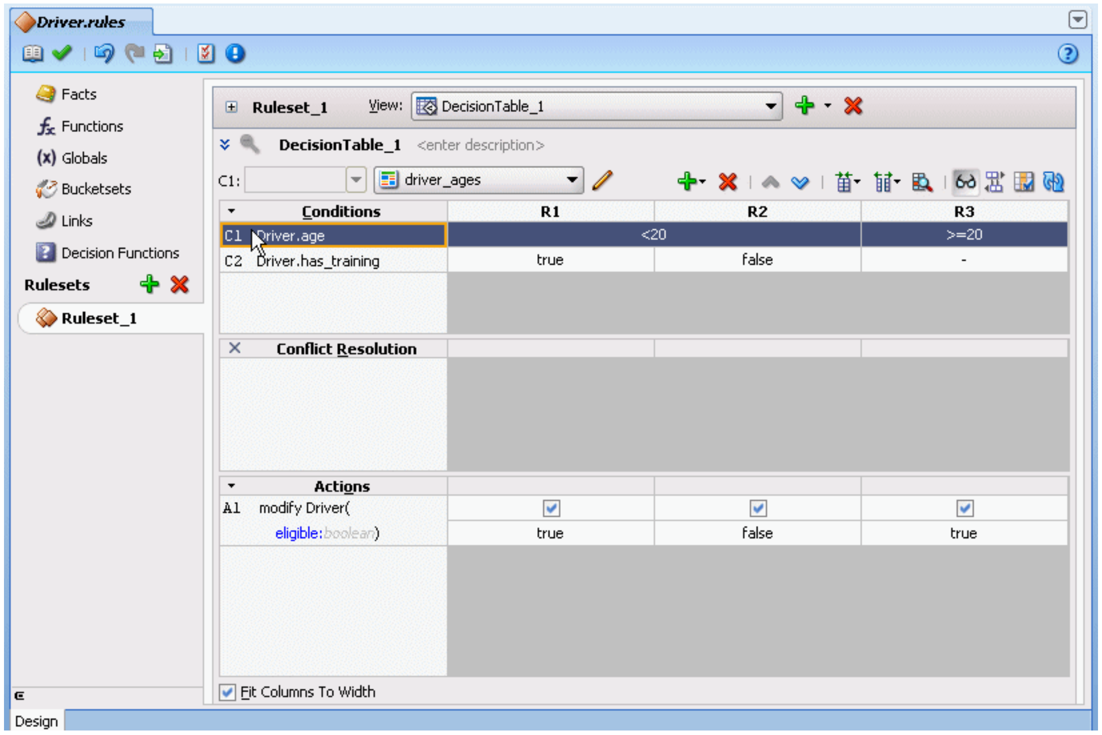
To help understand Decision Table concepts, consider a set of IF/THEN rules that determine if a driver is eligible for a license, and an equivalent Decision Table. Note if a driver has taken a driver training class then the driver has training certification.

The IF/THEN rules follow:

```
if driver.age < 20 and driver.has training then driver.eligible = 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



Conditions	R1	R2	R3
C1 Driver.age	<20		>=20
C2 Driver.has_training	true	false	-
Conflict Resolution			
Actions			
A1 modify Driver(	✓	✓	✓
eligible:boolean)	true	false	true

5.1.1.1 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. Usually a condition expression is shown in a condition row and applies to a data model fact, either, a Java Fact, an XML Fact, an RL Fact, or an ADF Business Components fact. A bucketset is associated with every condition expression and you can define the bucketset on the fly using a local bucketset or you can use a global bucketset. The value or the range for a given **condition cell** takes its value or its range from one or more buckets in the associated LOV or Ranges bucketset. For more information on bucketsets, see Section 3.6, "Working with Bucketsets".

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 buckets for the ranges `<20`, and `>=20`. The values in the cells come from the global bucketset 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 bucketset associated with boolean types (this consists of default buckets for the values `true` and `false`).

Decision Tables show rules in bucket order, and to change the order of rules you need to change the order of buckets in the bucketsets. Thus, the order of the buckets in the bucketset 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 bucketset associated with a condition row; however, you cannot reorder range buckets. For information on ordering buckets in a bucketset, see Section 3.6.1, "How to Define a List of Values Global Bucketset".

5.1.1.2 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. In the **Actions** area the cells corresponding to an individual action for a rule are called **action cells**. Note, in advanced mode there are additional options for actions. For more information on advanced mode, see Section 4.5.2, "How to Select the Advanced Mode Option".

Table 5–1 Decision Table Actions for Action Cells

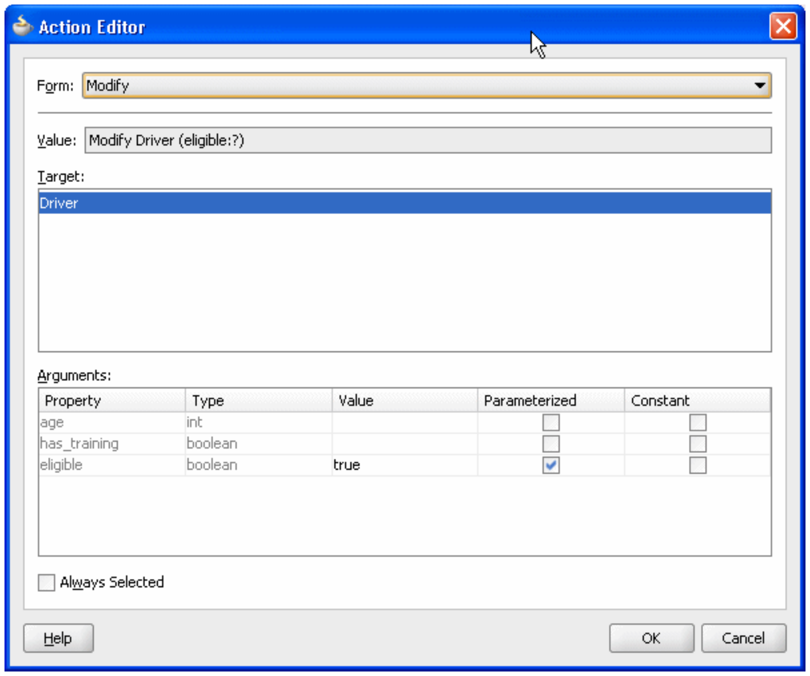
Action	Description
`assert new`	Assert a new fact
`call`	Call a function
`retract`	Retract a fact
`modify`	Modify a data value associated with a matched fact

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** checkbox. This checkbox 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 checkbox 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



Form: Modify

Value: Modify Driver (eligible:?)

Target:

Driver

Arguments:

Property	Type	Value	Parameterized	Constant
age	int		☐	☐
has_training	boolean		☐	☐
eligible	boolean	true	✓	☐

☐ Always Selected

5.1.1.3 What You Need to Know About Decision Table Rules

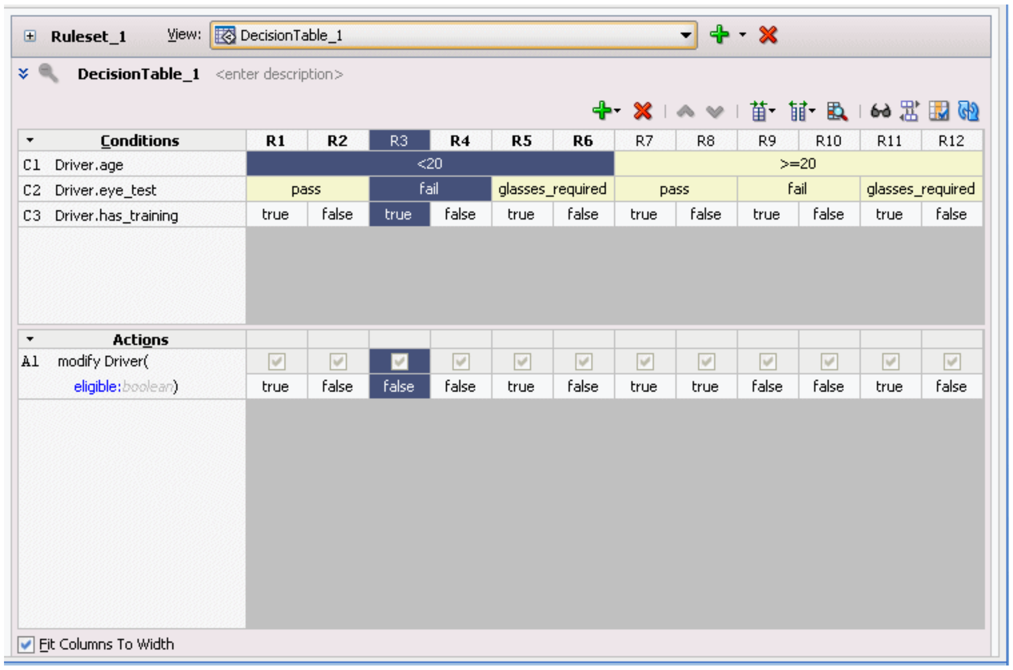
A ruleset contains a Decision Table; this provides a way to group the Decision Table along with IF/THEN rules. When rules and Decision Tables are grouped in a ruleset, the IF/THEN rules and the Decision Table rules all execute as a set of interrelated rules.

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` bucket), the second cell from condition 2 (the `Driver.eye_test` equals `fail` bucket), and the third cell from condition 3 (the `Driver.has_training` equals `true` bucket). 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



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

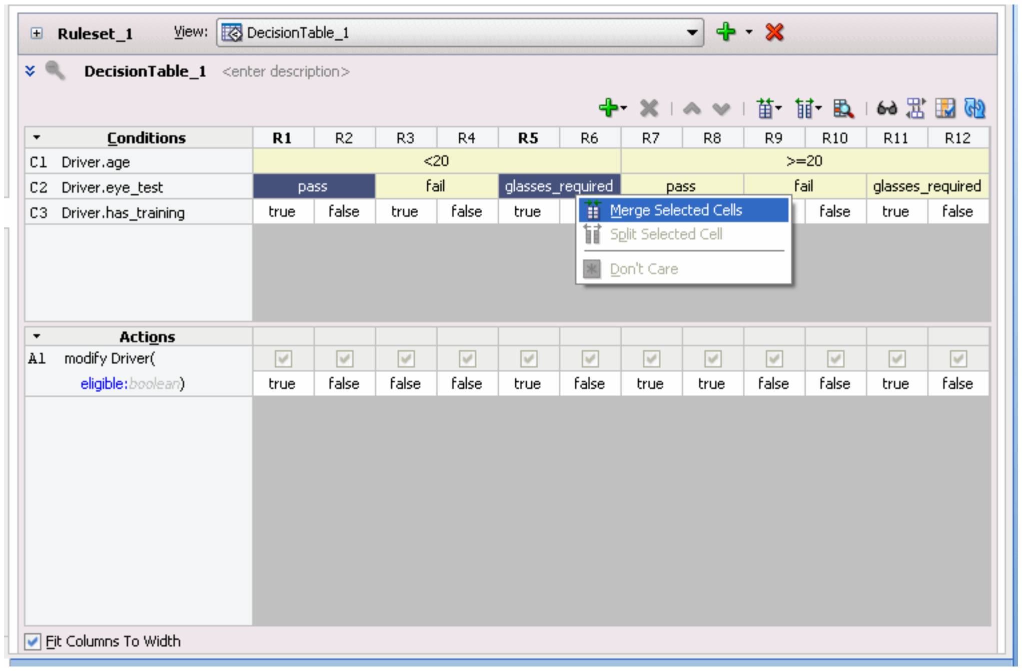
Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
C1 Driver.age	<20						>=20					
C2 Driver.eye_test	pass		fail		glasses_required		pass		fail		glasses_required	
C3 Driver.has_training	true	false	true	false	true	false	true	false	true	false	true	false

Actions												
A1 modify Driver(	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
eligible:boolean)	true	false	false	false	true	false	true	true	false	false	true	false

Fit Columns To Width

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 bucket 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 Section 5.3.3, "How to Merge or Split Conditions in a Decision Table".

Figure 5–4 Sibling Condition Cells in a Decision Table



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
C1 Driver.age	<20						>=20					
C2 Driver.eye_test	pass		fail		glasses_required		pass		fail		glasses_required	
C3 Driver.has_training	true	false	true	false	true					false	true	false

Actions												
A1 modify Driver(	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
eligible:boolean)	true	false	false	false	true	false	true	true	false	false	true	false

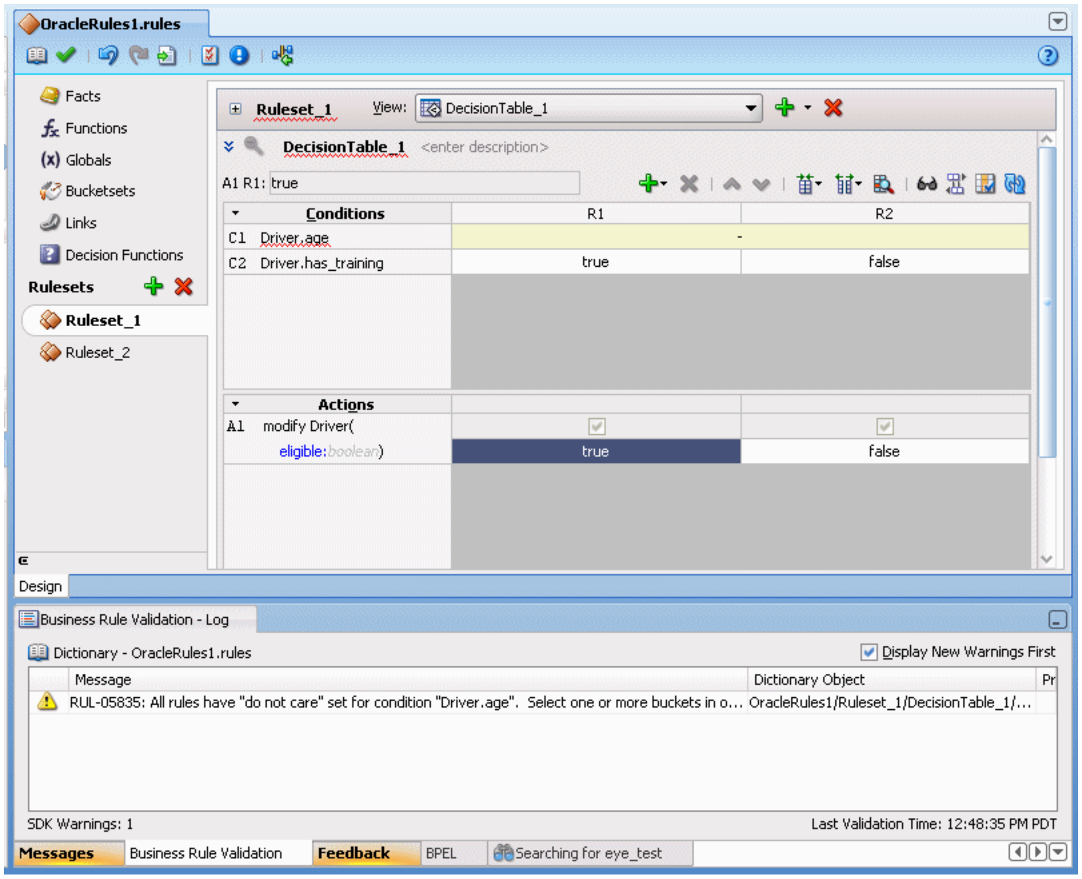
Fit Columns To Width

Rules Designer lets you easily reorder rows by selecting the row and clicking a **Move** icon. 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.

5.1.2 Understanding Decision Table 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 "-" means "do not care" or "match any bucket" in the bucketset. For example, Figure 5–5 shows a "do not care" value for `Driver.age`.

Figure 5–5 Sample Decision Table Showing Do Not Care Value in Condition Cell



Ruleset_1 View: DecisionTable_1

DecisionTable_1

A1 R1: true

Conditions	R1	R2
C1 Driver.age	-	
C2 Driver.has_training	true	false

Actions		
A1 modify Driver(	✓	✓
eligible:boolean)	true	false

Dictionary - OracleRules1.rules

Message	Dictionary Object	Pr
RUL-05835: All rules have "do not care" set for condition "Driver.age". Select one or more buckets in o...	OracleRules1/Ruleset_1/DecisionTable_1/...	

SDK Warnings: 1

Last Validation Time: 12:48:35 PM PDT

In the Decision Table **Actions** area you can specify that an action cell "do nothing". In this case, deselect the action cell. When the action cell checkbox is unselected 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 bucket that has been removed from the bucketset, Rules Designer provides a validation warning such as the following:

```
RUL-05831: Decision table bucket 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 buckets from the dropdown list.
- Add the missing bucket to the bucketset or associate the condition with another bucketset that contains the missing bucket.

5.1.3 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.

5.2 Creating Decision Tables

You add a Decision Table by performing several steps. These steps include:

- Create a Decision Table
- Add conditions to the Decision Table
- Add actions to the Decision Table
- Use Decision Table operations to validate, correct, and modify the Decision Table

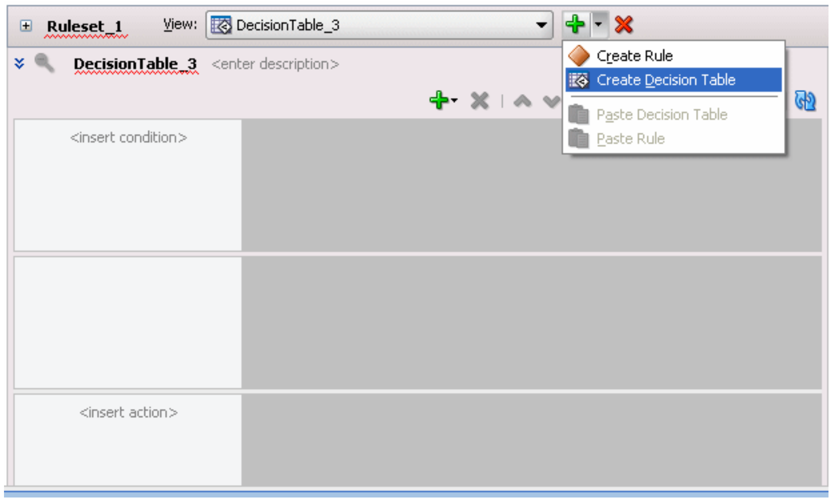
5.2.1 How to Create a Decision Table

To work with a Decision Table you 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 Ruleset...**.
2. Click the **Add** icon and from the dropdown list select **Create Decision Table**, 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.

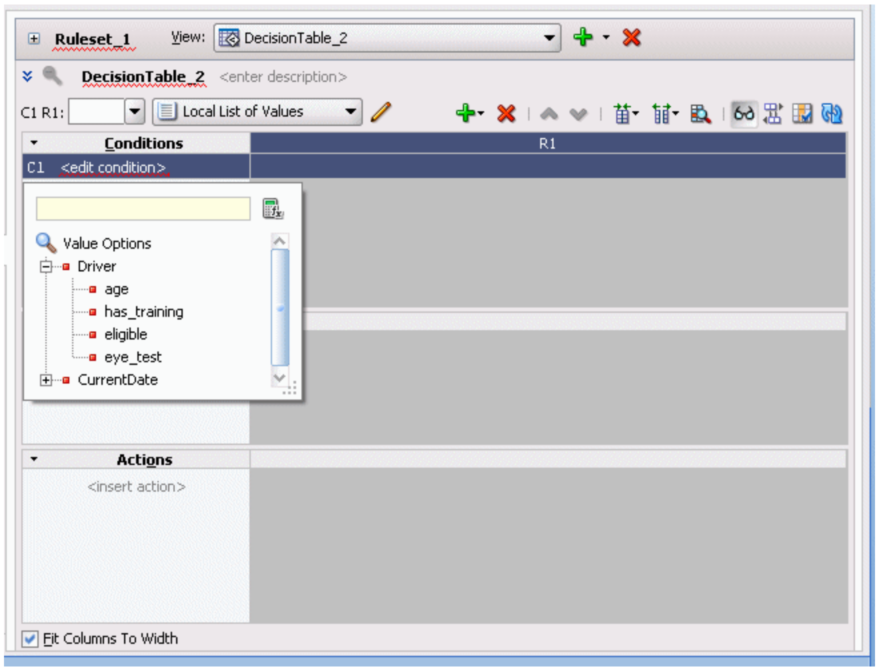
5.2.2 How to Add Condition Rows to a Decision Table

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 dropdown list next to the **Add** icon 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



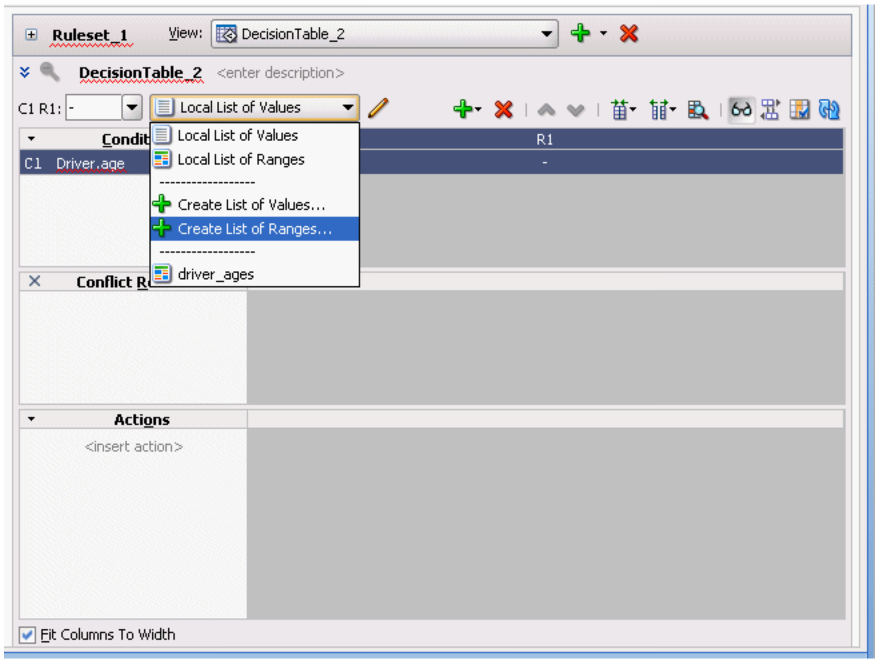
4. Enter an expression by clicking in the navigator to select a variable or click the **Expression Builder** icon to display the **Expression Builder** window. The **Expression Builder** lets you build expressions.
5. Each condition row requires a bucketset from which to draw the values for each cell. When the value you select has an associated global bucketset, then by default the bucketset is associated with the condition row.
6. Repeat **Step 2** through **Step 5**, as required to add additional condition rows in the Decision Table.

To use a local bucketset or specify the bucketset for a decision table condition:

1. Each condition row requires a bucketset from which to draw the values for each cell. When the value you select has an associated global bucketset, then by default the bucketset is associated with the condition row.

2. If there is no global bucketset 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 bucketset to associate with the condition row, or specify an existing global bucketset. To add a bucketset for the condition, in the **Conditions** area select the condition and then select from the Bucketset dropdown list to associate a bucketset, as shown in Figure 5–8. The bucketset list includes available global bucketsets of the appropriate type.

Figure 5–8 Specifying a Bucketset For a Condition Row in a Decision Table



3. If you do not specify a global bucketset, then you can create and use a local bucketset by selecting either **Local List of Values** or **Local List of Ranges** to create and use the specified type of bucketset.

4. Repeat **Step 2** through **Step 3**, as required to define additional condition rows in the Decision Table.

For more information on creating bucketsets, see Section 3.6, "Working with Bucketsets".

5.2.3 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 you need to do the following:

1. Add actions to a Decision Table.

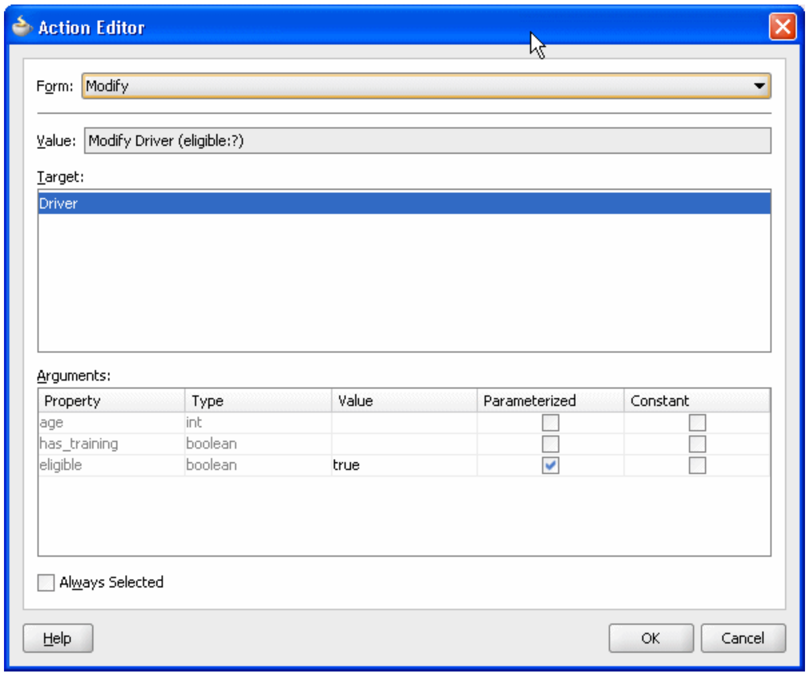
2. For each action cell, where specific values apply, set the values for the action cells.

3. 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 dropdown list next to the **Add** icon select **Action** and select an available action from the dropdown 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



Property	Type	Value	Parameterized	Constant
age	int		☐	☐
has_training	boolean		☐	☐
eligible	boolean	true	✓	☐

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 dropdown 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.

Table 5–2 (Cont.) Action Editor Dialog Arguments Fields

Field	Description
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** checkbox.
6. Repeat **Step 2** through **Step 5**, as required to define additional actions for the Decision Table.

To set values for action cells 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 specify action cell values.
2. In the **Actions** area, check that the appropriate action cells are selected. If the **Always Selected** checkbox 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 checkbox.
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 dropdown list, or click the **Expression Builder** icon to use the Expression Builder dialog.

To deselect an action cell in a decision table:

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 checkbox 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** icon or **Move Up** icon to change the order of actions.

5.2.4 How to Add a Rule to a Decision Table

You can add a rule to a Decision Table. Rules Designer shows a column for the rule and each condition cell is initialized to "-" do not care.

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 dropdown list next to the **Add** icon, select **Rule**.
3. Enter values for the condition cells. Notice that the rule moves as required to keep bucket values in their defined order.
4. Enter values for the action cells.

5.3 Performing Operations on Decision Tables

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 in a Decision Table

5.3.1 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.

5.3.1.1 Understanding Decision Table Split and Compact Operations

The split and compact operations allow you to manipulate the contents of the condition cells in a Decision Table.

The split table operation creates a rule for every combination of buckets 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 bucket 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 bucket in the bucketset 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 bucketset. 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

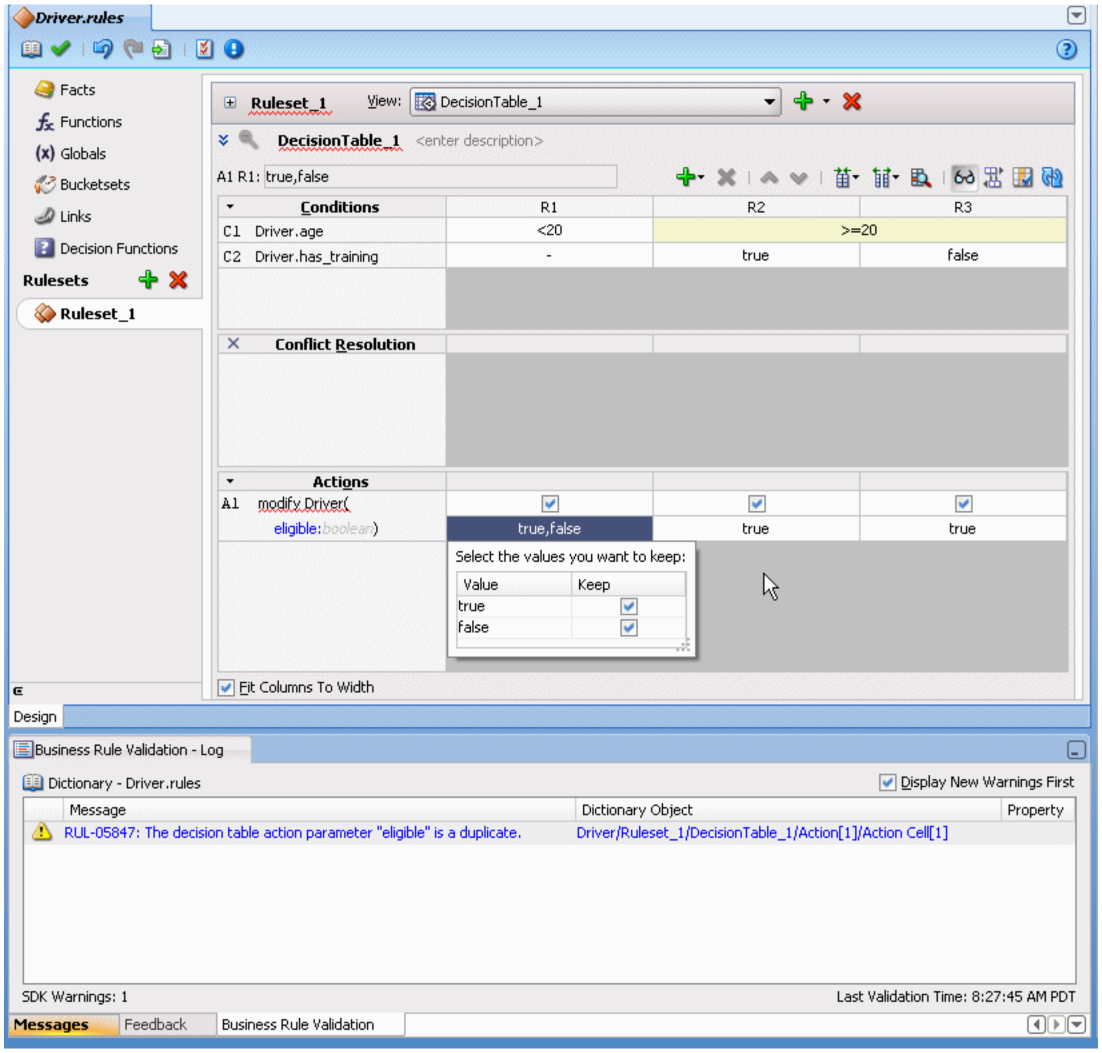
Operator	Description
Condition Cell	Merging two or more condition cells adds all buckets in the cells to a single cell, and removes all but one of the cells. If one of the cells represents "do not care", then the merged cell represents "do not care". This operation may merge action cells and this can create warnings for duplicate action cells, such as, `RUL-05847: Duplicate decision table action parameter`.
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 buckets 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 5–10 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 checkboxes to allow you to select a single value for the action cell. As shown in the validation log in Figure 5–10, Rules Designer shows a validation warning until you select a single value.

Figure 5–10 Conflicting Properties to be Resolved for a Merged Action Cell



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

A1 R1: true,false

Conditions	R1	R2	R3
C1 Driver.age	<20	>=20	
C2 Driver.has_training	-	true	false

Conflict Resolution

Actions			
A1 modify Driver(	✓	✓	✓
eligible:boolean)	true,false	true	true

Select the values you want to keep:

Value	Keep
true	✓
false	✓

Fit Columns To Width

Business Rule Validation - Log

Dictionary - Driver.rules

Display New Warnings First

Message	Dictionary Object	Property
RUL-05847: The decision table action parameter "eligible" is a duplicate.	Driver/Ruleset_1/DecisionTable_1/Action[1]/Action Cell[1]	

SDK Warnings: 1

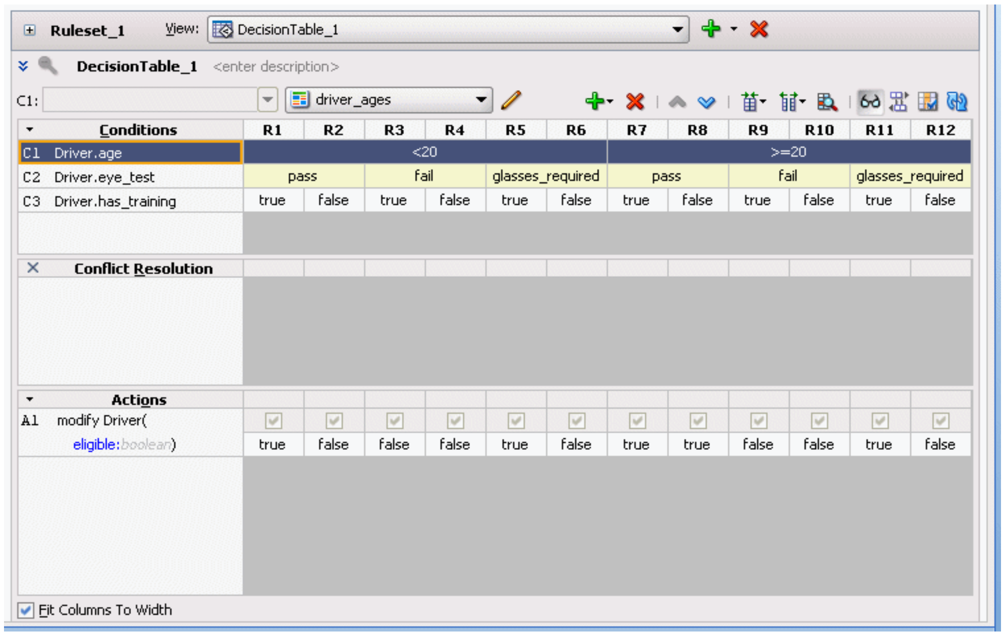
Last Validation Time: 8:27:45 AM PDT

5.3.1.2 Understanding Decision Table Move Operations

You can move the conditions or actions in a Decision Table. The **Move** icons 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** icon 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 5–11.

Figure 5–11 Rules in a Decision Table



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

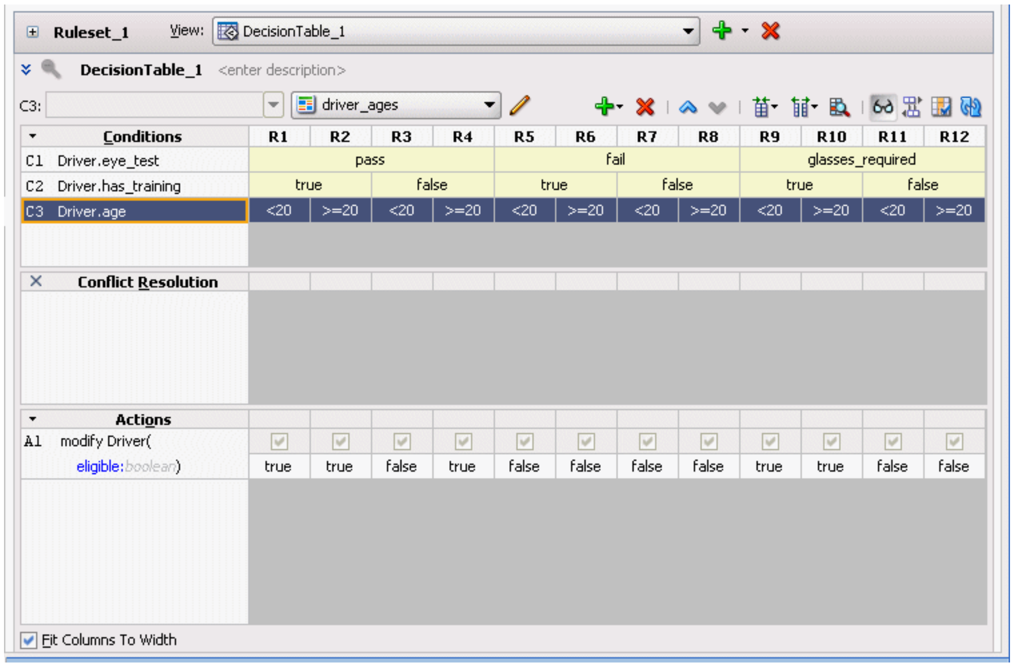
C1: driver_ages

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
C1 Driver.age	<20						>=20					
C2 Driver.eye_test	pass		fail		glasses_required		pass		fail		glasses_required	
C3 Driver.has_training	true	false	true	false	true	false	true	false	true	false	true	false
Conflict Resolution												
Actions												
A1 modify Driver(eligible:boolean)	true	false	false	false	true	false	true	true	false	false	true	false

Fit Columns To Width

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 5–12.

Figure 5–12 Decision Table After Move Down with Age Condition Last



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

C3: driver_ages

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
C1 Driver.eye_test	pass				fail				glasses_required			
C2 Driver.has_training	true		false		true		false		true		false	
C3 Driver.age	<20	>=20	<20	>=20	<20	>=20	<20	>=20	<20	>=20	<20	>=20
Conflict Resolution												
Actions												
A1 modify Driver(eligible:boolean)	true	true	false	true	false	false	false	false	true	true	false	false

Fit Columns To Width

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.

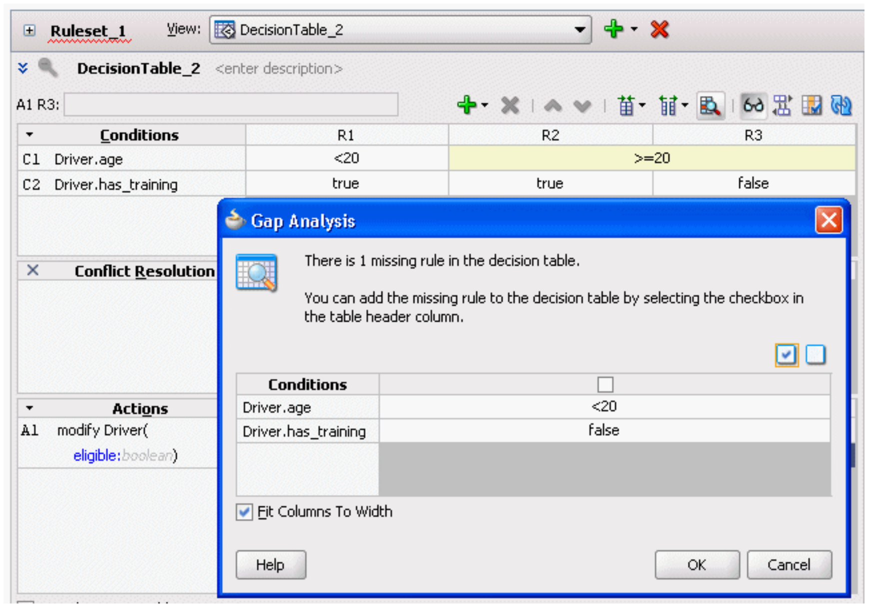
5.3.1.3 Understanding Decision Table Gap Analysis

A gap is a "missing" rule in a Decision Table. A Decision Table has a gap if there is a combination of buckets, one from each condition, that is not covered by an existing rule. Rules Designer provides **Gap Analysis** to check for gaps. When you click the **Gap Analysis** icon 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 deselect **Allow Gaps** in the **Advanced Settings** area, the Decision Table reports a validation warning when a gap is found. For more information, see Section 4.5, "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 5–13. Clicking **OK** with the checkboxes selected adds either all rules or the selected rules to the Decision Table (this example only shows a single rule to add).

Figure 5–13 Using Gap Analysis



Conditions	R1	R2	R3
C1 Driver.age	<20	>=20	
C2 Driver.has_training	true	true	false

A1 modify Driver(
eligible:boolean)

There is 1 missing rule in the decision table.

You can add the missing rule to the decision table by selecting the checkbox in the table header column.

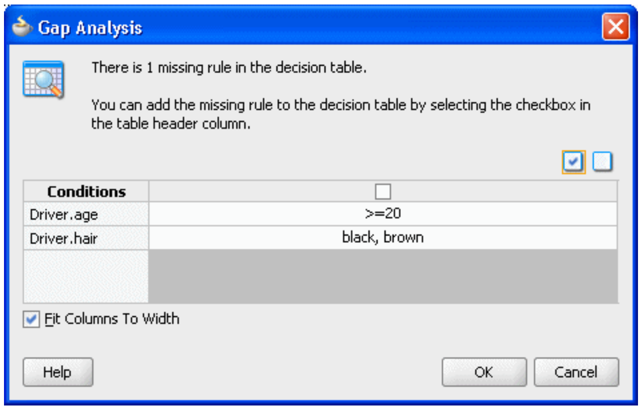
Conditions	☐
Driver.age	<20
Driver.has_training	false

Gap analysis 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 analysis shows a single rule as shown in Figure 5–14.

- 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 analysis provides two rules, as shown in Figure 5–15.

Figure 5–14 Gap Analysis with Missing Sibling Rules



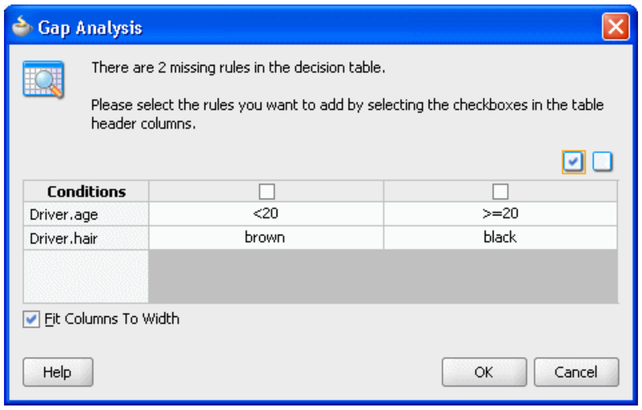
There is 1 missing rule in the decision table.

You can add the missing rule to the decision table by selecting the checkbox in the table header column.

Conditions	☐
Driver.age	>=20
Driver.hair	black, brown

☑ Fit Columns To Width

Figure 5–15 Gap Analysis with Missing Non-Sibling Rules



There are 2 missing rules in the decision table.

Please select the rules you want to add by selecting the checkboxes in the table header columns.

Conditions	☐	☐
Driver.age	<20	>=20
Driver.hair	brown	black

☑ Fit Columns To Width

In both of these cases shown in Figure 5–14 and Figure 5–15 there are two missing buckets, but for sibling rules the multiple buckets are combined in a single new rule. Thus, in general gap analysis suggests fewer more general rules in preference to many more specific rules.

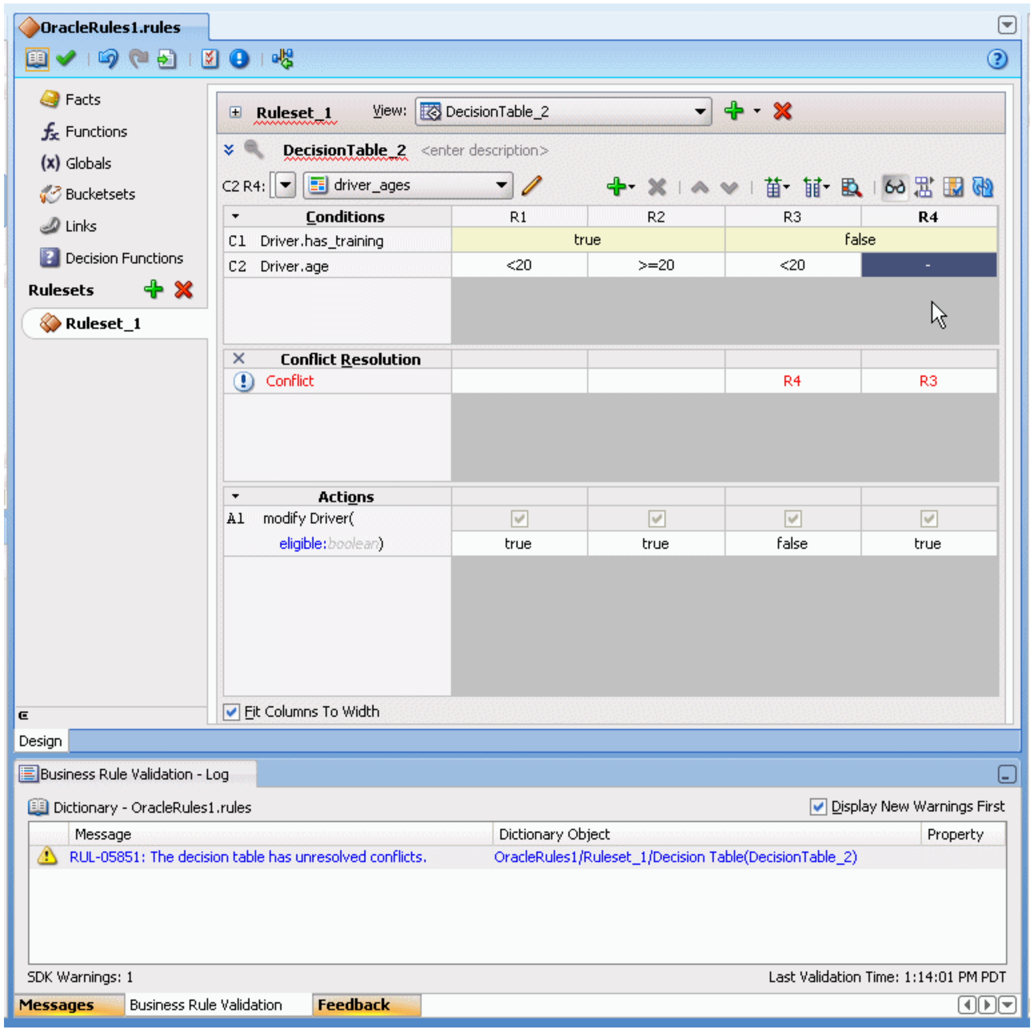
For sibling rules you can add multiple rules then edit each cell to pick the buckets you want. Alternatively, you can use **Find Gaps** to add a rule and then split the cell with multiple values, and delete the rules you do not want to keep.

5.3.1.4 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 bucket 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**. For example, Figure 5–16 shows a Decision Table with conflicting rules.

Figure 5–16 Decision Table Showing Conflicting Rules in the Conflicts Area



Ruleset_1 View: DecisionTable_2

DecisionTable_2 <enter description>

C2 R4: driver_ages

Conditions	R1	R2	R3	R4
C1 Driver.has_training	true		false	
C2 Driver.age	<20	>=20	<20	-
Conflict Resolution				
Conflict			R4	R3
Actions				
A1 modify Driver(	✓	✓	✓	✓
eligible:boolean)	true	true	false	true

Fit Columns To Width

Business Rule Validation - Log

Dictionary - OracleRules1.rules   Display New Warnings First

Message	Dictionary Object	Property
RUL-05851: The decision table has unresolved conflicts.	OracleRules1/Ruleset_1/Decision Table(DecisionTable_2)	

SDK Warnings: 1   Last Validation Time: 1:14:01 PM PDT

By clicking on the cells in the Decision Table **Conflict Resolution** area Rules Designer lets you resolve conflicts between rules as follows:

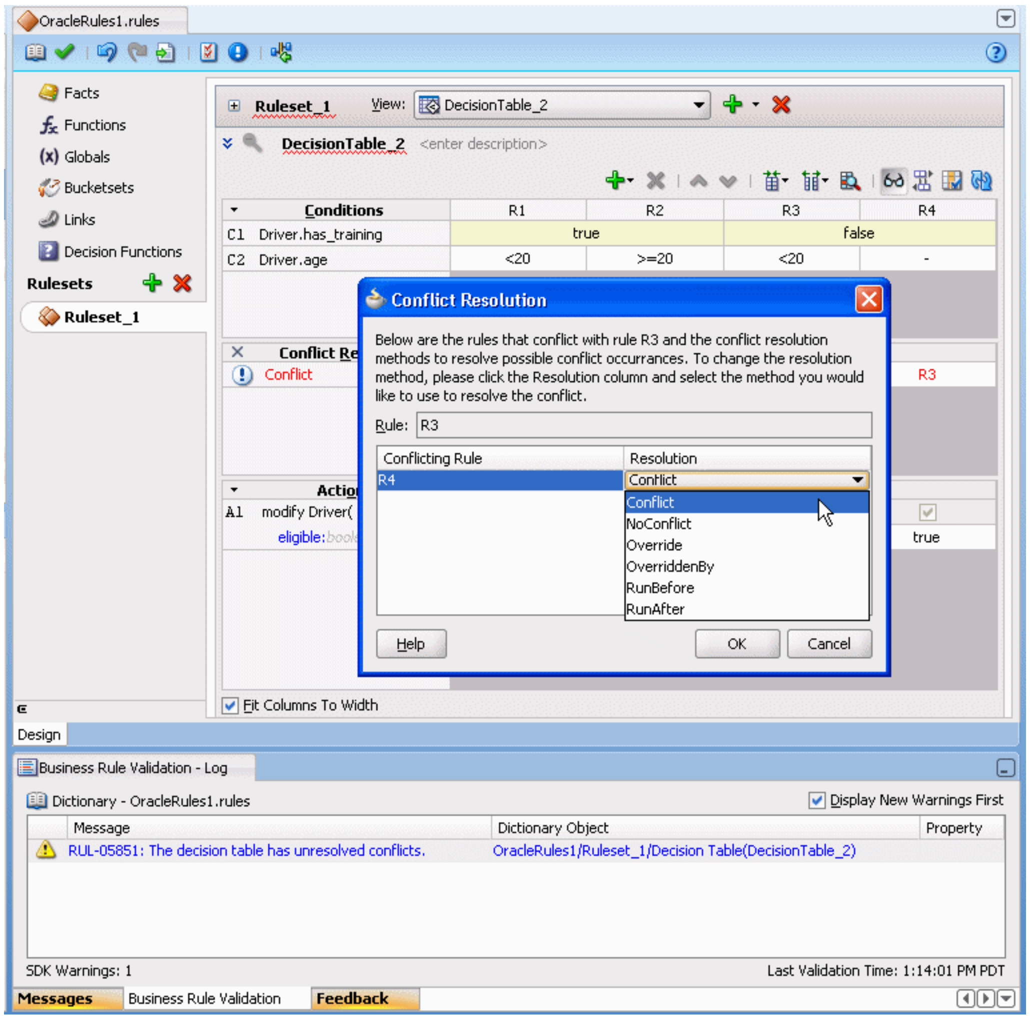
- Override (**Override** and **OverriddenBy**): You override one rule with the other. Override specifies that one rule fires. Override is a combination of prioritization and mutual exclusion. Prioritization is transitive and not symmetric. Mutual exclusion is both transitive and symmetric. If A overrides 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 5–17 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 5–17 Using the Decision Table Conflict Resolution Dialog



Conditions	R1	R2	R3	R4
C1 Driver.has_training	true		false	
C2 Driver.age	<20	>=20	<20	-

Conflict Resolution

Below are the rules that conflict with rule R3 and the conflict resolution methods to resolve possible conflict occurrances. To change the resolution method, please click the Resolution column and select the method you would like to use to resolve the conflict.

Rule: R3

Conflicting Rule	Resolution
R4	Conflict

Conflict
NoConflict
Override
OverriddenBy
RunBefore
RunAfter

Message	Dictionary Object	Property
RUL-05851: The decision table has unresolved conflicts.	OracleRules1/Ruleset_1/Decision Table(DecisionTable_2)	

SDK Warnings: 1

Last Validation Time: 1:14:01 PM PDT

You can use the Decision Table Advanced Settings **Auto Conflict Resolution** option to specify that, where possible, conflicts are automatically resolved. The automatic conflict resolution policy specifies that a special case overrides a more general case. For more information, see Section 4.5, "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 conflict resolution by selecting the **Auto Conflict Resolution** checkbox in the Decision Table.
- Use the Conflict Resolution dialog by selecting cells in the **Conflict Resolution** area with the **Show Conflicts** checkbox selected.
- Change the Decision Table to remove an overlap.
- Combine actions to remove conflicts.

5.3.2 How to Compact or Split a Decision Table

Use the **Compact Table** and **Split Table** icons to compact or split cells in a Decision Table. For more information, see Section 5.3.1.1, "Understanding Decision Table Split and Compact Operations."

To compact a decision table:

1. In Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table to compact.
2. Click the **Compact Table** icon.

To split cells in a decision table:

1. From Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table to split.
2. Click the **Split Table** icon.

5.3.3 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 Section 5.3.1.1, "Understanding Decision Table Split and Compact Operations."

To merge 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 merge a condition.
2. In the **Conditions** area, select the condition you want to merge.
3. Right-click, and from the dropdown list select **Merge Condition**.

To split 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 split a condition.
2. In the **Conditions** area, select the condition you want to split.
3. Right-click and from the dropdown list select **Split Condition**.

5.3.4 How to Merge, Split, and Specify Do Not Care for Condition Cells

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 Section 5.3.1.1, "Understanding Decision Table Split and Compact Operations."

To merge sibling cells 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 merge condition cells.
2. Select the sibling condition cells to merge.
3. Right-click, and from the dropdown list select **Merge selected cells**.

To split 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 split a condition cell.
2. Select the cell to split.
3. Right-click, and from the dropdown list select **Split selected cell**.

To specify 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 dropdown list select **Do Not Care**.

To select all buckets to specify a "Do Not Care" value for a cell in a condition:

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. Double-click, and from the dropdown list select all the available checkboxes for all possible values.

5.3.5 How to Perform Decision Table Gap Analysis

A gap is a "missing" rule in a Decision Table. A Decision Table has a gap if there is a combination of buckets, one from each condition, that is not covered by an existing rule. Rules Designer provides **Gap Analysis** to check for gaps. When you use this operation Rules Designer presents a window to fix gaps. For more information, see Section 5.3.1.3, "Understanding Decision Table Gap Analysis".

You can choose to make existence of gaps a validation warning. When you deselect **Allow Gaps** in the **Advanced Settings** area, the Decision Table reports a validation warning when a gap is found. For more information, see Section 4.5, "Using Advanced Settings with Rules and Decision Tables".

To perform decision table gap analysis:

1. From Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to perform.
2. Click the **Gap Analysis** icon.

5.3.6 How to Perform 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 bucket in common. For more information, see Section 5.3.1.4, "Understanding Decision Table Conflict Analysis"

To perform decision table conflict analysis:

1. From Rules Designer select a ruleset from the **Rulesets** navigation tab and select the Decision Table where you want to check conflicts.
2. 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.
3. In the Conflict Resolution dialog, for each conflicting rule, in the Resolution field select a resolution from the dropdown list.

5.3.7 How to Select the Auto Conflict Resolution Option

When you select the Advanced Settings option in a Decision Table, you can select that the Decision Table conflicts are automatically resolved using an Override conflict resolution (this applies only when this is possible to resolve the conflict using the automatic conflict resolution policies). The automatic conflict resolution policy is that a special case overrides a more general case. For more information, see Section 5.3.1.4, "Understanding Decision Table Conflict Analysis".

To select the auto conflict resolution option:

1. Select the rule or Decision Table where you want to use **Auto Conflict Resolution**.
2. Click the **Show Advanced Settings** icon next to the rule or Decision Table name.
3. Select **Auto Conflict Resolution**.

5.4 Creating and Running an Oracle Business Rules Decision Table Application

The Order Approval application demonstrates the integration of an 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 Bucketsets to the Data Model for Order Approval
- Associate Bucketsets 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

5.4.1 How to Obtain the Source Files for the Order Approval Application

The source code for Oracle Business Rules-specific samples is available online at

`http://www.oracle.com/technology/sample_code/products/rules`

For SOA samples online visit

`http://www.oracle.com/technology/sample_code/products/soa`

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 Example 5–1 locally to make it available to Oracle JDeveloper.

Example 5–1 shows the `order.xsd` schema file.

Example 5–1 Order.xsd Schema

```
<?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>
```

5.4.2 How to Create an Application for Order Approval

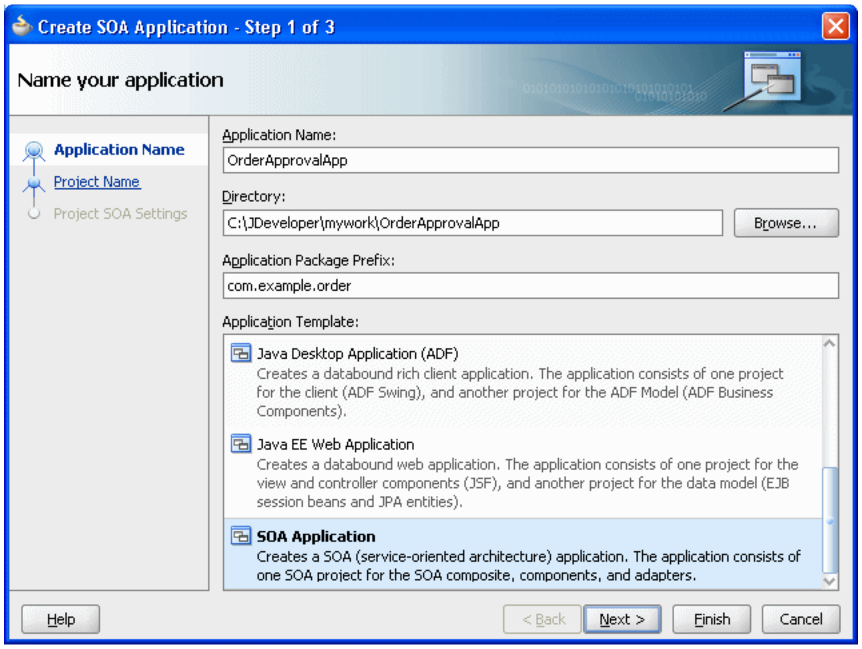
To work with Oracle Business Rules, you first create an application in Oracle JDeveloper.

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 an SOA composite with Oracle Business Rules, in the Application Template area select SOA Application for the application template. For example, see Figure 5–18.
   e. Click **Next**.

Figure 5–18 Adding the Order Approval Application



Application Name: OrderApprovalApp

Directory: C:\JDeveloper\mywork\OrderApprovalApp

Application Package Prefix: com.example.order

Application Template:

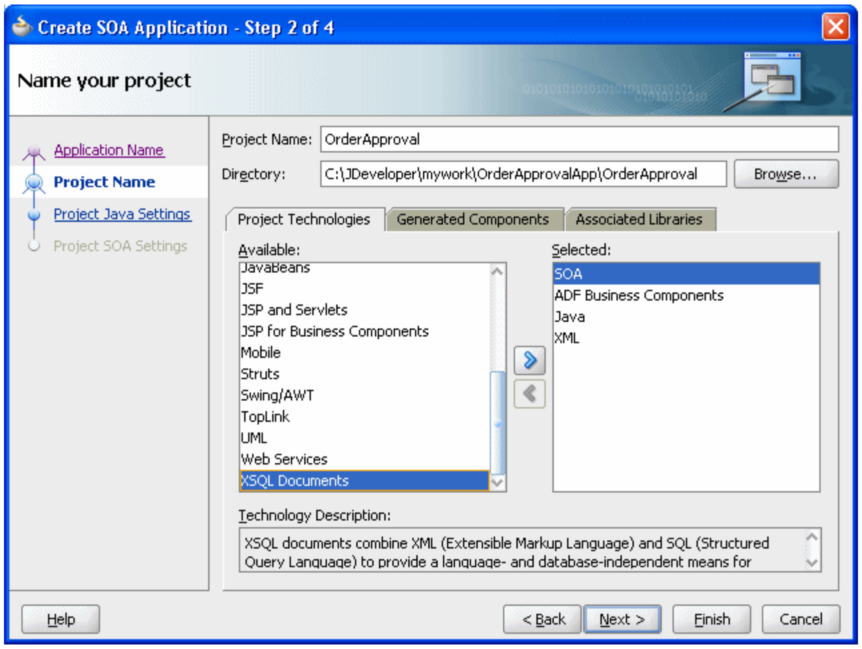
Java Desktop Application (ADF)
Creates a databound rich client application. The application consists of one project for the client (ADF Swing), and another project for the ADF Model (ADF Business Components).

Java EE Web Application
Creates a databound web application. The application consists of one project for the view and controller components (JSF), and another project for the data model (EJB session beans and JPA entities).

SOA Application
Creates a SOA (service-oriented architecture) application. The application consists of one SOA project for the SOA composite, components, and adapters.

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 5–19. If an item is missing, select it in the **Available** pane and add it to the **Selected** pane using the **Add** button.

Figure 5–19 Adding a Project to an Application


Create SOA Application - Step 2 of 4

Name your project

Project Name: OrderApproval

Directory: C:\JDeveloper\mywork\OrderApprovalApp\OrderApproval

Project Technologies

Available: JavaBeans, JSF, JSP and Servlets, JSP for Business Components, Mobile, Struts, Swing/AWT, TopLink, UML, Web Services, XSQL Documents

Selected: SOA, ADF Business Components, Java, XML

Technology Description:

XSQL documents combine XML (Extensible Markup Language) and SQL (Structured Query Language) to provide a language- and database-independent means for

4. Click **Finish**.

5.4.3 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:

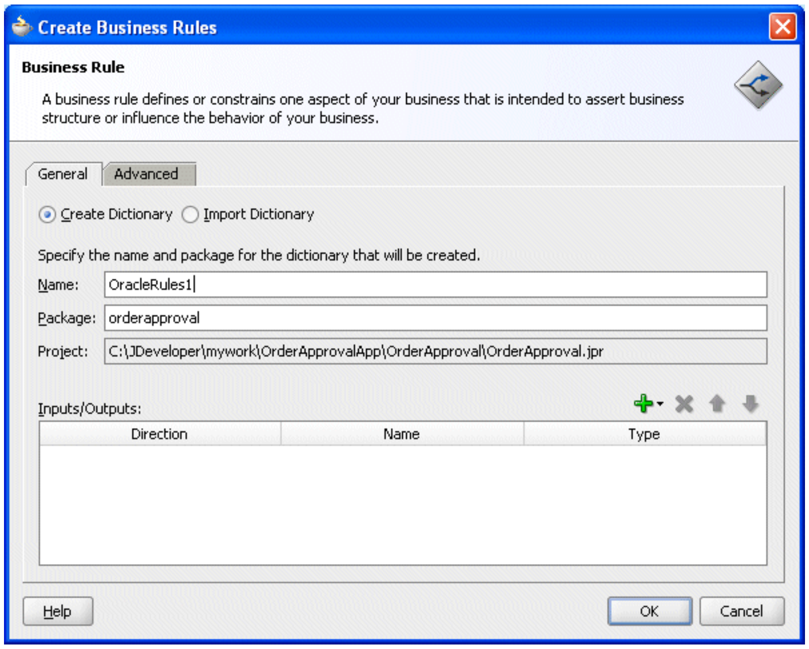
- Add a business rules service component
- Create input and output variables for the service component
- Create an Oracle Business Rules dictionary in the project

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 5–20.

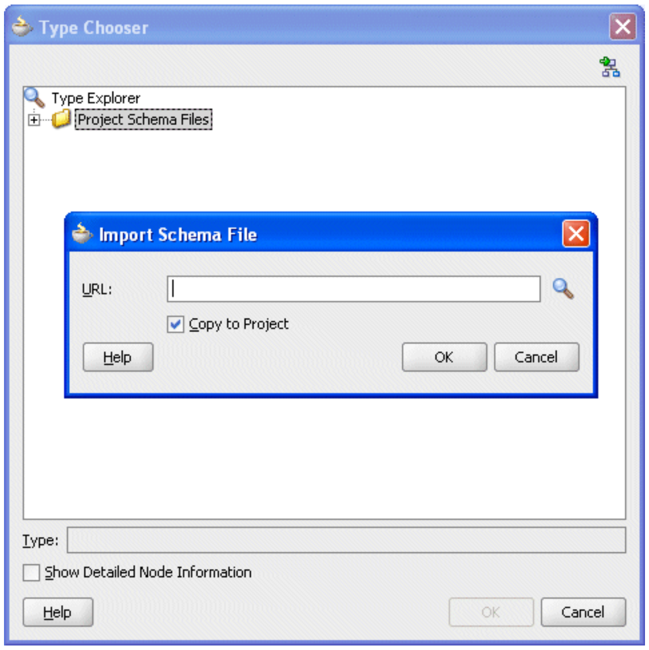
Figure 5–20 Adding a Business Rule Dictionary with the Create Business Rules Dialog


Create Business Rules
Business Rule
A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business.
General
Advanced
Create Dictionary
Import Dictionary
Specify the name and package for the dictionary that will be created.
Name: OracleRules1
Package: orderapproval
Project: C:\JDeveloper\mywork\OrderApprovalApp\OrderApproval\OrderApproval.jpr
Inputs/Outputs:
Direction
Name
Type
Help
OK
Cancel

3. To add an input, from the dropdown list next to the **Add** icon select **Input** to enter input for the business rule.

4. In the Type Chooser dialog, click the **Import Schema File**... icon. This displays the Import Schema File dialog, as shown in Figure 5–21.

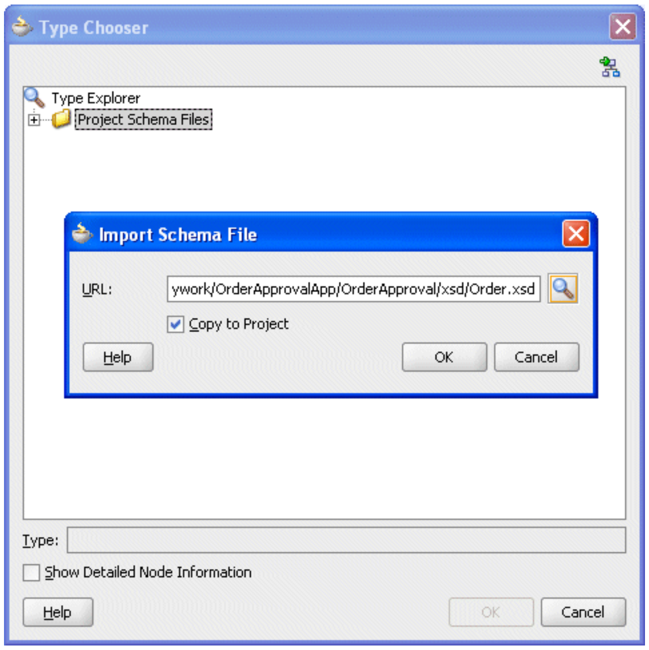
Figure 5–21 Import Schema File with Type Chooser



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 5–22.

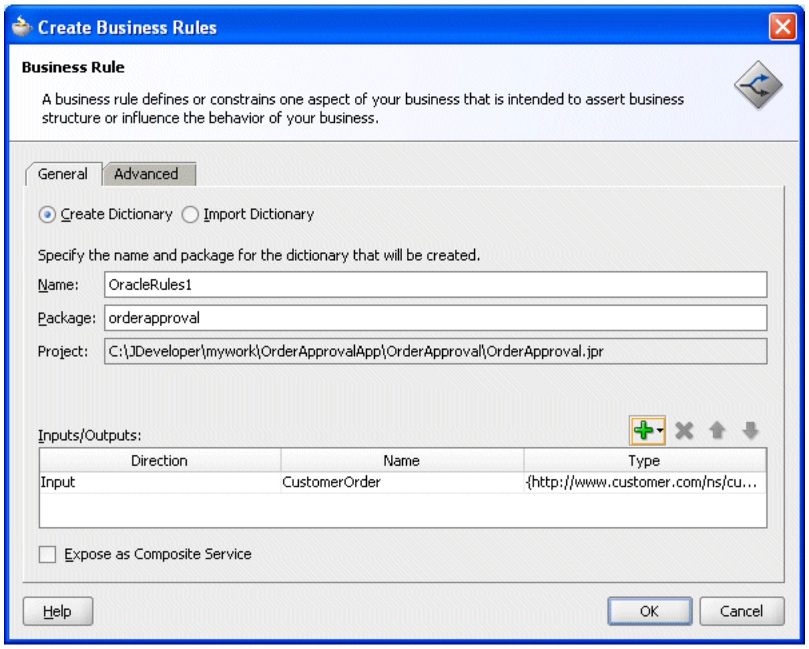
Figure 5–22 Importing the Order.xsd Schema File



8. In the Import Schema File dialog, click **OK**.
9. If the Localize Files dialog displays, click **OK** to copy the schema to the composite process directory.
10. 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.
11. On the Type Chooser window, click **OK**. This displays the Create Business Rules dialog, as shown in Figure 5–23.

Figure 5–23 Create Business Rules Dialog with CustomerOrder Input



Create Business Rules

Business Rule

A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business.

General | Advanced

(●) Create Dictionary ( ) Import Dictionary

Specify the name and package for the dictionary that will be created.

Name: OracleRules1

Package: orderapproval

Project: C:\JDeveloper\mywork\OrderApprovalApp\OrderApproval\OrderApproval.jpr

Inputs/Outputs:

Direction	Name	Type
Input	CustomerOrder	{http://www.customer.com/ns/cu...

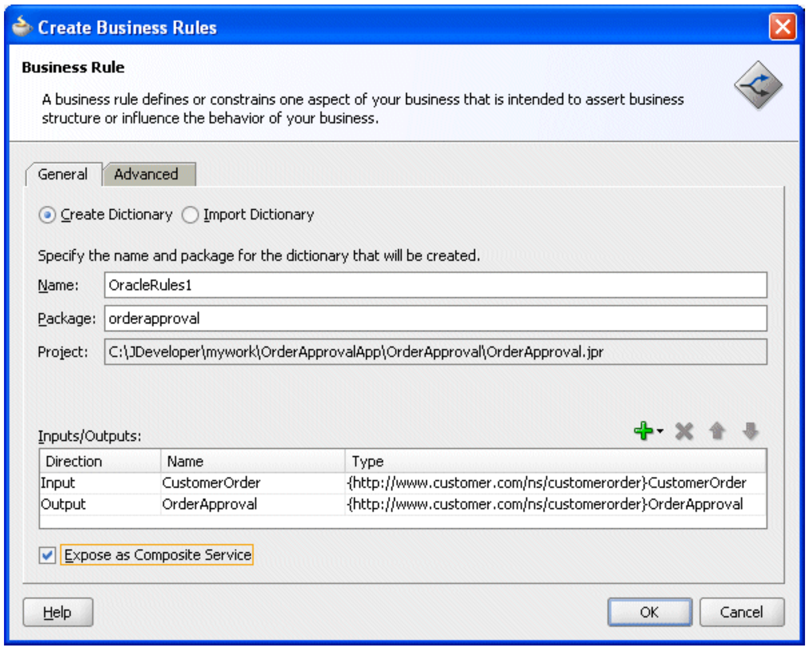
[ ] Expose as Composite Service

Help | OK | Cancel

12. In a similar manner, add the output fact type `OrderApproval` from the imported `order.xsd`.

13. In the Create Business Rules dialog, select **Expose as Composite Service**, as shown in Figure 5–24.

Figure 5–24 Create Business Rules Dialog with Input and OrderApproval Output



Create Business Rules

Business Rule

A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business.

General | Advanced

(●) Create Dictionary ( ) Import Dictionary

Specify the name and package for the dictionary that will be created.

Name: OracleRules1

Package: orderapproval

Project: C:\JDeveloper\mywork\OrderApprovalApp\OrderApproval\OrderApproval.jpr

Inputs/Outputs:

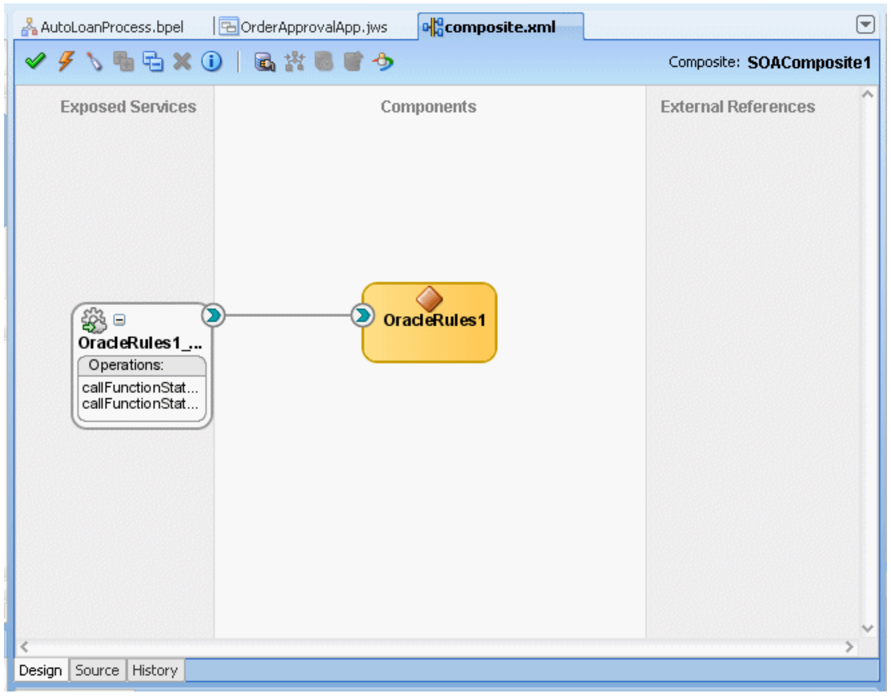
Direction	Name	Type
Input	CustomerOrder	{http://www.customer.com/ns/customerorder}CustomerOrder
Output	OrderApproval	{http://www.customer.com/ns/customerorder}OrderApproval

[✓] Expose as Composite Service

Help | OK | Cancel

14. Click **OK**. This creates the Business Rule component and Oracle JDeveloper shows the Business Rule in the canvas workspace, as shown in Figure 5–25.

Figure 5–25 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.

5.4.4 How to View Data Model Elements for Order Approval

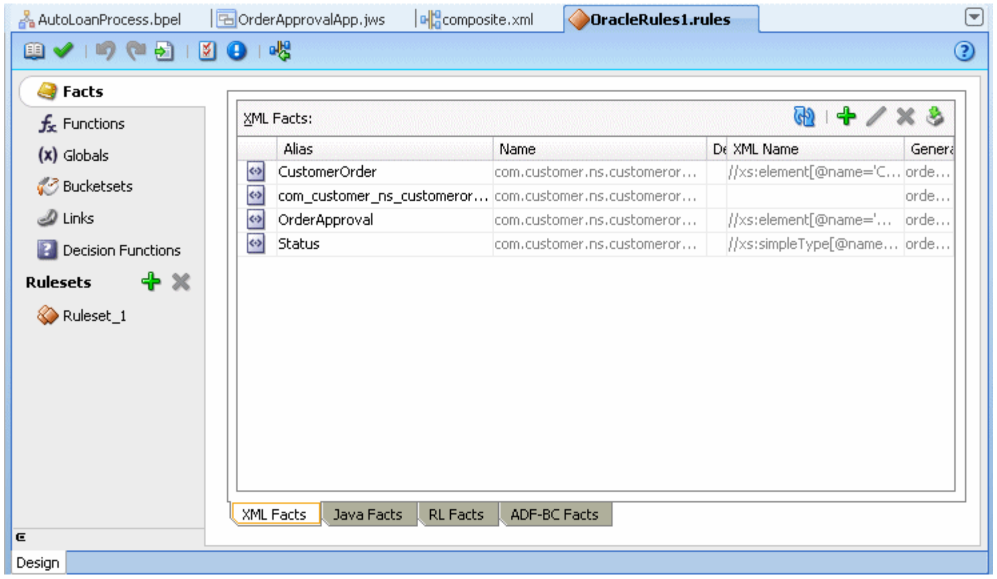
Before adding rules you need to create the Oracle Business Rules data model. The data model contains the business data definitions (types) and definitions for facts that you use to create rules. For example, for this sample the data model includes the XML schema elements from `order.xsd` that you specify 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 5–26.

Figure 5–26 Opening a Business Rules Dictionary with Rules Designer



Alias	Name	De	XML Name	Genera
CustomerOrder	com.customer.ns.customeror...		//xs:element[@name='C...	orde...
com_customer_ns_customeror...	com.customer.ns.customeror...			orde...
OrderApproval	com.customer.ns.customeror...		//xs:element[@name='...	orde...
Status	com.customer.ns.customeror...		//xs:simpleType[@name...	orde...

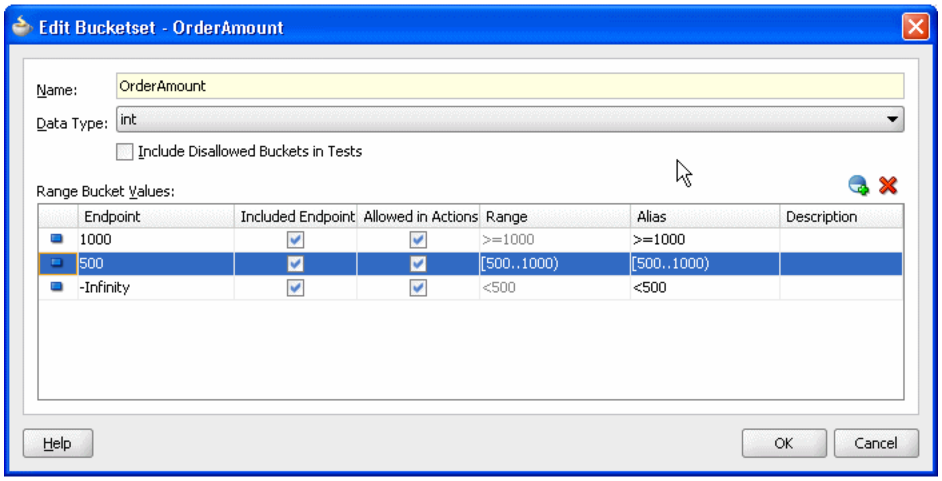
5.4.5 How to Add Bucketsets to the Data Model for Order Approval

To use a Decision Table you need to define bucketsets that specify how to draw values for each cell for the conditions that constitute the Decision Table. For this example the bucketsets are defined with a list of ranges that you define in Rules Designer.

To add OrderAmount bucketset to the data model:

1. In Rules Designer, select the **Bucketsets** navigation tab.
2. From the dropdown next to the **Create BucketSet...** icon, select **List of Ranges**.
3. In the **Name** field, enter `OrderAmount` (In Rules Designer be sure to press **Enter** to accept the name).
4. Double-click the **OrderAmount** bucketset icon to display the Edit Bucketset dialog.
5. Click **Add Bucket** to add a bucket.
6. Click **Add Bucket** again to add another bucket.
7. In the **Range Bucket Values** area, in the top **Endpoint** field enter 1000 for the endpoint value.
8. In the **Range Bucket Values** area, for the middle bucket in the **Endpoint** field enter 500 for the endpoint value.
9. In the **Included Endpoint** field for each bucket ensure the checkbox is selected, as shown in Figure 5–27.

Figure 5–27 Adding the OrderAmount Bucketset



Edit Bucketset - OrderAmount

Name: OrderAmount

Data Type: int

Include Disallowed Buckets in Tests

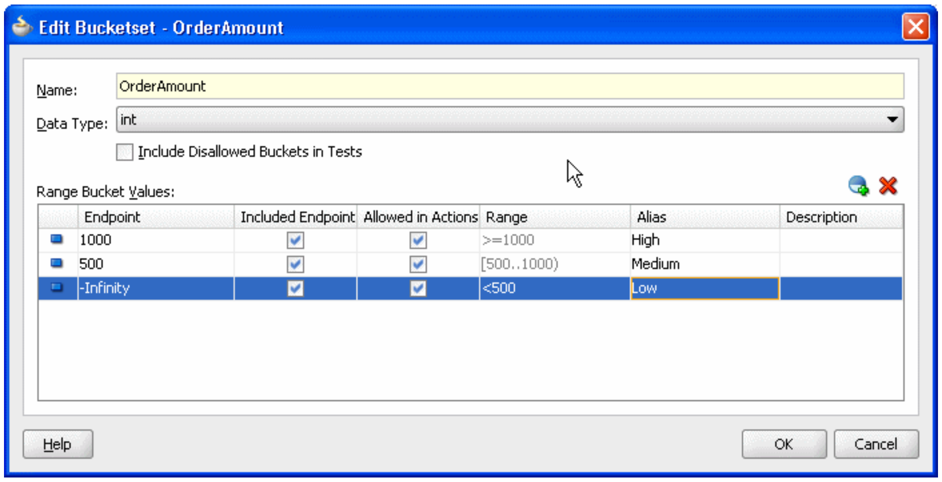
Range Bucket Values:

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
1000	✓	✓	>=1000	>=1000	
500	✓	✓	[500..1000)	[500..1000)	
-Infinity	✓	✓	<500	<500	

Help OK Cancel

10. Modify the **Alias** field for each value to **High, Medium, and Low**, as shown in Figure 5–28.

Figure 5–28 Adding the OrderAmount Bucketset with Low Medium and High Aliases



Edit Bucketset - OrderAmount

Name: OrderAmount

Data Type: int

Include Disallowed Buckets in Tests

Range Bucket Values:

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
1000	✓	✓	>=1000	High	
500	✓	✓	[500..1000)	Medium	
-Infinity	✓	✓	<500	Low	

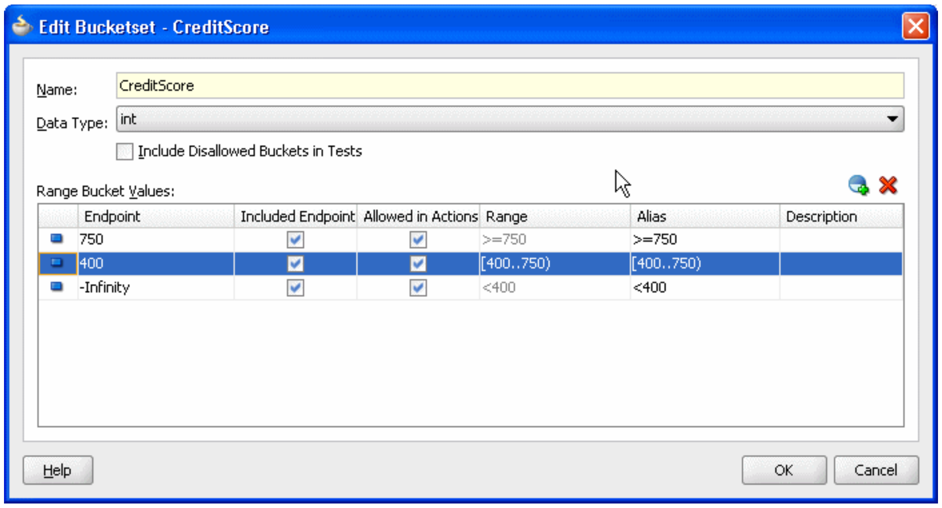
Help OK Cancel

11. Click **OK**.

To add CreditScore bucketset to data model:

1. In Rules Designer select the **Bucketsets** navigation tab.
2. From the dropdown next to the **Create BucketSet...** icon, select **List of Ranges**.
3. In the **Name** field, enter `CreditScore`.
4. Double-click the **CreditScore** bucketset icon to display the Edit Bucketset dialog.
5. Click **Add Bucket** to add a bucket.
6. Click **Add Bucket** again to add another bucket.
7. In the top bucket, in the **Endpoint** field enter `750`.
8. For the middle bucket, in the **Endpoint** field enter `400`.
9. In the **Included Endpoint** field for each bucket, ensure the checkbox is selected as shown in Figure 5–29.

Figure 5–29 Adding the CreditScore Bucketset


Edit Bucketset - CreditScore
Name:
Data Type:
Include Disallowed Buckets in Tests
Range Bucket Values:
Help
OK
Cancel

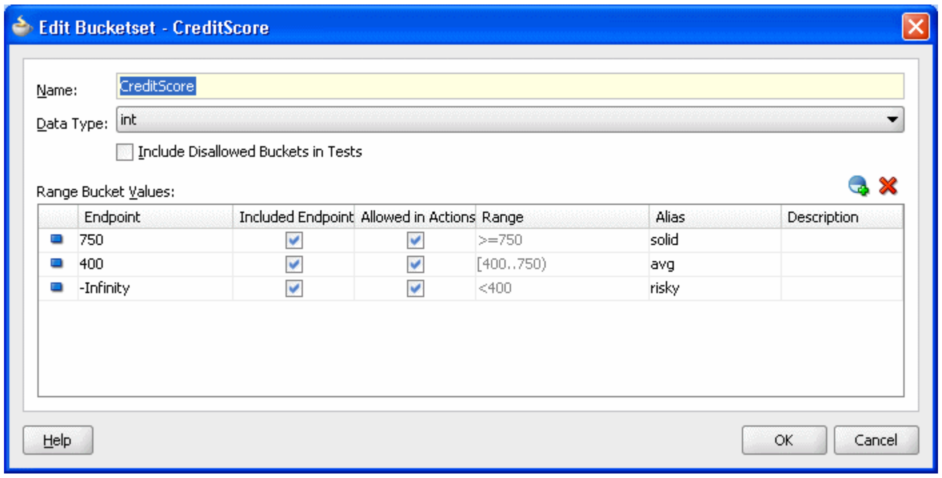
Name: CreditScore

Data Type: int

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
750	✓	✓	>=750	>=750	
400	✓	✓	[400..750)	[400..750)	
-Infinity	✓	✓	<400	<400	

10. Modify the **Alias** field for each endpoint value to **solid** for 750, **avg** for 400, and **risky** for -Infinity as shown in Figure 5–30.
11. Click **OK**.

Figure 5–30 Adding the CreditScore Bucketset with Risky Avg and Solid Aliases


Edit Bucketset - CreditScore
Name:
Data Type:
Include Disallowed Buckets in Tests
Range Bucket Values:
Help
OK
Cancel

Name: CreditScore

Data Type: int

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
750	✓	✓	>=750	solid	
400	✓	✓	[400..750)	avg	
-Infinity	✓	✓	<400	risky	

5.4.6 How to Associate Bucketsets with Order and CreditScore Properties

To prepare for creating Decision Tables you can associate a bucketset with fact properties in the data model. In this way condition cells in a Decision Table **Conditions** area can use the bucketset when you create a Decision Table.

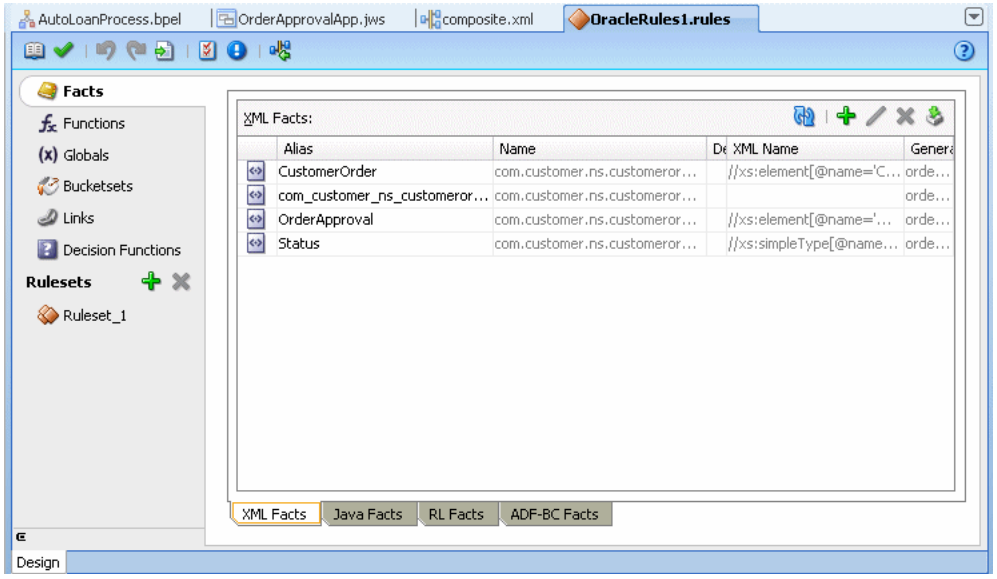
Note that the `OrderApproval.status` property is associated with the `Status` bucketset when the `OrderApproval` fact type is imported from the XML schema. In the schema, `Status` is a restricted `String` type and is therefore represented as an enum bucketset. Rules Designer creates the status bucketset. For more information, see Section 3.2.4, "What You Need to Know About XML Facts".

To associate bucketsets 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 5–31.

Figure 5–31 Opening a Business Rules Dictionary with Rules Designer

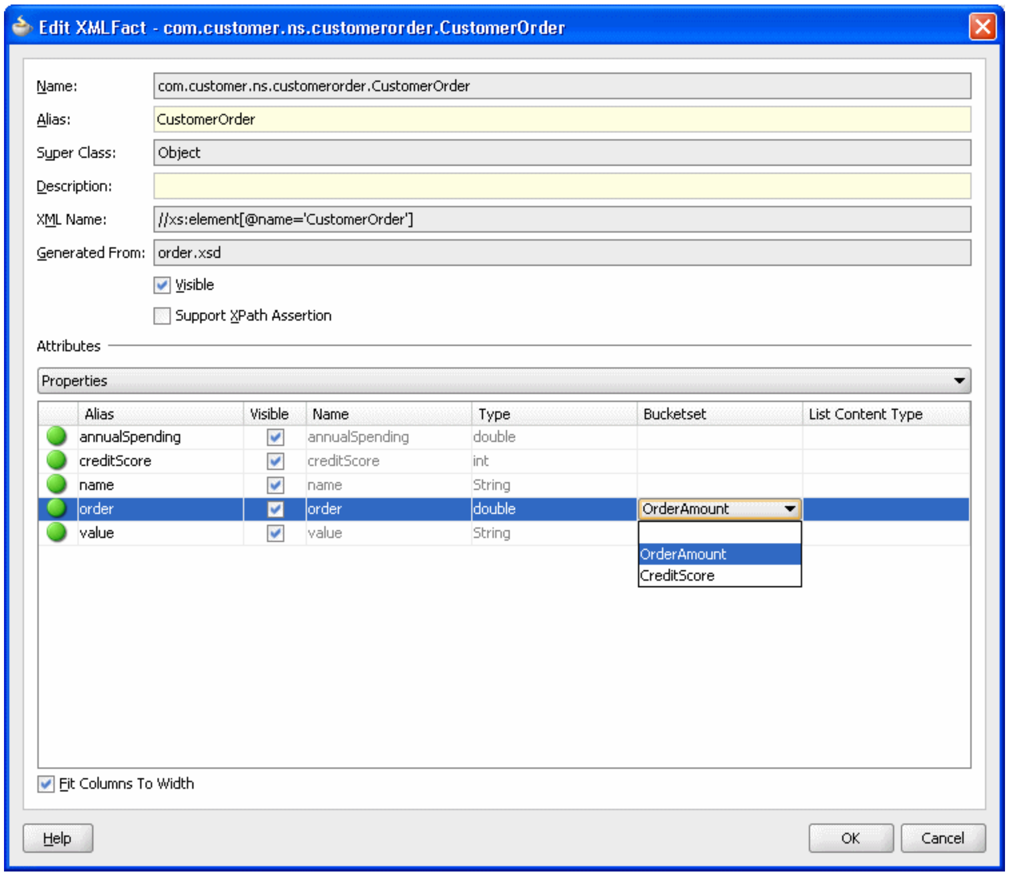


Alias	Name	De	XML Name	Genera
CustomerOrder	com.customer.ns.customeror…		//xs:element[@name='C…	orde…
com_customer_ns_customeror…	com.customer.ns.customeror…			orde…
OrderApproval	com.customer.ns.customeror…		//xs:element[@name='…	orde…
Status	com.customer.ns.customeror…		//xs:simpleType[@name…	orde…

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 **Bucketset** column select the cell for the appropriate property and from the dropdown list select the bucketset you want to use. For example, for the property **order** select the **OrderAmount** bucketset, as shown in Figure 5–32.

Figure 5–32 Associating the OrderAmount Bucketset with CustomerOrder.order



Name:	com.customer.ns.customerorder.CustomerOrder
Alias:	CustomerOrder
Super Class:	Object
Description:	
XML Name:	//xs:element[@name='CustomerOrder']
Generated From:	order.xsd

☑ Visible

☐ Support XPath Assertion

Attributes

Properties

Alias	Visible	Name	Type	Bucketset	List Content Type
annualSpending	✓	annualSpending	double		
creditScore	✓	creditScore	int		
name	✓	name	String		
order	✓	order	double	OrderAmount	
value	✓	value	String		

5. In a similar manner, for the property **creditScore** select the **CreditScore** bucketset.
6. Click **OK**.

5.4.7 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** icon 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 Section 4.4.2, "Understanding Rule Validation".

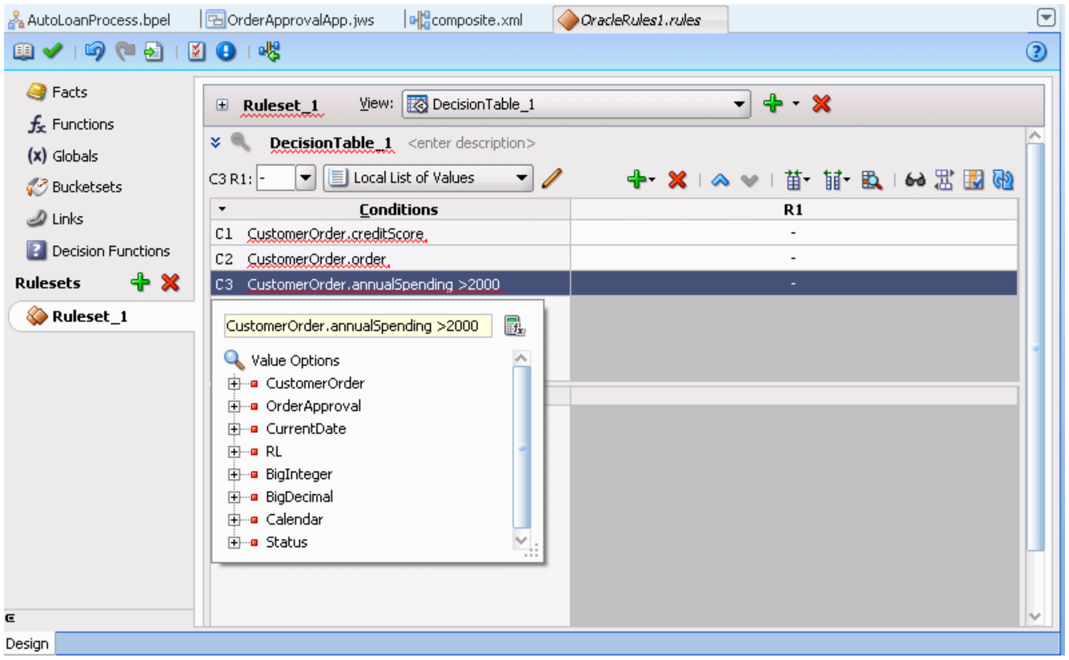
To use a Decision Table for rules in this sample application you work with facts representing a customer spending level and a customer credit risk for a particular customer and a particular order. Then, you use a Decision Table to create rules based on customer spending, the order amount, and the credit 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** icon and from the dropdown list and select **Create Decision Table**.

3. In the Decision Table, click the **Add** icon and from the dropdown 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** icon and from the dropdown list select **Condition**.
6. In the Decision Table, in the **Conditions** area double-click the **<edit condition>**. Then, in the dropdown navigator expand **CustomerOrder** and select **order**. This enters the expression `CustomerOrder.order`.
7. Again, in the Decision Table, click the **Add** icon and from the dropdown list select **Condition**.
8. In the Decision Table, double-click **<edit condition>**.
9. In the dropdown navigator expand **CustomerOrder** and select **annualSpending**. In the text entry area, add `>2000` as shown in Figure 5–33.

Figure 5–33 Adding the Annual Spending Entry to a Decision Table

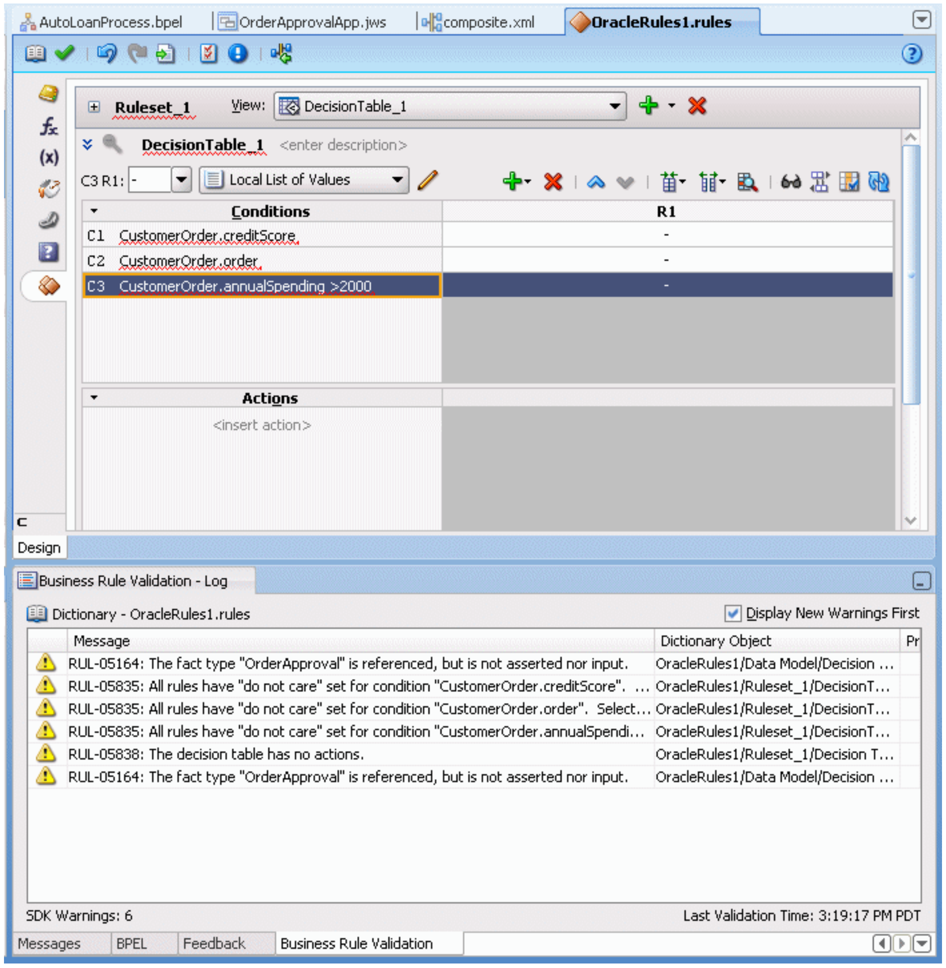


	Conditions	R1
C1	CustomerOrder.creditScore	-
C2	CustomerOrder.order	-
C3	CustomerOrder.annualSpending >2000	-

CustomerOrder.annualSpending >2000

10. Type **Enter** to accept the value, as shown in Figure 5–34. If you view the rules validation log you should see the warning messages as shown in Figure 5–34. You remove these warning messages as you modify the Decision Table in later steps.

Figure 5–34 Adding Conditions to the CustomerOrder Decision Table



Conditions	R1
C1 CustomerOrder.creditScore	-
C2 CustomerOrder.order	-
C3 CustomerOrder.annualSpending >2000	-

Actions

<insert action>

Dictionary - OracleRules1.rules

Display New Warnings First

Message	Dictionary Object	Pr
RUL-05164: The fact type "OrderApproval" is referenced, but is not asserted nor input.	OracleRules1/Data Model/Decision ...	
RUL-05835: All rules have "do not care" set for condition "CustomerOrder.creditScore". ...	OracleRules1/Ruleset_1/DecisionT...	
RUL-05835: All rules have "do not care" set for condition "CustomerOrder.order". Select...	OracleRules1/Ruleset_1/DecisionT...	
RUL-05835: All rules have "do not care" set for condition "CustomerOrder.annualSpendi...	OracleRules1/Ruleset_1/DecisionT...	
RUL-05838: The decision table has no actions.	OracleRules1/Ruleset_1/Decision T...	
RUL-05164: The fact type "OrderApproval" is referenced, but is not asserted nor input.	OracleRules1/Data Model/Decision ...	

SDK Warnings: 6

Last Validation Time: 3:19:17 PM PDT

To create an action in a decision table:

1. In the Decision Table click the **Add** icon and from the dropdown list select **Action** > **Assert New**.

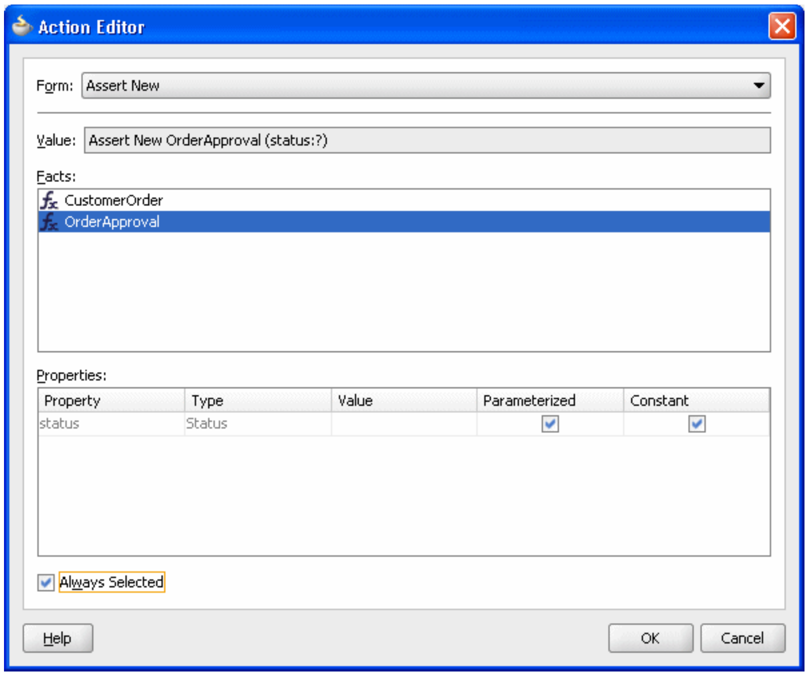
2. In the **Actions** area, double-click **assert new(**. This displays the Action Editor dialog.

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** checkbox and the **Constant** checkbox. This specifies that each rule independently sets the status.

5. In the Action Editor dialog, select the **Always Selected** checkbox as shown in Figure 5–35.

Figure 5–35 Adding an Action to a Decision Table with the Action Editor Dialog



Form: Assert New

Value: Assert New OrderApproval (status:?)

Facts:
- CustomerOrder
- OrderApproval

Properties:

Property	Type	Value	Parameterized	Constant
status	Status		✓	✓

✓ Always Selected

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.

5.4.7.1 Split the Cells in the Decision Table and Add Actions

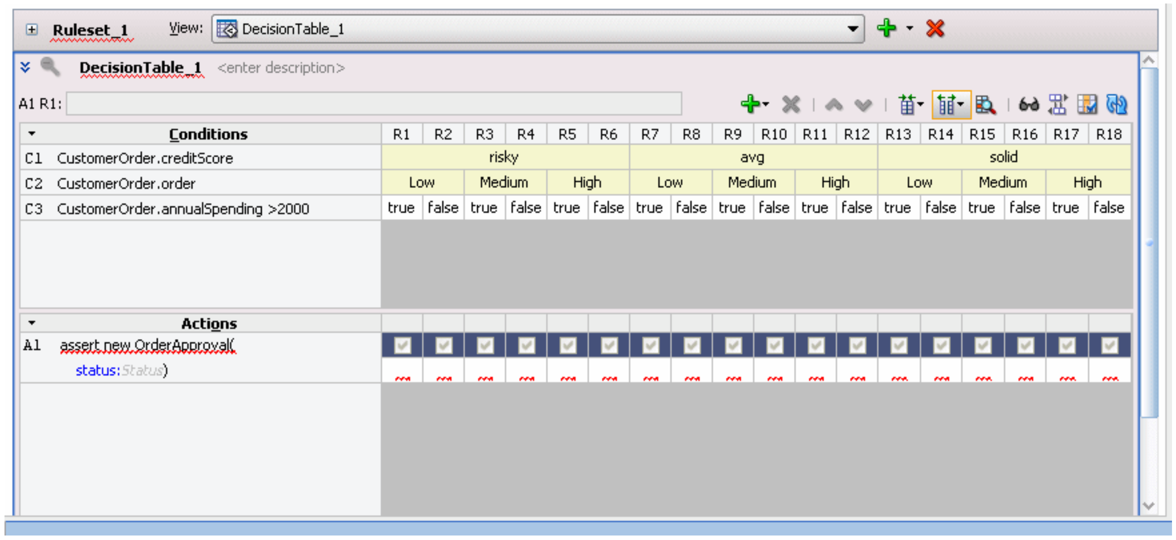
You can use the Decision Table split operation to create rules for the bucketsets associated with the condition rows in the Decision Table. This creates one rule for every combination of condition buckets. There are three order amount buckets, three credit score buckets, and two boolean buckets for the 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** icon and from the dropdown 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 5–36.

These steps produce validation warnings for action cells with missing expressions. You fix these in later steps.

Figure 5–36 Splitting a Decision Table Using Split Table Operation



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

A1 R1:

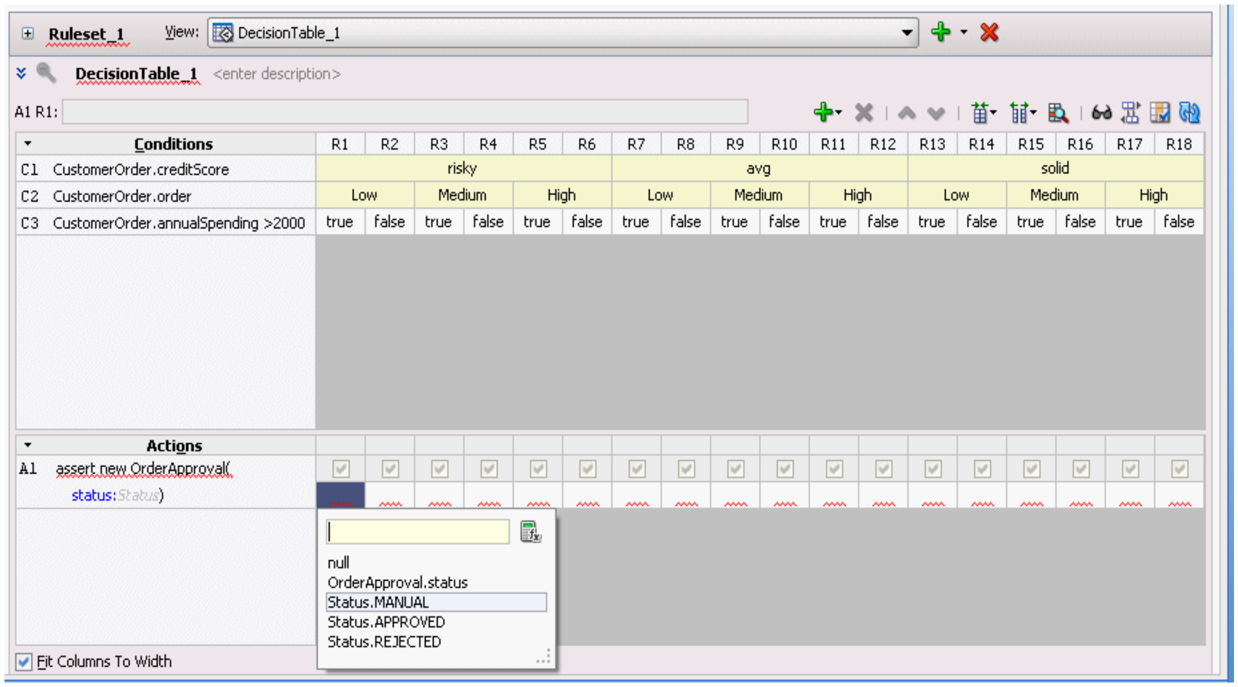
Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18
C1 CustomerOrder.creditScore	risky						avg						solid					
C2 CustomerOrder.order	Low		Medium		High		Low		Medium		High		Low		Medium		High	
C3 CustomerOrder.annualSpending >2000	true	false	true	false	true	false	true	false	true	false	true	false	true	false	true	false	true	false
Actions																		
A1 assert new OrderApproval(	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
status: Status)																		

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.

1. In the **Actions** area, double-click the action cell for the rule you want to work with, as shown in Figure 5–37.

Figure 5–37 Adding Action Cell Values to a Decision Table



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

A1 R1:

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18
C1 CustomerOrder.creditScore	risky						avg						solid					
C2 CustomerOrder.order	Low		Medium		High		Low		Medium		High		Low		Medium		High	
C3 CustomerOrder.annualSpending >2000	true	false	true	false	true	false	true	false	true	false	true	false	true	false	true	false	true	false
Actions																		
A1 assert new OrderApproval(	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
status: Status)																		

null
OrderApproval.status
Status.MANUAL
Status.APPROVED
Status.REJECTED

✓ Fit Columns To Width

2. In the dropdown 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`

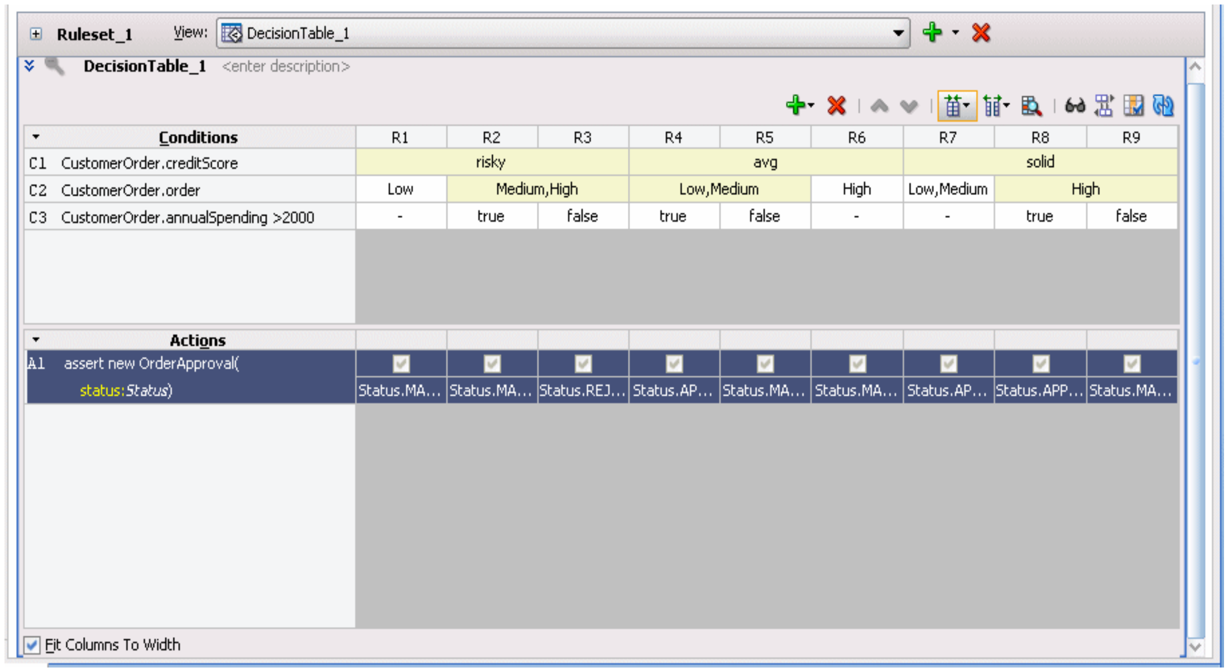
5.4.7.2 Compact the Decision Table

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** icon.
3. Click the **Compact Table** icon and from the dropdown 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 5–38.

Figure 5–38 Compacting a Decision Table Using Compact Table



Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9
C1 CustomerOrder.creditScore	risky			avg			solid		
C2 CustomerOrder.order	Low	Medium,High		Low,Medium		High	Low,Medium	High	
C3 CustomerOrder.annualSpending >2000	-	true	false	true	false	-	-	true	false

Actions	R1	R2	R3	R4	R5	R6	R7	R8	R9
A1 assert new OrderApproval(	✓	✓	✓	✓	✓	✓	✓	✓	✓
status:Status)	Status.MA...	Status.MA...	Status.REJ...	Status.AP...	Status.MA...	Status.MA...	Status.AP...	Status.APP...	Status.MA...

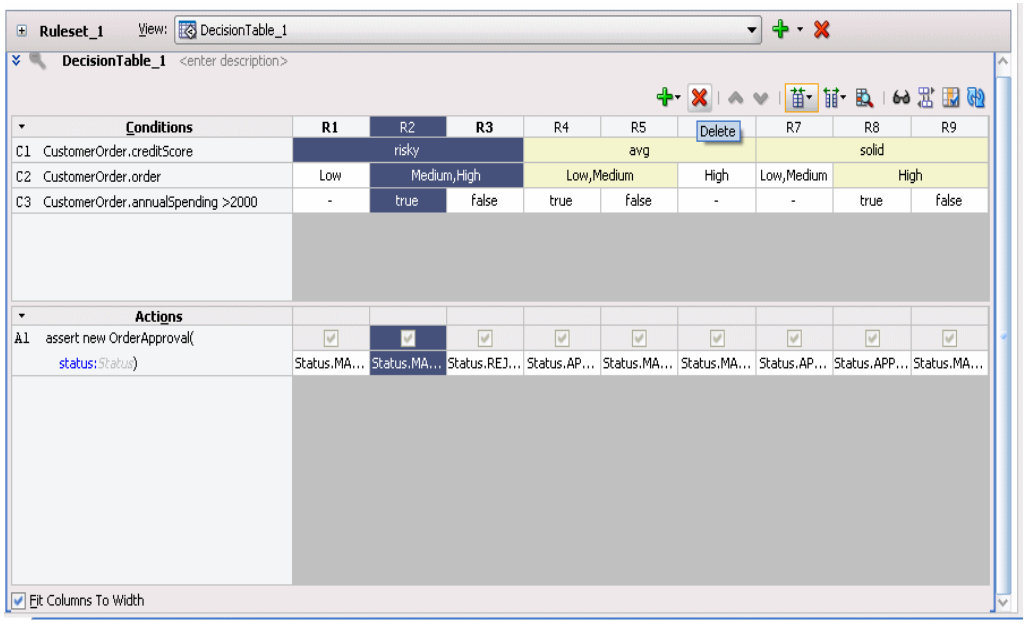
5.4.7.3 Replace Several Specific Rules with One General Rule

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 5–39.
3. Click **Delete** to remove a rule in the Decision Table. Be careful to click the delete icon in the Decision Table area to delete a rule in the decision table (there is also a delete icon shown in the **Ruleset** area that deletes the complete Decision Table).

Figure 5–39 Deleting Rules from a Decision Table



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

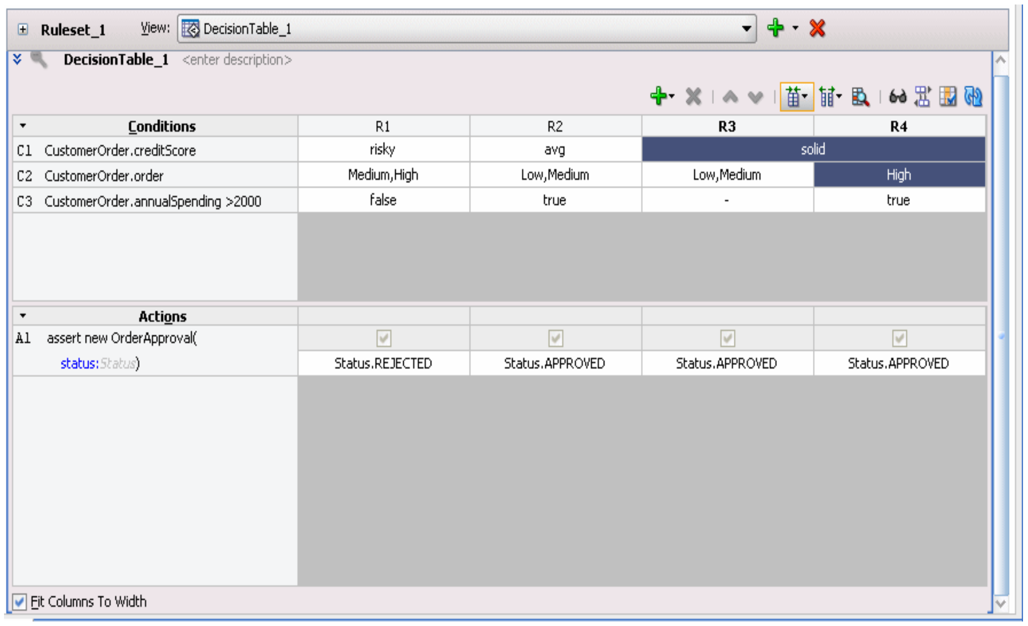
Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9
C1 CustomerOrder.creditScore	risky			avg			solid		
C2 CustomerOrder.order	Low	Medium,High		Low,Medium		High	Low,Medium	High	
C3 CustomerOrder.annualSpending >2000	-	true	false	true	false	-	-	true	false

Actions									
A1 assert new OrderApproval(	✓	✓	✓	✓	✓	✓	✓	✓	✓
status: Status)	Status.MA...	Status.MA...	Status.REJ...	Status.AP...	Status.MA...	Status.MA...	Status.AP...	Status.APP...	Status.MA...

Fit Columns To Width

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 5–40.

Figure 5–40 Decision Table After Manual Actions Removed



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

Conditions	R1	R2	R3	R4
C1 CustomerOrder.creditScore	risky	avg	solid	
C2 CustomerOrder.order	Medium,High	Low,Medium	Low,Medium	High
C3 CustomerOrder.annualSpending >2000	false	true	-	true

Actions				
A1 assert new OrderApproval(	✓	✓	✓	✓
status: Status)	Status.REJECTED	Status.APPROVED	Status.APPROVED	Status.APPROVED

Fit Columns To Width

5.4.7.4 Add a General Rule

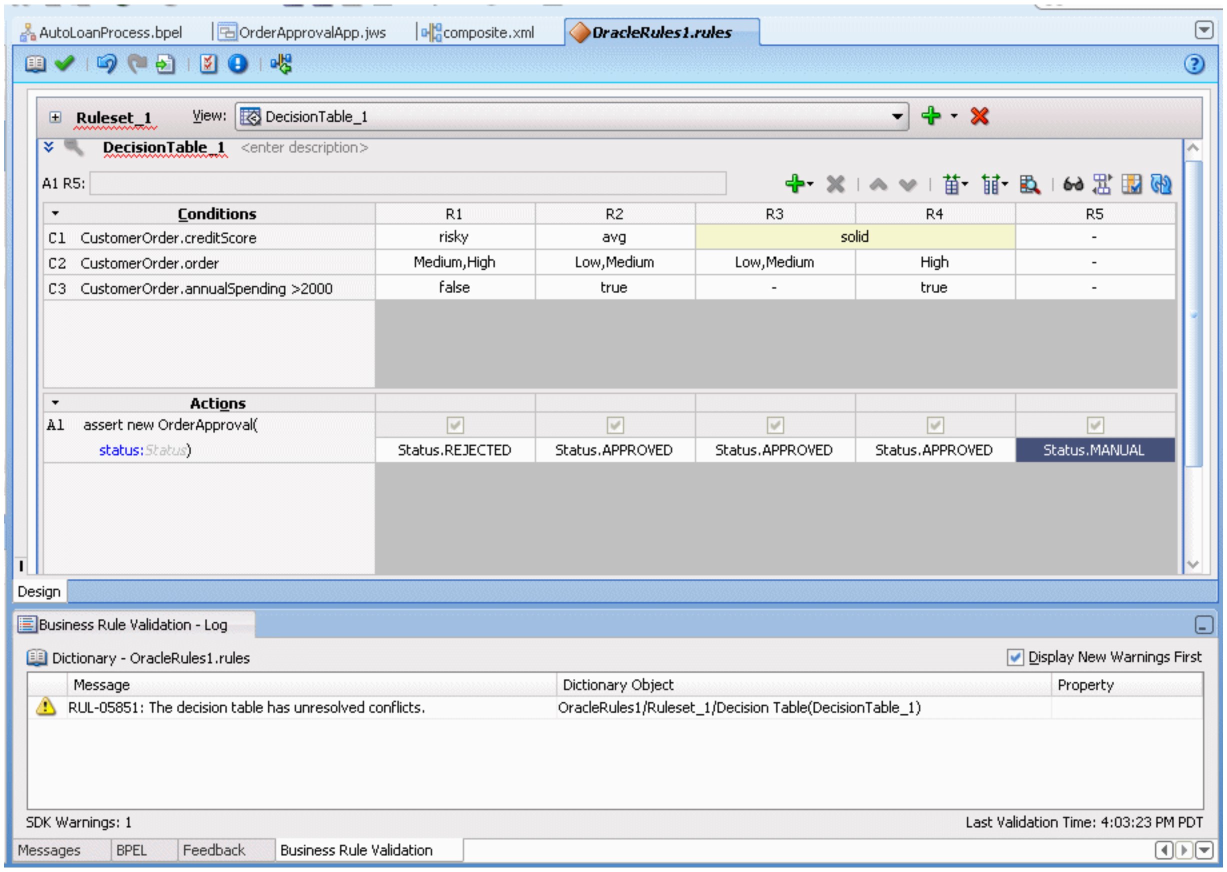
Now you can add a single rule to handle the manual case. After adding this rule you enable Auto Conflict Resolution.

To add a general rule:

1. In the Decision Table, click the **Add** icon and from the dropdown 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 5–41. Notice that the **Business Rule Validation** log includes the warning `RUL-05851` for unresolved conflicts.

Figure 5–41 Decision Table with Conflicting Rules



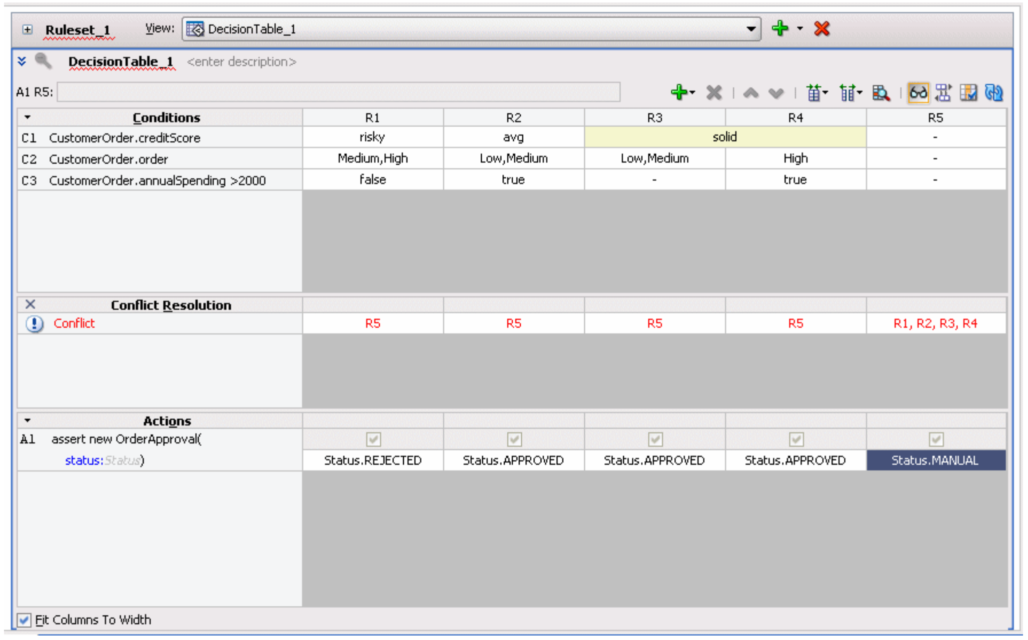
Conditions	R1	R2	R3	R4	R5
C1 CustomerOrder.creditScore	risky	avg	solid		-
C2 CustomerOrder.order	Medium,High	Low,Medium	Low,Medium	High	-
C3 CustomerOrder.annualSpending >2000	false	true	-	true	-
Actions					
A1 assert new OrderApproval(	✓	✓	✓	✓	✓
status: Status)	Status.REJECTED	Status.APPROVED	Status.APPROVED	Status.APPROVED	Status.MANUAL

	Message	Dictionary Object	Property
⚠	RUL-05851: The decision table has unresolved conflicts.	OracleRules1/Ruleset_1/Decision Table(DecisionTable_1)	

SDK Warnings: 1
Last Validation Time: 4:03:23 PM PDT

4. Show the conflicting rules by clicking the **Toggle Display of Conflict Resolution** icon, as shown in Figure 5–42.

Figure 5–42 Adding a Rule to Handle Status Manual



Ruleset_1 View: DecisionTable_1

DecisionTable_1 <enter description>

A1 R5:

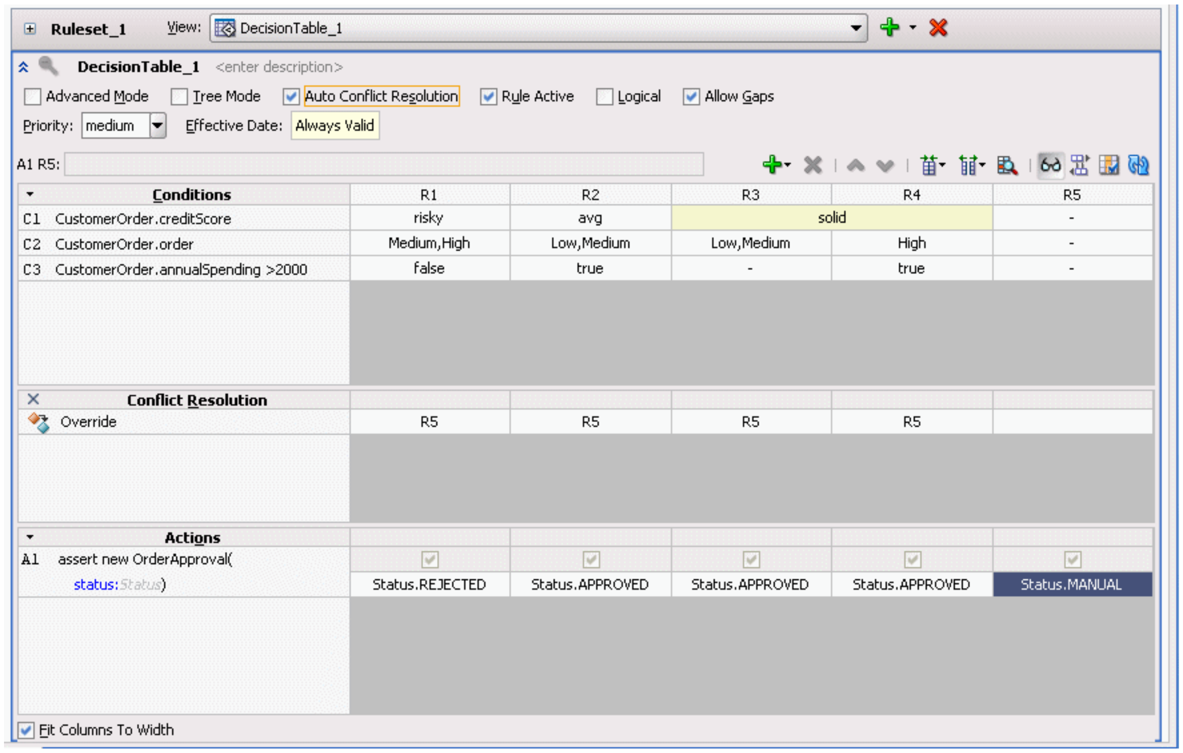
Conditions	R1	R2	R3	R4	R5
C1 CustomerOrder.creditScore	risky	avg	solid		-
C2 CustomerOrder.order	Medium,High	Low,Medium	Low,Medium	High	-
C3 CustomerOrder.annualSpending >2000	false	true	-	true	-
Conflict Resolution					
Conflict	R5	R5	R5	R5	R1, R2, R3, R4
Actions					
A1 assert new OrderApproval(	✓	✓	✓	✓	✓
status: Status)	Status.REJECTED	Status.APPROVED	Status.APPROVED	Status.APPROVED	Status.MANUAL

Fit Columns To Width

To enable auto conflict resolution:

1. In the Decision Table click **Show Advanced Settings** (the icon next to the Decision Table name).
2. Select **Auto Conflict Resolution**. After adding the manual case rule and selecting **Auto Conflict Resolution**, notice that the conflicts are resolved and special cases override the general case, as shown in Figure 5–43.

Figure 5–43 Adding a Rule to Handle Status Manual with Auto Conflict Resolution Set



Advanced Mode ☐ Tree Mode ☐ Auto Conflict Resolution ✓ Rule Active ✓ Logical ☐ Allow Gaps ✓

Priority: medium Effective Date: Always Valid

A1 R5:

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 ✓

5.4.8 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 fix any associated problems.

To validate the dictionary:

1. In the Business Rule Validation Log, check for validation warnings.
2. If there are validation warnings, perform appropriate actions to correct the problems.

5.4.9 How to Deploy the Order Approval Application

Business rules created in an SOA application are deployed as part of the SOA composite when you create a deployment profile in Oracle JDeveloper. You deploy an 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** > **SOAComposite1** > **to** > ***WLS Server Name***.

   Then the SOA Deployment Configuration dialog displays.
3. Click **OK**.
4. In the Authorization Request dialog, enter your authorization.

5. Click **OK**.

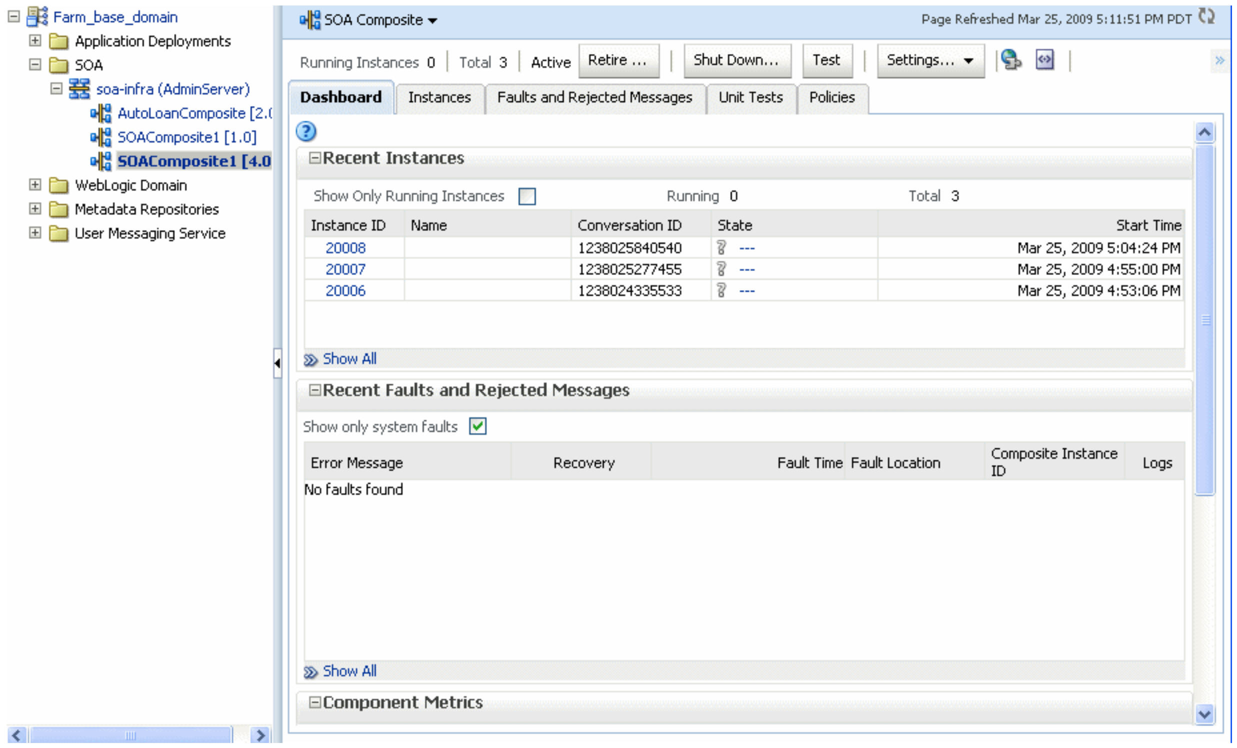
5.4.10 How to Test the Order Approval Application

After deploying the application you can test the Decision Table in the SOA composite application with Oracle Enterprise Manager Fusion Middleware Control Console.

To test the application:

1. Open the composite application in Fusion Middleware Control Console, as shown in Figure 5–44.

Figure 5–44 Testing the Order Approval Application



Page Refreshed Mar 25, 2009 5:11:51 PM PDT

Running Instances 0 | Total 3 | Active

Recent Instances

Show Only Running Instances ☐ Running 0 Total 3

Instance ID	Name	Conversation ID	State	Start Time
20008		1238025840540	---	Mar 25, 2009 5:04:24 PM
20007		1238025277455	---	Mar 25, 2009 4:55:00 PM
20006		1238024335533	---	Mar 25, 2009 4:53:06 PM

Recent Faults and Rejected Messages

Show only system faults ☑

Error Message	Recovery	Fault Time	Fault Location	Composite Instance ID	Logs
No faults found					

2. Click **Test**.

3. In the **Input Arguments** area, select **XML View**. Replace the XML with the contents of example Example 5–2.

Example 5–2 Sample Input for Testing Order Approval Application

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body xmlns:ns1="http://xmlns.oracle.com/OracleRules1/OracleRules1_
DecisionService_1">
        <ns1:callFunctionStateful>
            <ns1:configURL></ns1:configURL>

            <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:callFunctionStateful>
    </soap:Body>
</soap:Envelope>
```

4. Replace the values in the input shown in Example 5–2 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>
```

6 Working with Decision Functions

Use a decision function to call rules from a Java application, from a composite, or from a BPEL process.

This chapter includes the following sections:

6.1 Introduction to Decision Functions

A decision function is a function that is configured declaratively. A decision function performs the following operations:

- Collects rulesets and other decision functions under a single executable umbrella
- Handles the assertion of inputs as rule facts into the Oracle Business Rules Engine working memory
- Collects the consequent actions to be executed
- Runs rulesets
- Returns results

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.

6.2 Working with Decision Functions

A decision function is a function that is configured declaratively.

6.2.1 How to Add or Edit a Decision Function

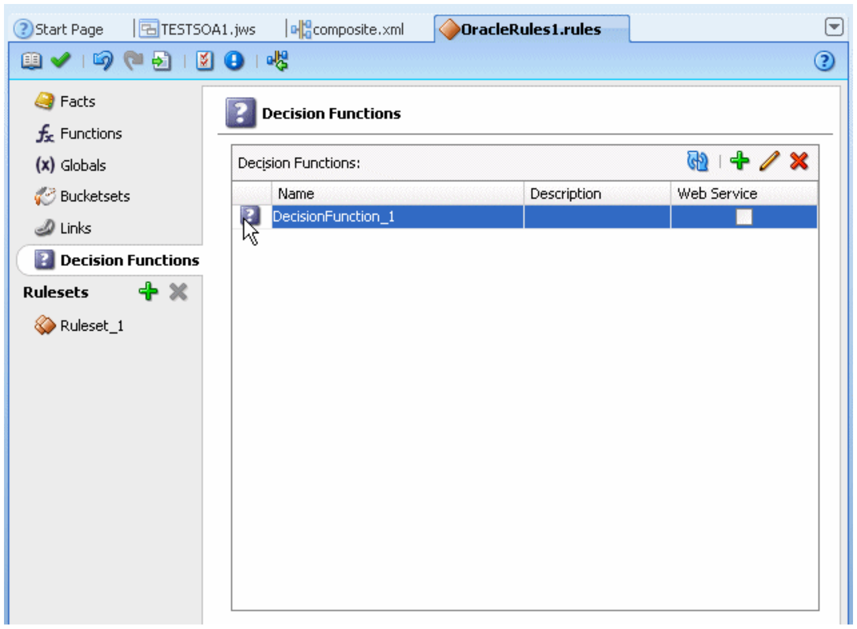
You use Rules Designer to add a decision function.

To add a decision function:

1. In Rules Designer, select the **Decision Functions** navigation tab.
2. In the **Decision Functions** area, click **Create...**.

3. Enter the decision function name in the **Name** field, or use the default name as Figure 6–1 shows.

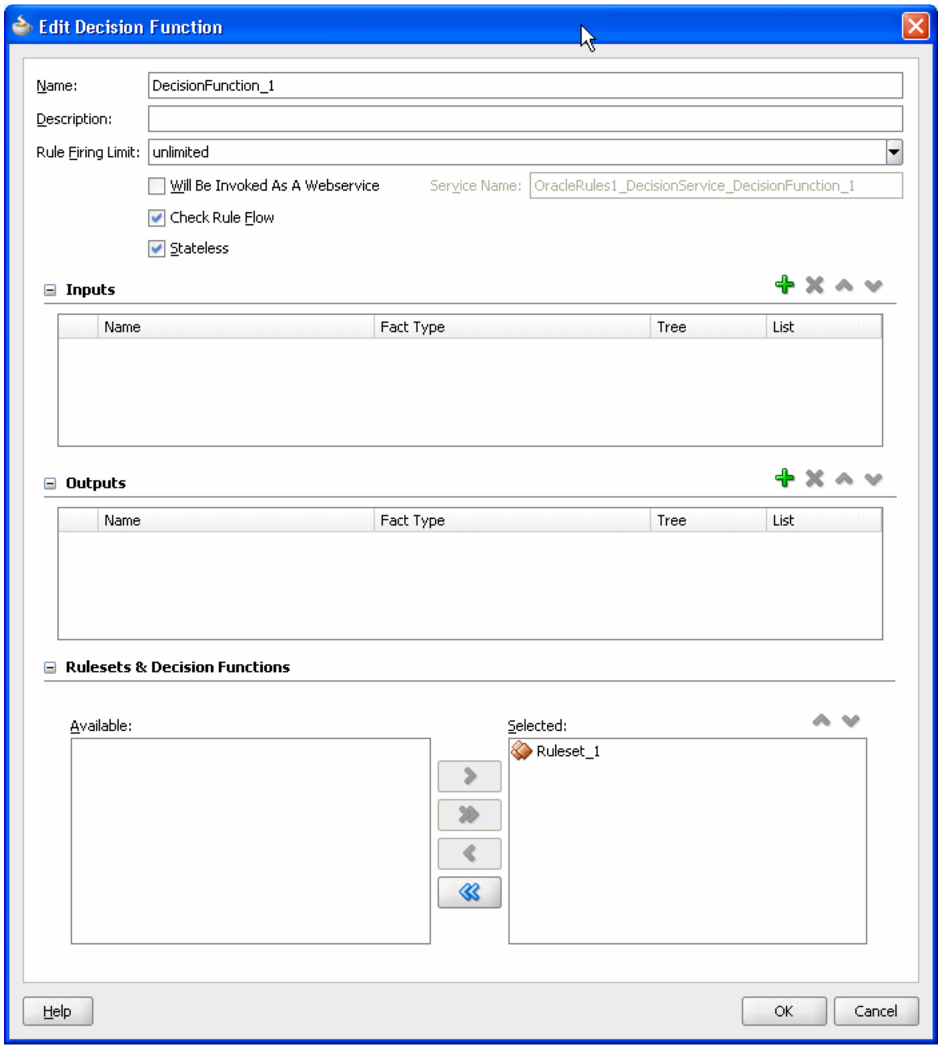
Figure 6–1 The Decision Functions Area in Rules Designer



Name	Description	Web Service
DecisionFunction_1		☐

4. In the decision functions table, double-click the decision function icon. For example, double-click the decision function icon for **DecisionFunction_1**. This displays the Edit Decision Function dialog, as shown in Figure 6–2.

Figure 6–2 Edit Decision Function Dialog


Edit Decision Function

Name: DecisionFunction_1

Description:

Rule Firing Limit: unlimited

☐ Will Be Invoked As A Webservice   Service Name: OracleRules1_DecisionService_DecisionFunction_1

☑ Check Rule Flow

☑ Stateless

Inputs

Name	Fact Type	Tree	List

Outputs

Name	Fact Type	Tree	List

Rulesets & Decision Functions

Available:

Selected: Ruleset_1

5. In the **Name** field, enter a name or accept the default value.
6. In the **Description** field, optionally enter a description.
7. In the **Rule Firing Limit** field, select **unlimited**. In some cases when you are debugging a decision function, you may want to enter a value other than `unlimited` for the rule firing limit. For more information, see Section 6.3.2, "What You Need to Know About Rule Firing Limit Option for Debugging Rules".
8. Select the appropriate decision function options:
   - **Will be invoked as a Web Service**: select whether the decision function will be invoked as a Web Service. For more information, see Section 6.3.4, "What You Need to Know About the Will Be Invoked As Web Service Option".
   - **Check Rule Flow**: when selected, this option specifies that the rule flow is checked to ensure that facts of the appropriate type are either explicit inputs to the decision function or are asserted by rules in the rule flow. However, when this is selected this does not always produce useful information; there are cases where facts can be asserted in Java code that uses the decision function, but this code might not be available at design time. In this case, validation warnings may produced with Check Rule Flow selected may not be useful.

- **Stateless**: when selected specifies the decision function is stateless. For more information, see Section 6.3.5, "What You Need to Know About the Decision Function Stateless Option".

9. In the Inputs Table, 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 dropdown list.
   - **Tree** - When unselected, the input is asserted using the `assert` function. When selected, the input is asserted using the `assertTree` function. When selected it is expected that the input object or objects are the root of an object tree that is connected in one-to-many relationships with other objects using `List` or array type properties. For more information, see Section 4.8, "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`).
10. In the Outputs Table, 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 dropdown 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 Section 4.8, "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 Section 6.3.3, "What You Need to Know to About Decision Function Arguments".
11. In the **Rulesets and Decision Functions** area, use the shuttle to move items from the **Available** box to the **Selected** box.
12. Select an item in the **Selected** box, and click **Move Up** or **Move Down** as appropriate to order the rulesets and the decision functions.

To edit an existing decision function:

1. In Rules Designer, select the **Decision Functions** navigation tab.
2. Select the decision function to edit and click the **Edit** icon 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.

To change the order of inputs:

1. In Rules Designer, select the **Decision Functions** navigation tab.

2. Select the decision function to edit and click the **Edit** icon 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.

To change the order of outputs:

1. In Rules Designer, select the **Decision Functions** navigation tab.
2. Select the decision function to edit and click the **Edit** icon 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.

6.3 What You Need to Know About Decision Functions

A decision function is a function that is configured declaratively.

6.3.1 What You Need to Know About Using Undo Operation with Decision Functions

Rules Designer enables the option to undo and redo a delete operation on a decision function. You can perform this operation by clicking the **Undo Decision Function Changes** icon. However, if you perform the undo operation after deleting a decision function, this operation shows the decision function you deleted but can cause serious dictionary problems and can make the dictionary unusable.

In Rules Designer, do not use the **Undo Decision Function** icon after you delete a decision function.

6.3.2 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.

6.3.3 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: 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 unselected the input is asserted using the `assert` function. When **Tree** is selected, the input is asserted using the `assertTree` function. When **Tree** is selected it is expected that the input object or objects are the root of an object tree that is connected in one-to-many relationships with other objects using `List` or array type properties.

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 includes output for the decision function Outputs, depending on the number of output parameters. Table 6–1 describes the decision function output options.

Table 6–1 Decision Function Output Options Depending on Number of Outputs

Output Parameters	Description
0	The return value is null.
1	Exactly 1 element it is returned.
More than 1	The output then depends on the value of the **list** property: `true`: a `List[OutputFactType`*i*`]` is built for each decision function output. `false`: a `OutputFactType`*i* is built for each decision function output.

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 "Using the Business Rule Service Component" in the *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

6.3.4 What You Need to Know About the Will Be Invoked As Web Service Option

When the decision function option **Will be invoked as a Web Service** is selected, Oracle Business Rules generates an RL Language function which has input and output as XML datatypes. This option supports using a decision function from a Web Service.

This additional RL Language function has the following signature:

```
function <name>Service(T1 input1, ... Tn inputN, org.w3.dom.Node outputParentNode)
```

T*i* expands depending on the value of the decision function Input **List** option for input, input *i*, such that:

- **List** option unselected: `org.w3.dom.Element`
- **List** option selected: `java.util.List<org.w3.dom.Element>`

Input `Element` arguments are unmarshalled to result in `InputFactType` inputs. The resulting fact types are each marshalled to an `Element` and returned. These are passed to the generated function, which returns no result, an `OutputFactType`, or a `List`.

The marshalled output elements, if any, are added as child elements to the value passed in as the last argument of the function, `outputParentNode`. This node must be an `Element`, `Document`, or `DocumentFragment`.

For complete API information on `DecisionFunction`, see *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*.

6.3.5 What You Need to Know About the Decision Function Stateless Option

A decision function supports either stateful or stateless operation. The **Stateless** checkbox in the Edit Decision Function dialog provides support for these two modes of operation.

By default the **Stateless** checkbox 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 deselected 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 deselected 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 Working with Rules SDK Decision Point API

Oracle Business Rules SDK (Rules SDK) lets you write applications that access, create, modify, and execute rules in Oracle Business Rules dictionaries (and work with the contents of a dictionary). This chapter provides a brief description of Rules SDK and shows how to work with the Rules SDK Decision Point API.

This chapter includes the following sections:

For more information, see *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*.

7.1 Introduction to Rules SDK and the Car Rental Sample Application

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.

7.1.1 Introduction to 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.

7.1.2 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 is available online at

`http://www.oracle.com/technology/sample_code/products/rules`

For SOA samples online visit

`http://www.oracle.com/technology/sample_code/products/soa`

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.

7.1.3 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.

7.2 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:

- Section 7.2.1, "How to Create Data Model Elements for Use with a Decision Point"
- Section 7.2.2, "How to View a Decision Function to Call from the Decision Point"
- Section 7.2.3, "How to Create Rules or Decision Tables for the Decision Function"

7.2.1 How to Create Data Model Elements for Use with a Decision Point

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 Chapter 2, "Working with Data Model Elements". For more information, see Chapter 10, "Working with Oracle Business Rules and ADF Business Components".
- A dictionary containing fact definitions. For more information, see Chapter 3, "Working with Facts and Bucketsets".

To view the data model in the supplied car rental sample application:

1. In Rules Designer, click the **Facts** navigation tab.
2. Select the **Java Facts** tab, as shown in Figure 7–1.

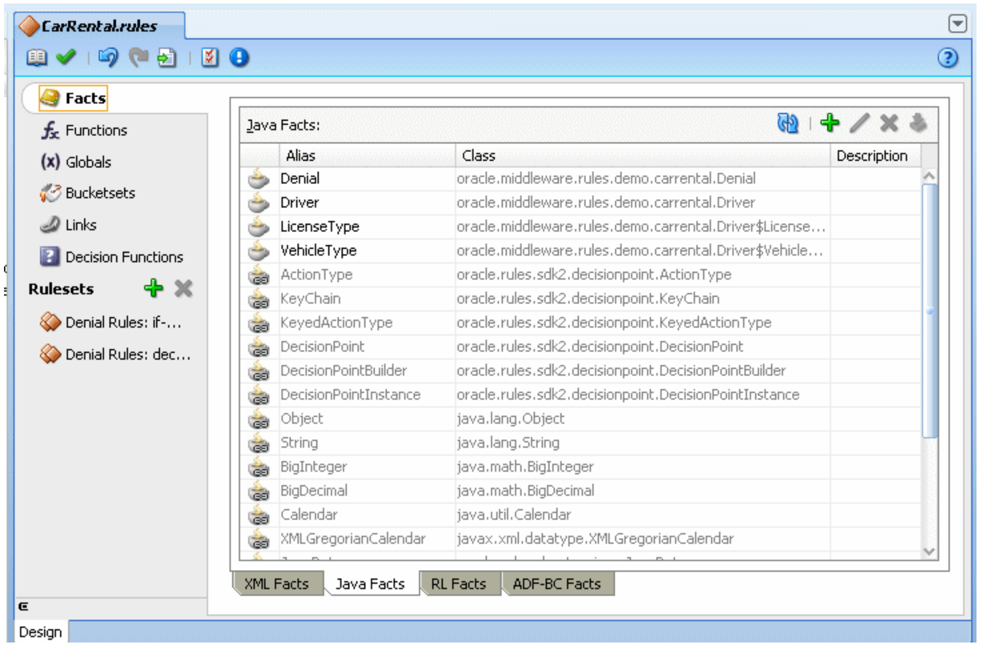
   The **Java Facts** tab shows four fact types imported, in addition to the fact types provided as built-in to the dictionary.

   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 7–1 Defined Java Facts for the Car Rental Sample Application



Alias	Class	Description
Denial	oracle.middleware.rules.demo.carrental.Denial	
Driver	oracle.middleware.rules.demo.carrental.Driver	
LicenseType	oracle.middleware.rules.demo.carrental.Driver$License...	
VehicleType	oracle.middleware.rules.demo.carrental.Driver$Vehicle...	
ActionType	oracle.rules.sdk2.decisionpoint.ActionType	
KeyChain	oracle.rules.sdk2.decisionpoint.KeyChain	
KeyedActionType	oracle.rules.sdk2.decisionpoint.KeyedActionType	
DecisionPoint	oracle.rules.sdk2.decisionpoint.DecisionPoint	
DecisionPointBuilder	oracle.rules.sdk2.decisionpoint.DecisionPointBuilder	
DecisionPointInstance	oracle.rules.sdk2.decisionpoint.DecisionPointInstance	
Object	java.lang.Object	
String	java.lang.String	
BigInteger	java.math.BigInteger	
BigDecimal	java.math.BigDecimal	
Calendar	java.util.Calendar	
XMLGregorianCalendar	javax.xml.datatype.XMLGregorianCalendar	

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.

To view the ruleset in the supplied car rental sample application:

1. In Rules Designer, expand the **CarRentalApplication**.
2. In the CarRentalApplication, expand **Resources**.
3. Double-click the **CarRental.rules**.

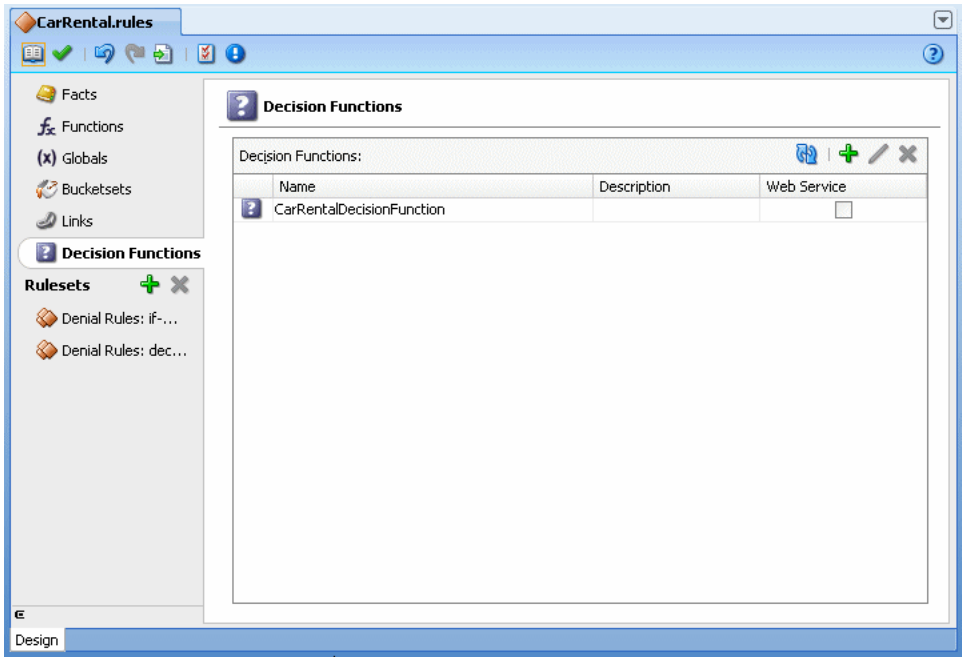
7.2.2 How to View a Decision Function to Call from the Decision Point

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 Chapter 6, "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 7–2.

Figure 7–2 Car Rental Sample Decision Function



Name	Description	Web Service
CarRentalDecisionFunction		☐

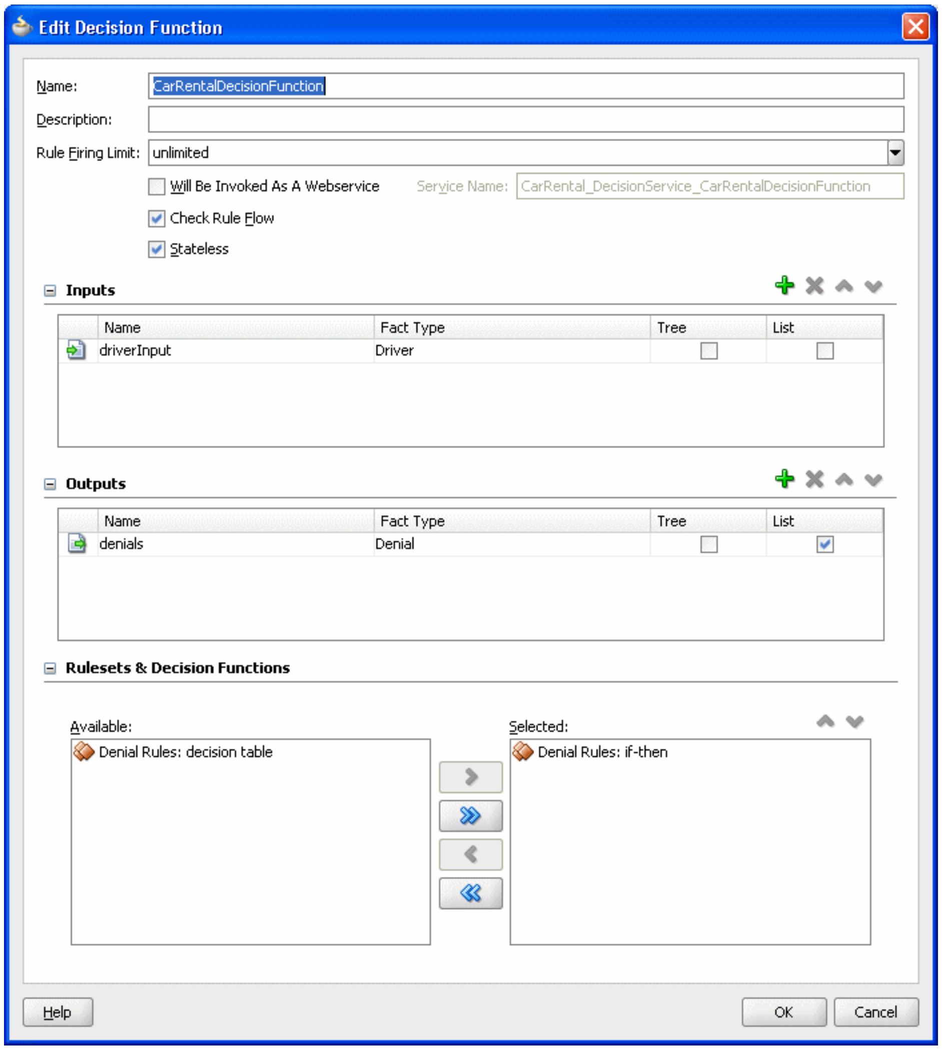
2. Select the row with CarRentalDecisionFunction and double-click the decision function icon. This opens the Edit Decision Function dialog as shown in Figure 7–3.

   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 7–3 Car Rental Decision Function for the Car Rental Sample Application



Name: CarRentalDecisionFunction
Description:
Rule Firing Limit: unlimited
☐ Will Be Invoked As A Webservice   Service Name: CarRental_DecisionService_CarRentalDecisionFunction
☑ Check Rule Flow
☑ Stateless

Inputs

Name	Fact Type	Tree	List
driverInput	Driver	☐	☐

Outputs

Name	Fact Type	Tree	List
denials	Denial	☐	☑

Rulesets & Decision Functions

Available: Denial Rules: decision table
Selected: Denial Rules: if-then

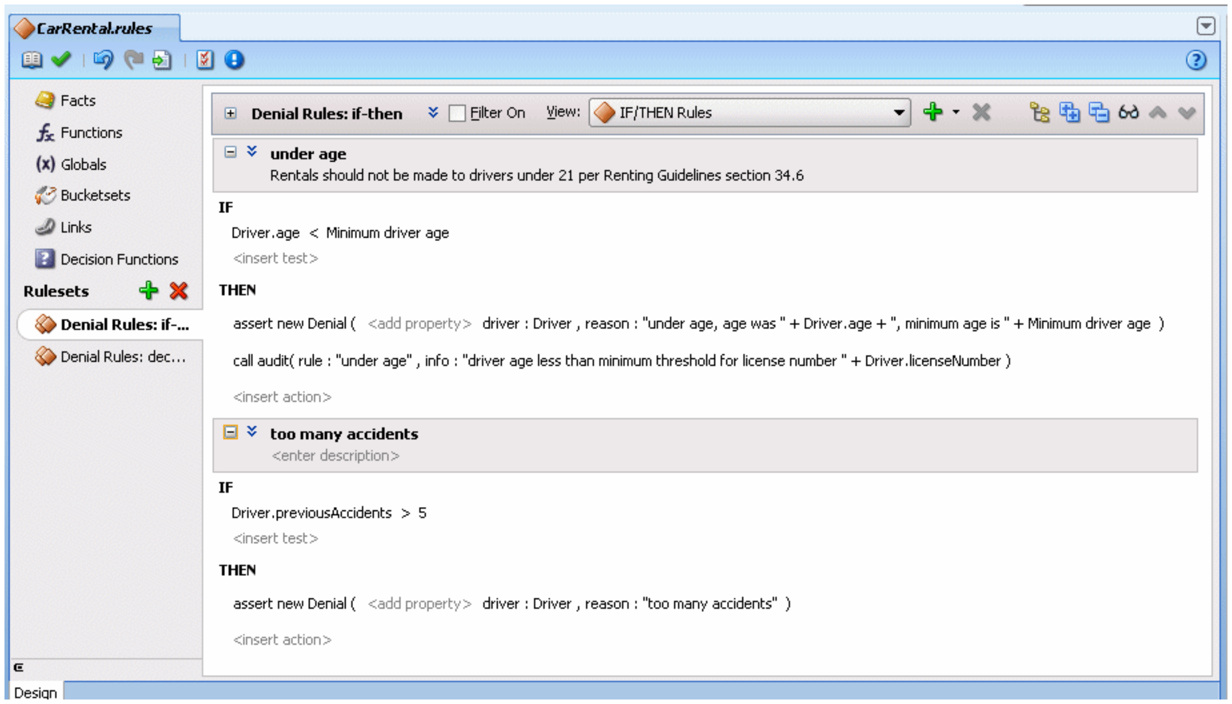
7.2.3 How to Create Rules or Decision Tables for the Decision Function

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 7–4.

Figure 7–4 Ruleset with IF/THEN Rules for the Car Rental Sample Application



under age
Rentals should not be made to drivers under 21 per Renting Guidelines section 34.6

IF
Driver.age < Minimum driver age
<insert test>

THEN
assert new Denial (<add property> driver : Driver , reason : "under age, age was " + Driver.age + ", minimum age is " + Minimum driver age)
call audit(rule : "under age" , info : "driver age less than minimum threshold for license number " + Driver.licenseNumber)
<insert action>

too many accidents
<enter description>

IF
Driver.previousAccidents > 5
<insert test>

THEN
assert new Denial (<add property> driver : Driver , reason : "too many accidents")
<insert action>

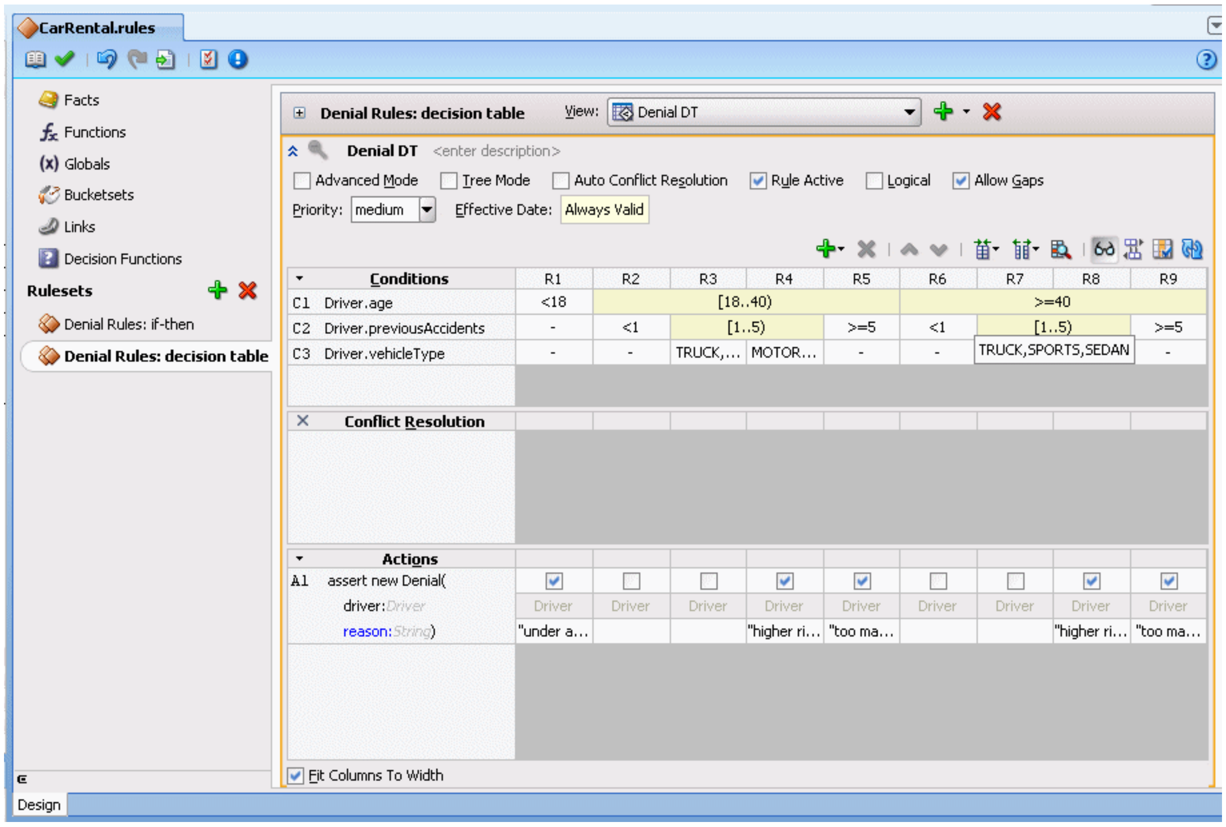
The **Denial Rules:if-then** ruleset includes two rules:

- **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. The rule also calls a user-defined function, `audit`, to provide some auditing output about why the `Denial` is created.
- **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.

To view the Decision Table in the car rental application:

1. In Rules Designer, click the **Denial Rules:decision table** ruleset, as shown in Figure 7–5.

Figure 7–5 Ruleset with Decision Table for the Car Rental Sample Application



CarRental.rules

Facts
Functions
Globals
Bucketsets
Links
Decision Functions
Rulesets
Denial Rules: if-then
Denial Rules: decision table

Denial Rules: decision table View: Denial DT

Denial DT <enter description>

☐ Advanced Mode ☐ Tree Mode ☐ Auto Conflict Resolution ☑ Rule Active ☐ Logical ☑ Allow Gaps

Priority: medium Effective Date: Always Valid

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9
C1 Driver.age	<18	[18..40)				>=40			
C2 Driver.previousAccidents	-	<1	[1..5)		>=5	<1	[1..5)		>=5
C3 Driver.vehicleType	-	-	TRUCK,...	MOTOR...	-	-	TRUCK,SPORTS,SEDAN		-

Conflict Resolution

Actions	R1	R2	R3	R4	R5	R6	R7	R8	R9
A1 assert new Denial(	✓	☐	☐	✓	✓	☐	☐	✓	✓
driver: Driver	Driver	Driver	Driver	Driver	Driver	Driver	Driver	Driver	Driver
reason: String)	"under a...			"higher ri...	"too ma...			"higher ri...	"too ma...

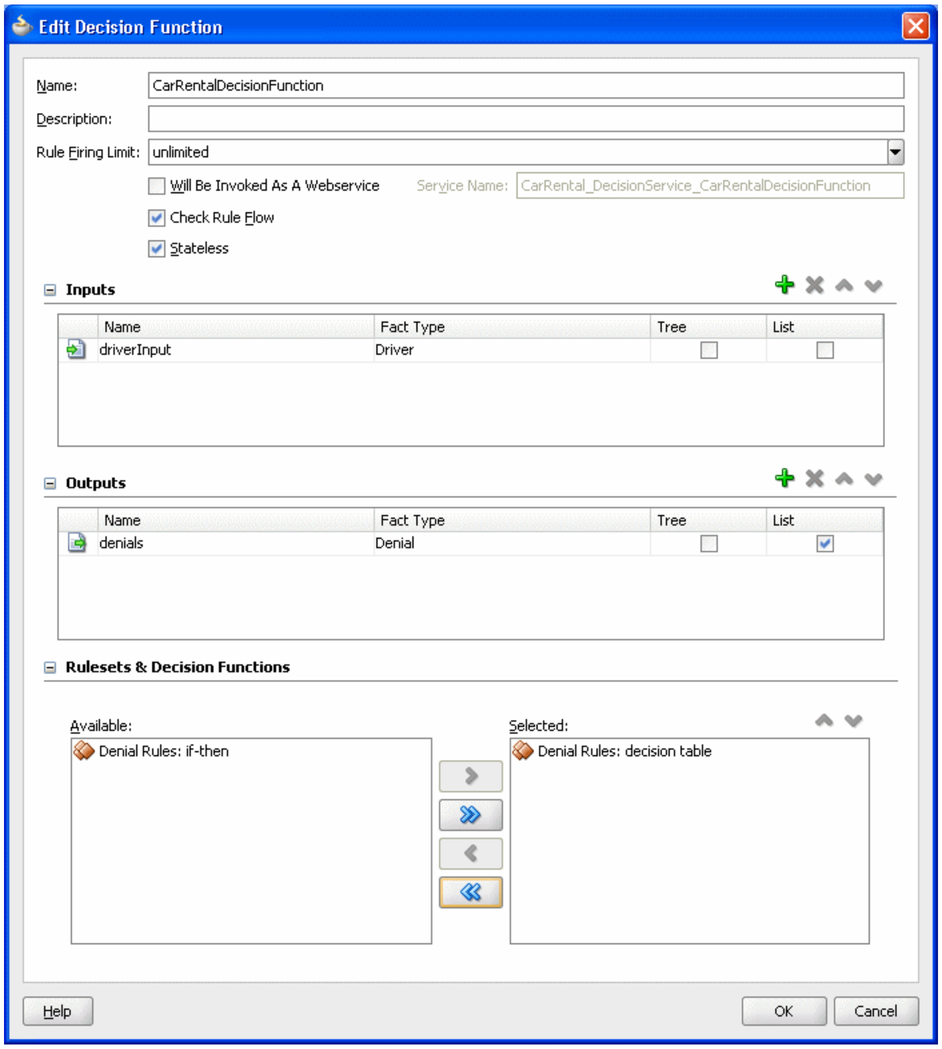
☑ Fit Columns To Width

Design

7.2.4 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 7–6.

Figure 7–6 Decision Function for Car Rental Sample with Decision Table Ruleset



Name: CarRentalDecisionFunction

Description:

Rule Firing Limit: unlimited

☐ Will Be Invoked As A Webservice   Service Name: CarRental_DecisionService_CarRentalDecisionFunction

☑ Check Rule Flow

☑ Stateless

Inputs

Name	Fact Type	Tree	List
driverInput	Driver	☐	☐

Outputs

Name	Fact Type	Tree	List
denials	Denial	☐	☑

Rulesets & Decision Functions

Available: Denial Rules: if-then

Selected: Denial Rules: decision table

7.3 Creating a Java Application Using Rules SDK Decision Point

When use Rules SDK in a development environment you of 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 Section 7.5, "What You Need to Know About Using Decision Point in a Production Environment".

The source code for Oracle Business Rules-specific samples is available online at

`http://www.oracle.com/technology/sample_code/products/rules`

For SOA samples online visit

`http://www.oracle.com/technology/sample_code/products/soa`

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 7–1 provides a summary of the different versions of the car rental sample.

For more information on working with the Rules SDK Decision Point API, see *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*.

Table 7–1 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.

7.3.1 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 7–1.

Example 7–1 Using the Decision Point Builder

```
    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();
        }
    }
```

Example 7–1 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 7–1 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 7–2 shows the `loadDictionary()` method. For more information, see Section 7.3.2, "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.

7.3.2 How to Use a Decision Point with a Pre-loaded Dictionary

Example 7–2 shows the `loadRuleDictionary()` method that loads an instance of RuleDictionary from a file.

Example 7–2 Load Rule Dictionary Method

```
    private static RuleDictionary loadRuleDictionary(){
        RuleDictionary dict = null;
        Reader reader = null;
        Writer writer = null;
        try {
            reader = new FileReader(new File(DICT_LOCATION));
            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();}}
            if (writer != null) { try { writer.close(); } catch (IOException ioe)
{ioe.printStackTrace();}}
        }

        return dict;
    }
```

7.3.3 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. Example 7–3 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.

Example 7–3 Checking Drivers with Threads that Invoke Decision Point

```
        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));
        }
```

Example 7–3 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 Section 7.3.4, "How to Create and Use Decision Point Instances".

Example 7–4 shows the code that waits for the threads to complete.

Example 7–4 Code to Await Thread Termination

```
        try {
            exec.awaitTermination(5, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        exec.shutdown();
```

```
    }
```

7.3.4 How to Create and Use Decision Point Instances

The `DriverCheckerRunnable` instances call the `checkDriver()` method. Example 7–5 shows the `checkDriver()` method that is defined in `CarRentalWithDecisionPoint`. The `checkDriver()` method handles invoking Decision Point with a `Driver` instance.

Example 7–5 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();
        }
    }

}
```

Example 7–5 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 caste it 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());
                  }
              }
  ```

7.4 Running the Car Rental Sample

In the car rental sample installed on your system, for the code shown in Example 7–2, 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 `CarRentalWithDecisionPointUsingPreloadedDictionary.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 dropdown list select **Run**.

Example 7–6 shows sample output.

Example 7–6 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
```

7.5 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 Section 7.3.1, "How to Add a Decision Point Using Decision Point Builder" and Section 7.3.2, "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 Section 9.4, "Adding a Servlet with Rules SDK Calls for Grades Sample Application".

Example 7–7 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`.

Example 7–7 Using Decision Point Builder with MDS Repository

```
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 Example 7–1, Example 7–7 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. Example 7–8 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.

Example 7–8 Using the DictionaryFQN Method with MDS Repository

```
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";
```

8 Testing Business Rules

You can test your rules and Decision Tables from Rules Designer by creating an Oracle Business Rules Function. In an SOA application or in an application that accesses Oracle Business Rules with a decision function with a web service, you can test the rules at runtime with Oracle Enterprise Manager Fusion Middleware Control Console using the Test function.

This chapter includes the following sections:

- Section 8.1, "Testing Oracle Business Rules at Design Time"
- Section 8.2, "Testing Oracle Business Rules at Runtime"

8.1 Testing Oracle Business Rules at Design Time

You can define a test function to run without deploying an application. This allows you to call a decision function at runtime and to test data model elements and rulesets.

8.1.1 How to Test Rules Using a Test Function in Rules Designer

You can use Oracle Business Rules Functions to test rules from within Rules Designer. The **Test Function** icon is active only for functions that take no parameters and return `boolean`. In the body of the function you create input facts, call a decision function, and check the output to validate the facts the decision function returns are as expected.

To enable logging you call `RL.watch.all()`. To run the function you click the **Test Function** icon in the **Functions** table.

For more information about functions, see Section 2.5, "Working with Oracle Business Rules Functions".

To test rules using a test function:

1. Confirm that your dictionary is valid.

   For more information on dictionary validation, see Section 4.4.4, "How to Validate a Dictionary".

2. In Rules Designer, select the **Functions** navigation tab.
3. In the **Functions** area click **Create...**.
4. Enter the function name in the **Name** field, or use the default name.
5. Select the return type from the **Return Type** dropdown 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.

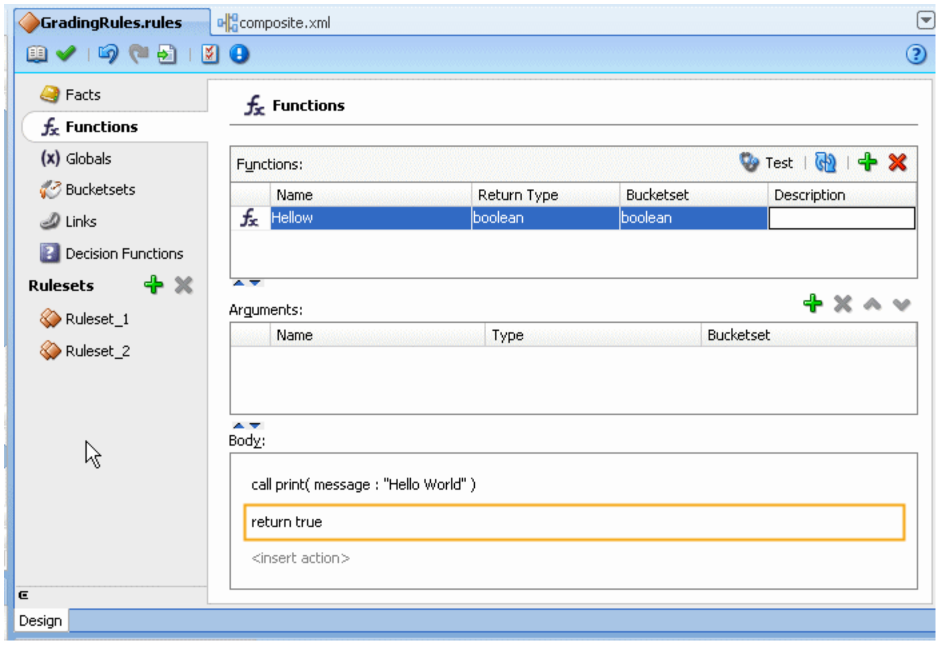
   In the body of the function you can call a decision function using `assign new` to call and get the return value of the decision function. Thus, to test a decision function you create the input data and call the decision function.

   Example 8–1 shows a simple test function that calls `print`. Figure 8–1 shows the test function definition. For information on adding a test function that calls a decision function, see Section 8.1.3, "How to Test a Decision Function Using an Oracle Business Rules Function".

Example 8–1 Test Function Body

```
call print("Hello World")
return true
```

Figure 8–1 Adding a Test Function



Name	Return Type	Bucketset	Description
Hellow	boolean	boolean	

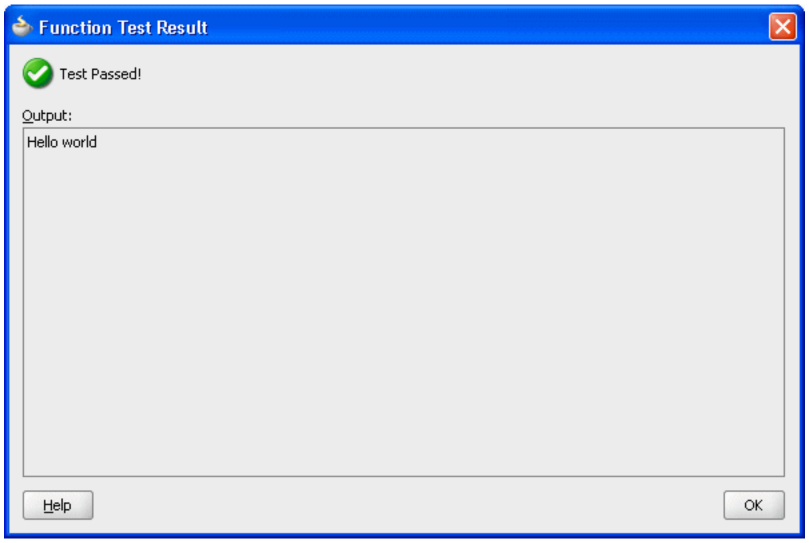
Name	Type	Bucketset

```
call print( message : "Hello World" )
return true
<insert action>
```

8. In the **Functions** table select the function and click the **Test Function** icon.

   The output is displayed in a Function Test Result dialog, as Figure 8–2 shows.

Figure 8–2 Test Function Results Dialog



Test Passed!

Output:

```
Hello world
```

9. Click **OK** to dismiss the Function Test Result dialog.

8.1.2 What You Need to Know About Testing Using an Oracle Business Rules Function

The **Test Function** icon is only active when the dictionary is valid (the Business Rule validation log is empty). The **Test Function** icon is gray if the dictionary associated with the function contains any validation warnings.

8.1.3 How to Test a Decision Function Using an Oracle Business 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 Section 6.1, "Introduction to Decision Functions" and Section 2.5, "Working with 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 Section 4.4.4, "How to Validate a Dictionary"

2. In Rules Designer, select the **Functions** navigation tab.
3. In the **Functions** area click the **Create...** icon.
4. Enter the function name in the **Name** field, or use the default name.
5. Select the return type from the **Return Type** dropdown 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).

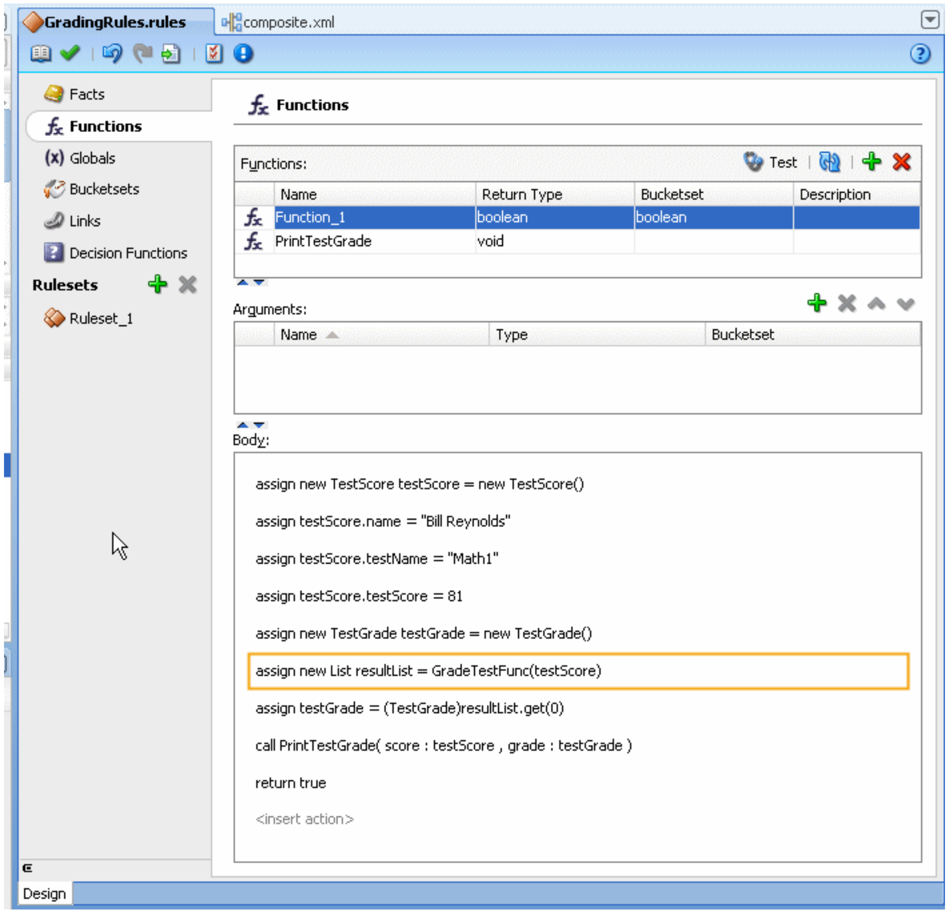
A decision function call returns a `List`. Thus, to test a decision function in a test function you do the following:

- 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)
```

Figure 8–3 shows a test function that calls a decision function.

Figure 8–3 Test Function to Call a Decision Function that Returns a List



Functions:

	Name	Return Type	Bucketset	Description
fx	Function_1	boolean	boolean	
fx	PrintTestGrade	void		

Arguments:

	Name	Type	Bucketset

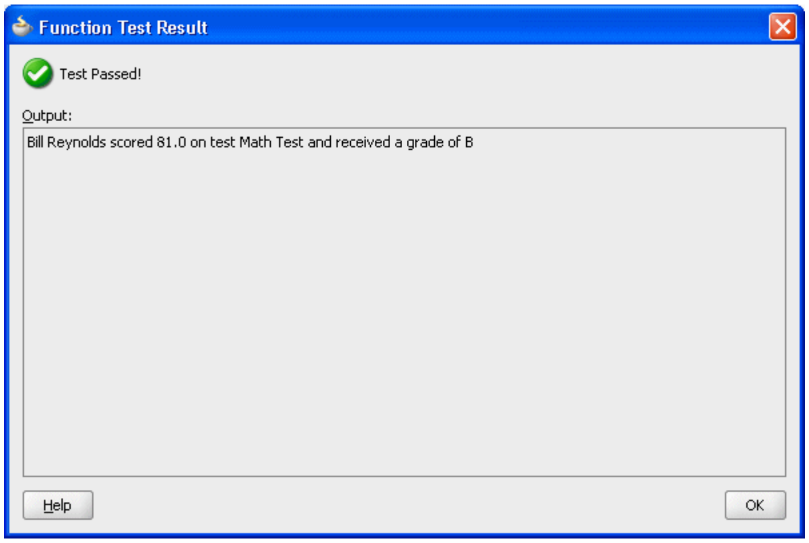
Body:

```
assign new TestScore testScore = new TestScore()
assign testScore.name = "Bill Reynolds"
assign testScore.testName = "Math1"
assign testScore.testScore = 81
assign new TestGrade testGrade = new TestGrade()
assign new List resultList = GradeTestFunc(testScore)
assign testGrade = (TestGrade)resultList.get(0)
call PrintTestGrade( score : testScore , grade : testGrade )
return true
<insert action>
```

8. Select the function and click the **Test Function** icon.

The function is executed. The output is shown in a Function Test Result dialog, as Figure 8–4 shows.

Figure 8–4 Test Function Results for Grade Test



Test Passed!

Output:

```
Bill Reynolds scored 81.0 on test Math Test and received a grade of B
```

9. Click **OK** to dismiss the Function Test Result dialog.

8.1.4 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:

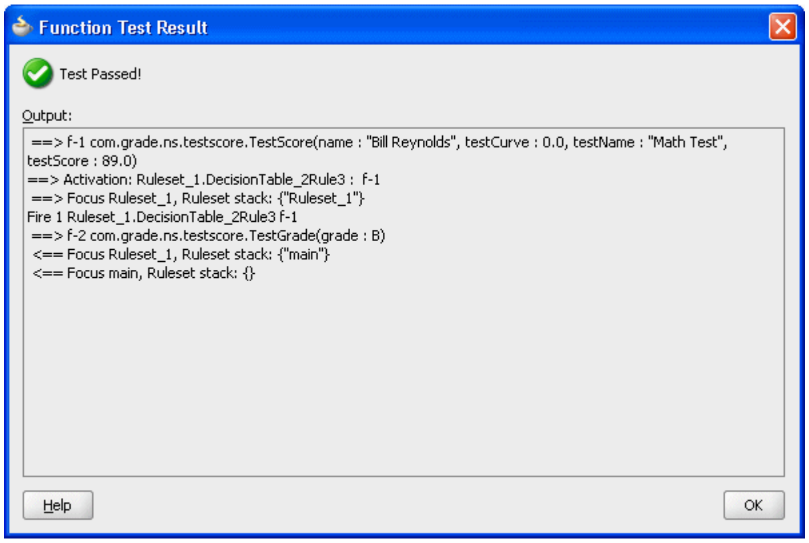
- The **Test Function** icon is gray if the dictionary associated with the test Oracle Business Rules Function contains any validation warnings. The **Test Function** icon is only shown when the dictionary validates without warnings.
- To enable logging you can call `RL.watch.all()`. For more information on RL Language functions, see *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*. In this guide, `RL.watch.all()` is an alias for the RL Language function `watchAll()`.
- As an alternative to the example shown in Figure 8–3, you can enter the function body that is shown in Example 8–2. This function runs and shows the `RL.watch.all()` output. The dialog shows "Test Passed" when the grade is in the B range as shown in Figure 8–5. The dialog shows "Test Failed" when the grade asserted is not in the B range, as shown in Figure 8–6.

Example 8–2 Function Body with True or False Return Value

```
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" as shown in Figure 8–5. For the `testScore` value 91, this returns "Test Failed", as shown in Figure 8–6.

Figure 8–5 Test Passed Test Function Output

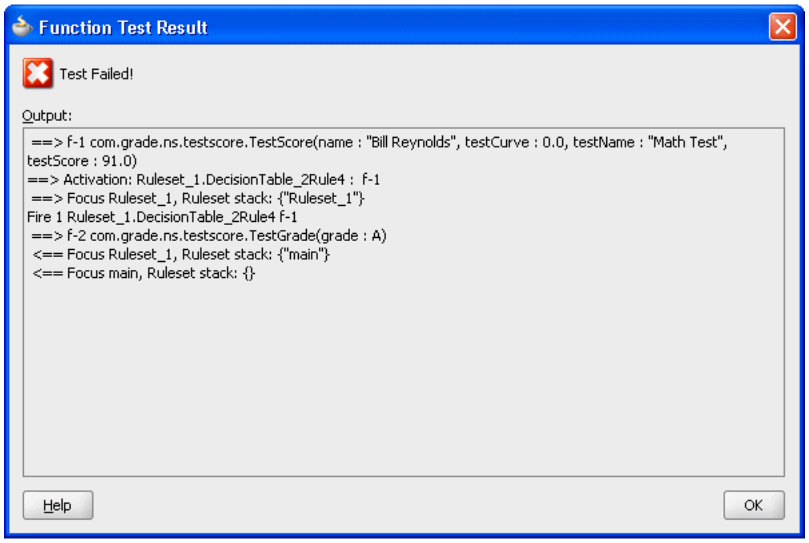


Test Passed!

Output:

```
 ==> f-1 com.grade.ns.testscore.TestScore(name : "Bill Reynolds", testCurve : 0.0, testName : "Math Test",
testScore : 89.0)
==> Activation: Ruleset_1.DecisionTable_2Rule3 :  f-1
 ==> Focus Ruleset_1, Ruleset stack: {"Ruleset_1"}
Fire 1 Ruleset_1.DecisionTable_2Rule3 f-1
 ==> f-2 com.grade.ns.testscore.TestGrade(grade : B)
 <== Focus Ruleset_1, Ruleset stack: {"main"}
 <== Focus main, Ruleset stack: {}
```

Figure 8–6 Test Failed Test Function Output



Test Failed!

Output:

```
 ==> f-1 com.grade.ns.testscore.TestScore(name : "Bill Reynolds", testCurve : 0.0, testName : "Math Test",
testScore : 91.0)
==> Activation: Ruleset_1.DecisionTable_2Rule4 :  f-1
 ==> Focus Ruleset_1, Ruleset stack: {"Ruleset_1"}
Fire 1 Ruleset_1.DecisionTable_2Rule4 f-1
 ==> f-2 com.grade.ns.testscore.TestGrade(grade : A)
 <== Focus Ruleset_1, Ruleset stack: {"main"}
 <== Focus main, Ruleset stack: {}
```

8.2 Testing Oracle Business Rules at Runtime

In an 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 function.

For more information on using the Test function, see *Oracle Fusion Middleware Administrator's Guide for Oracle SOA Suite*.

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

You can use Oracle JDeveloper to create a rule-enabled non-SOA Java EE application with Oracle Business Rules. This chapter shows a sample application, a Java Servlet, that runs as a Java EE application using Oracle Business Rules (this describes using of Oracle Business Rules without an SOA composite).

This chapter includes the following sections:

The source code for Oracle Business Rules-specific samples is available online at

`http://www.oracle.com/technology/sample_code/products/rules`

For SOA samples online visit

`http://www.oracle.com/technology/sample_code/products/soa`

9.1 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.

9.2 Creating an Application and a Project for Grades Sample Application

To create the application and the project for the grades sample application, you 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

9.2.1 How to Create a Fusion Web Application for the Grades Sample Application

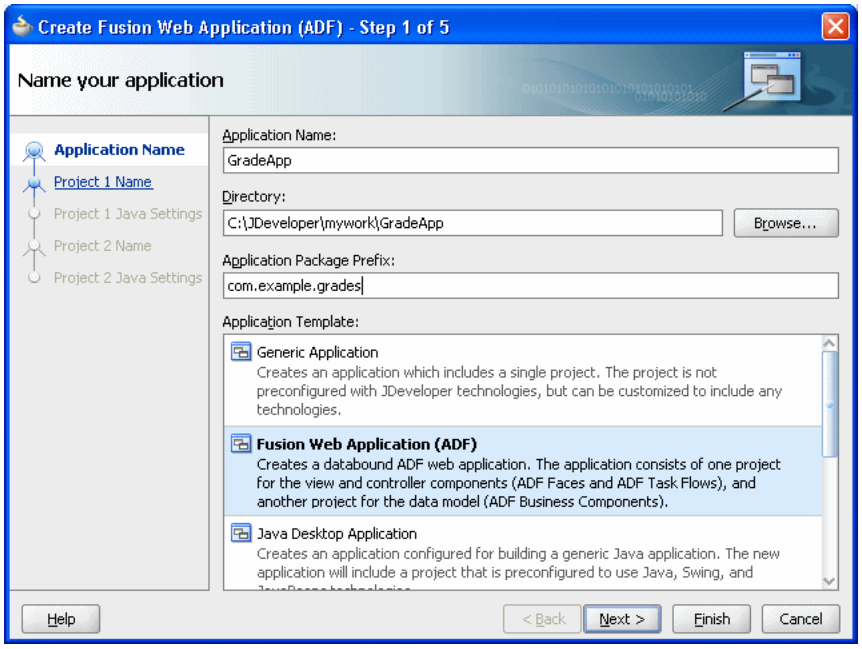
To work with Oracle Business Rules and create a Java EE application, you first need to create the application in Oracle JDeveloper.

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 dropdown 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



Create Fusion Web Application (ADF) - Step 1 of 5

Name your application

Application Name: GradeApp

Directory: C:\JDeveloper\mywork\GradeApp

Application Package Prefix: com.example.grades

Application Template:

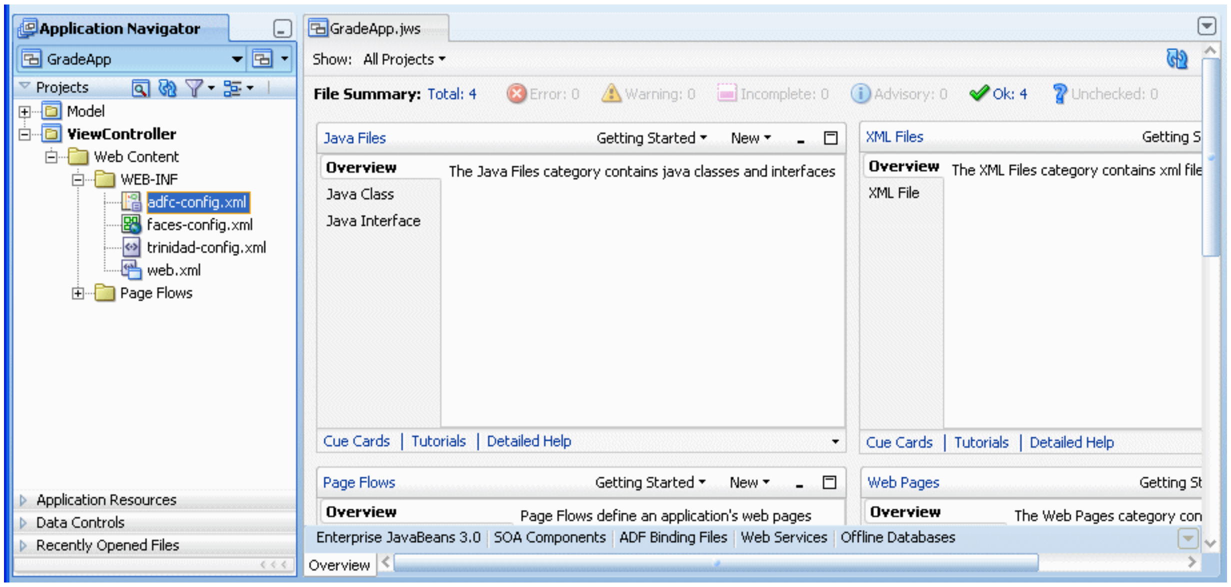
Generic Application
Creates an application which includes a single project. The project is not preconfigured with JDeveloper technologies, but can be customized to include any technologies.

Fusion Web Application (ADF)
Creates a databound ADF web application. The application consists of one project for the view and controller components (ADF Faces and ADF Task Flows), and another project for the data model (ADF Business Components).

Java Desktop Application
Creates an application configured for building a generic Java application. The new application will include a project that is preconfigured to use Java, Swing, and

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



File Summary: Total: 4 Error: 0 Warning: 0 Incomplete: 0 Advisory: 0 Ok: 4 Unchecked: 0

The Java Files category contains java classes and interfaces

The XML Files category contains xml file

Page Flows define an application's web pages

The Web Pages category con

9.2.2 How to Create the Grades Project

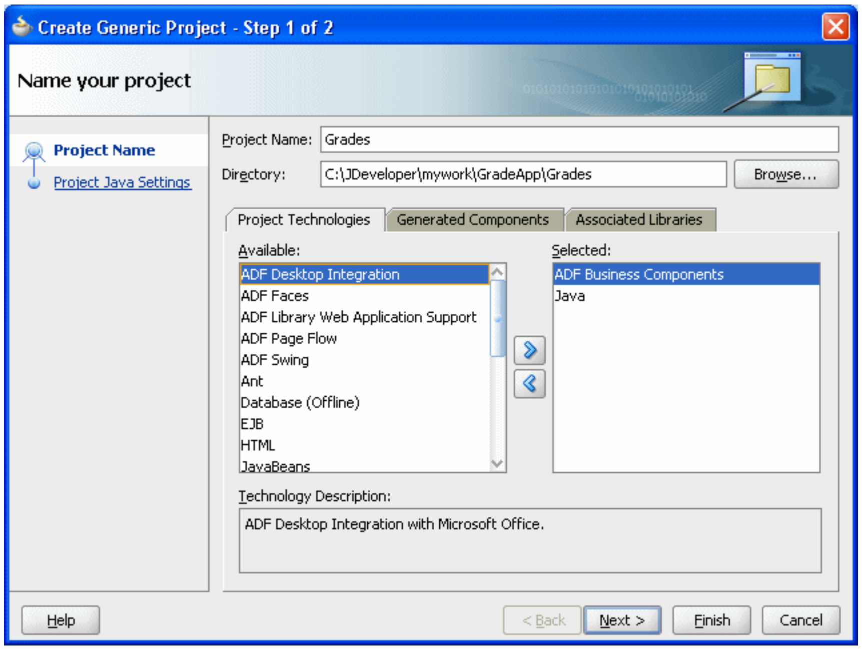
In the Grades sample application you do not use the Model or ViewController projects. You create a project in the application for the grades sample project.

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



Create Generic Project - Step 1 of 2

Name your project

Project Name: Grades

Directory: C:\JDeveloper\mywork\GradeApp\Grades

Available:
ADF Desktop Integration
ADF Faces
ADF Library Web Application Support
ADF Page Flow
ADF Swing
Ant
Database (Offline)
EJB
HTML
JavaBeans

Selected:
ADF Business Components
Java

Technology Description:
ADF Desktop Integration with Microsoft Office.

5. Click **Finish**. This adds the Grades project.

9.2.3 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, shown in Example 9–1. You can create and store the schema file locally and then use Oracle JDeveloper to copy the file to your project.

Example 9–1 grades.xsd Schema

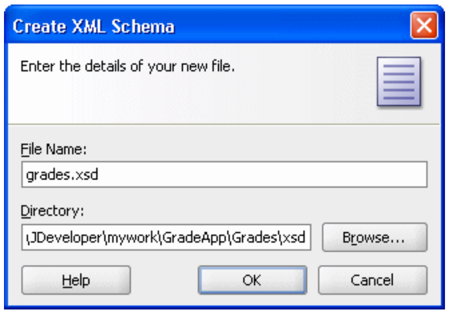
```
<?xml version= '1.0' encoding= 'UTF-8' ?>
<xs:schema targetNamespace="http://example.com/grades"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://example.com/grades"
        attributeFormDefault="qualified" elementFormDefault="qualified"
        xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
        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 shown in Example 9–1 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**.
7. Click **OK**.
8. In the Create XML Schema dialog, in the **File Name** field enter `grades.xsd`.
9. 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

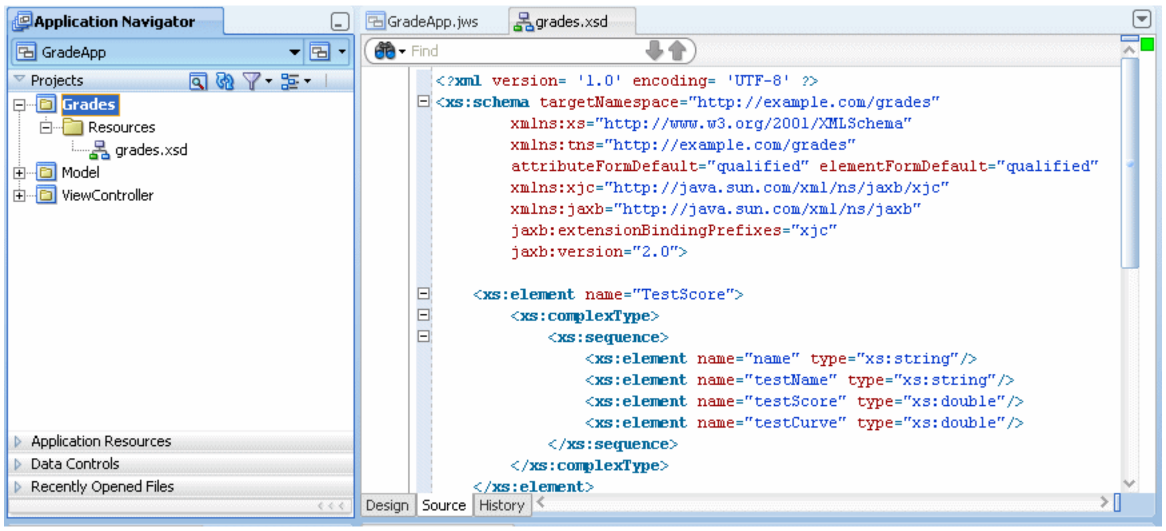


File Name: grades.xsd
Directory: \JDeveloper\mywork\GradeApp\Grades\xsd

10. Click **OK**.

11. In the `grades.xsd` file, select the **Source** tab.
12. Copy the schema shown in Example 9–1 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



```
<?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>
```

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**.

9.2.4 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.

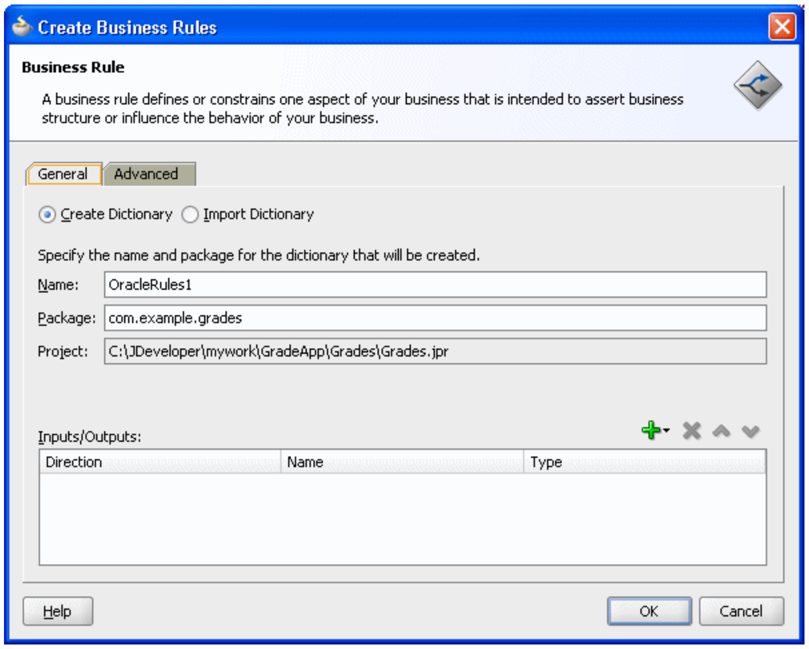
To use business rules with Oracle JDeveloper, you do the following:

- 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...**.
3. Select the **All Technologies** tab.
4. In the New Gallery, in the **Categories** area, expand Business Tier and select **Business Rules**.
5. In the New Gallery, in the **Items** area, select **Business Rules**.
6. 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



A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business.

Create Dictionary [✓] Import Dictionary

Specify the name and package for the dictionary that will be created.

Name: OracleRules1

Package: com.example.grades

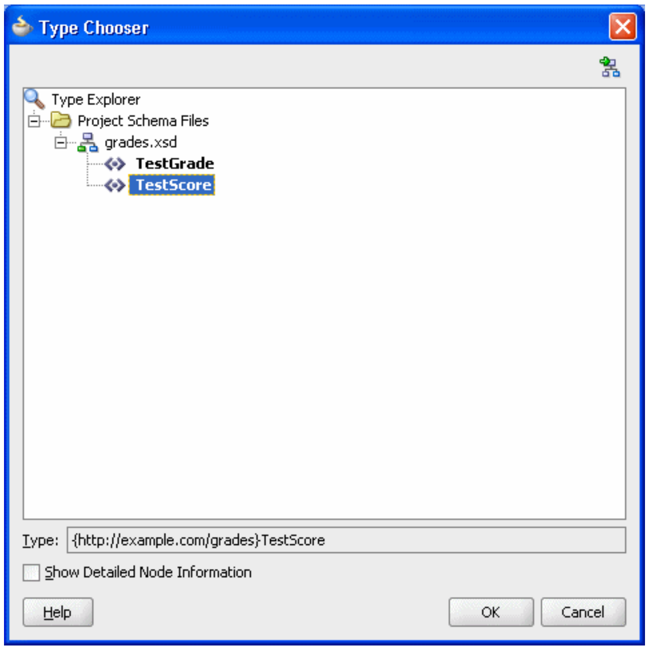
Project: C:\JDeveloper\mywork\GradeApp\Grades\Grades.jpr

Inputs/Outputs:

Direction	Name	Type

7. In the **Name** field, enter a name to name the dictionary. For example, enter `GradingRules`.
8. To add an input, from the dropdown list next to the **Add** icon select **Input...**.
9. In the Type Chooser, expand the Project Schemas Files folder and expand `grades.xsd`.
10. Select the input **TestScore**, as shown in Figure 9–7.

Figure 9–7 Shows the Type Chooser Dialog with TestScore Element

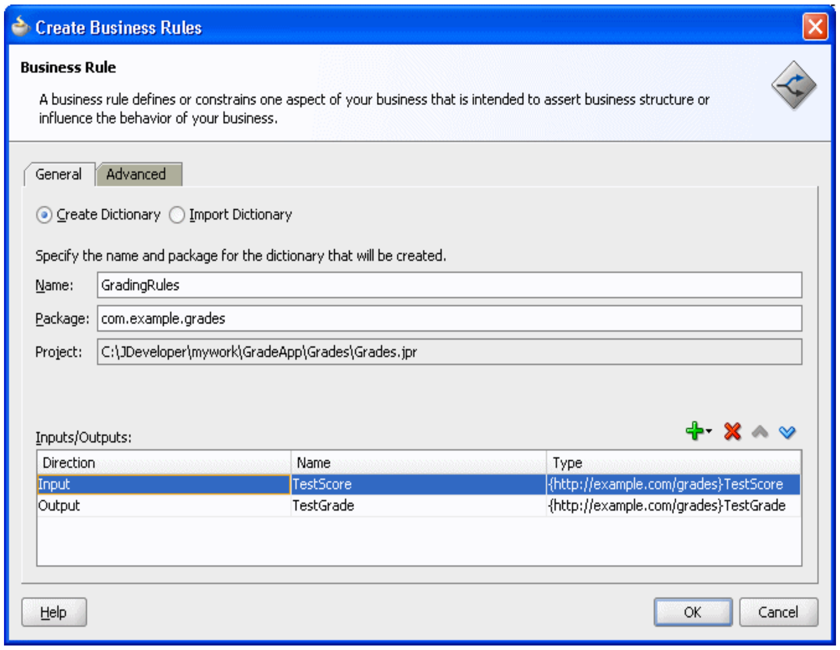


Type: {http://example.com/grades}TestScore

11. On the Type Chooser window, click **OK**. This displays the Create Business Rules dialog.
12. 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



A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business.

(● Create Dictionary) ○ Import Dictionary

Specify the name and package for the dictionary that will be created.

Name: GradingRules

Package: com.example.grades

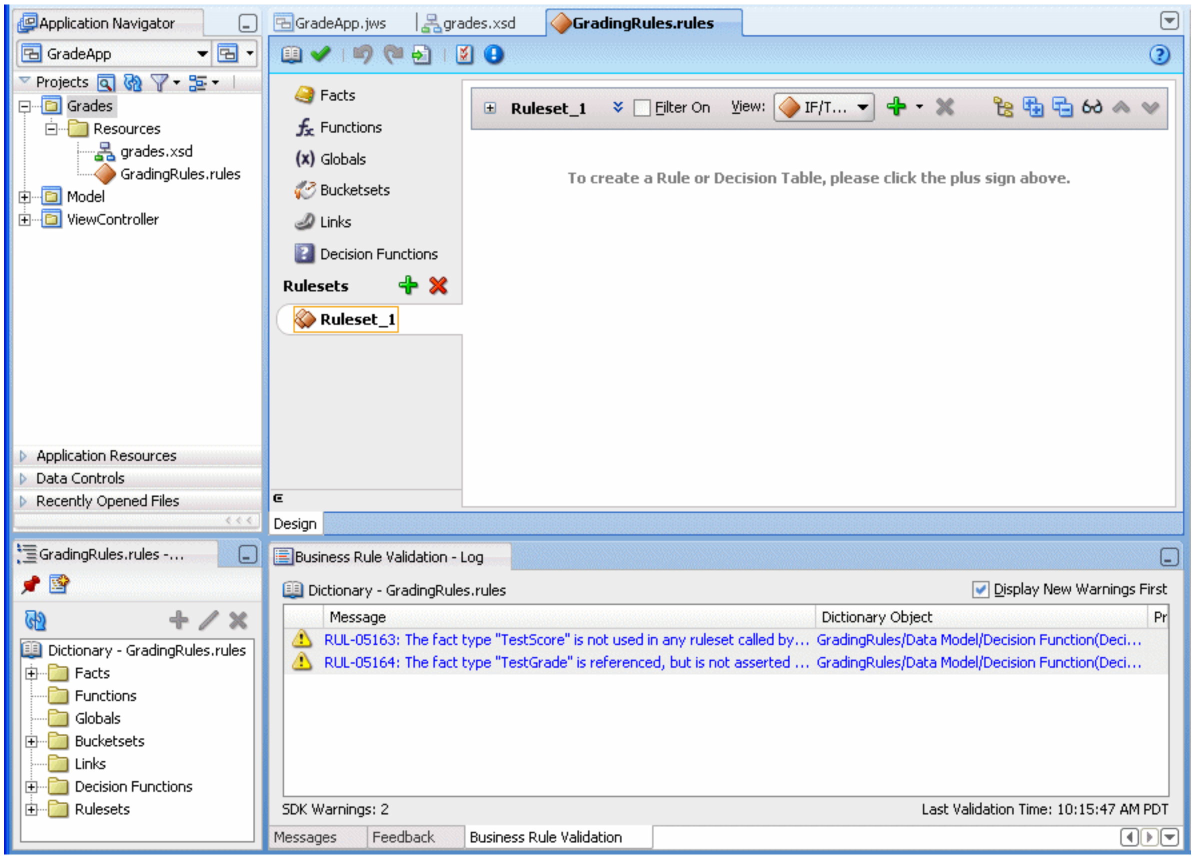
Project: C:\JDeveloper\mywork\GradeApp\Grades\Grades.jpr

Inputs/Outputs:

Direction	Name	Type
Input	TestScore	{http://example.com/grades}TestScore
Output	TestGrade	{http://example.com/grades}TestGrade

13. Click **OK**. Oracle JDeveloper creates the GradingRules dictionary as shown in Figure 9–9.
14. In the **File** menu, select **Save All** to save your work.

Figure 9–9 Shows the New Grading Rules Dictionary



To create a Rule or Decision Table, please click the plus sign above.

Dictionary - GradingRules.rules

Message	Dictionary Object
RUL-05163: The fact type "TestScore" is not used in any ruleset called by...	GradingRules/Data Model/Decision Function(Deci...
RUL-05164: The fact type "TestGrade" is referenced, but is not asserted ...	GradingRules/Data Model/Decision Function(Deci...

SDK Warnings: 2

Last Validation Time: 10:15:47 AM PDT

Note that the business rule validation log area for the new dictionary shows several validation warnings. You remove these validation warning messages as you modify the dictionary in later steps.

9.3 Creating Data Model Elements and Rules for the Grades Sample Application

To create the data model and the business rules for the Grades sample application, you do the following:

- Create Bucketsets 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

9.3.1 How to Create Bucketsets for Grades Sample Application

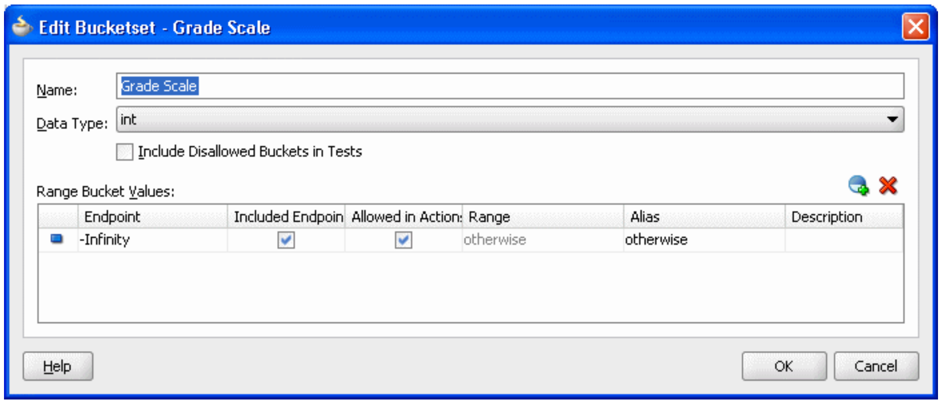
In this example you associate a bucketset with a fact type. This supports using a Decision Table where you need bucketsets that specify how to draw values for each cell in the Decision Table (for the conditions in the Decision Table).

To create the bucketset for the grades sample application:

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

2. From the dropdown list next to the **Create BucketSet...** icon, select **List of Ranges**.
3. For the bucketset, double-click in the Name field to select the default name.
4. Enter `Grade Scale`, and press **Enter** to accept the bucketset name.
5. In the **Bucketsets** table, double-click the bucket icon for the Grade Scale bucketset to display the Edit Bucketset dialog as shown in Figure 9–10.

Figure 9–10 Grade Scale Bucketset



Name: Grade Scale

Data Type: int

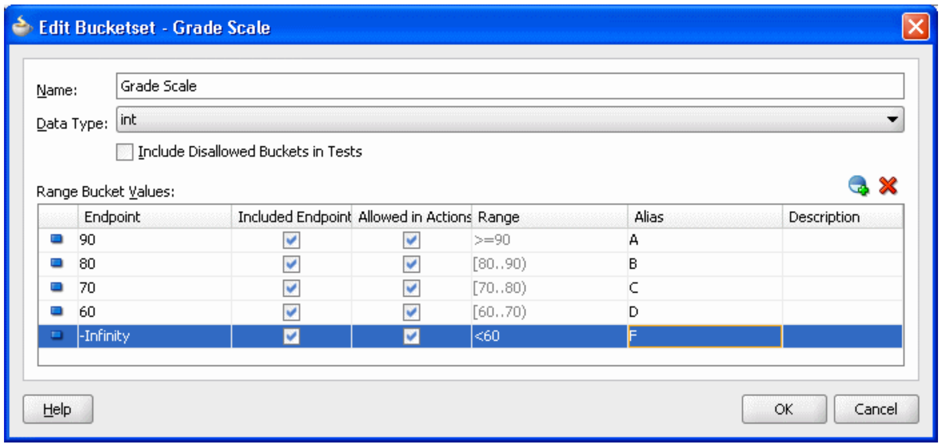
Include Disallowed Buckets in Tests

Range Bucket Values:

Endpoint	Included Endpoin	Allowed in Action:	Range	Alias	Description
-Infinity	✓	✓	otherwise	otherwise	

6. In the Edit Bucketset dialog, click **Add Bucket** to add a bucket.
7. Click **Add Bucket** three times to add three more buckets.
8. In the **Endpoint** field, enter 90 for the top endpoint and press **Enter** to accept the new value.
9. For the next bucket, in the **Endpoint** field enter 80 and press **Enter** to accept the new value.
10. Similarly, for the next two buckets enter values in the **Endpoint** field, values 70 and 60.
11. In the **Included Endpoint** field for each bucket select each checkbox.
12. Modify the **Alias** field for each value to enter the values A, B, C, D, and F, for each corresponding range, as shown in Figure 9–11 (press **Enter** after you add each alias).

Figure 9–11 Grade Scale Bucketset with Grade Values Added



Name: Grade Scale

Data Type: int

Include Disallowed Buckets in Tests

Range Bucket Values:

Endpoint	Included Endpoint	Allowed in Actions	Range	Alias	Description
90	✓	✓	>=90	A	
80	✓	✓	[80..90)	B	
70	✓	✓	[70..80)	C	
60	✓	✓	[60..70)	D	
-Infinity	✓	✓	<60	F	

To associate a bucketset with a fact property:

To prepare for creating Decision Tables you can associate a global bucketset with fact properties in the data model. In this way condition cells in a Decision Table **Conditions** area can use the bucketset when you create a Decision Table.

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 **Bucketset** field, from the dropdown list select **Grade Scale**.
6. Click **OK**.

9.3.2 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** navigation tab.
2. In **Ruleset_1**, click the **Add** icon and from the dropdown list select **Create Decision Table**. 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** icon and from the dropdown list select **Condition**.
4. In the Decision Table, double-click **<edit condition>**. Then, in the dropdown 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.

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** icon and from the dropdown 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** checkbox and the **Constant** checkbox.

   This specifies that each rule independently sets the grade.

5. In the Action Editor dialog select the **Always Selected** checkbox.
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.

9.3.3 How to Add Actions in the Decision Table for Grades Sample Application

You can use the Decision Table split operation to create rules for the bucketset associated with the conditions row in the Decision Table. This creates one rule for every bucket.

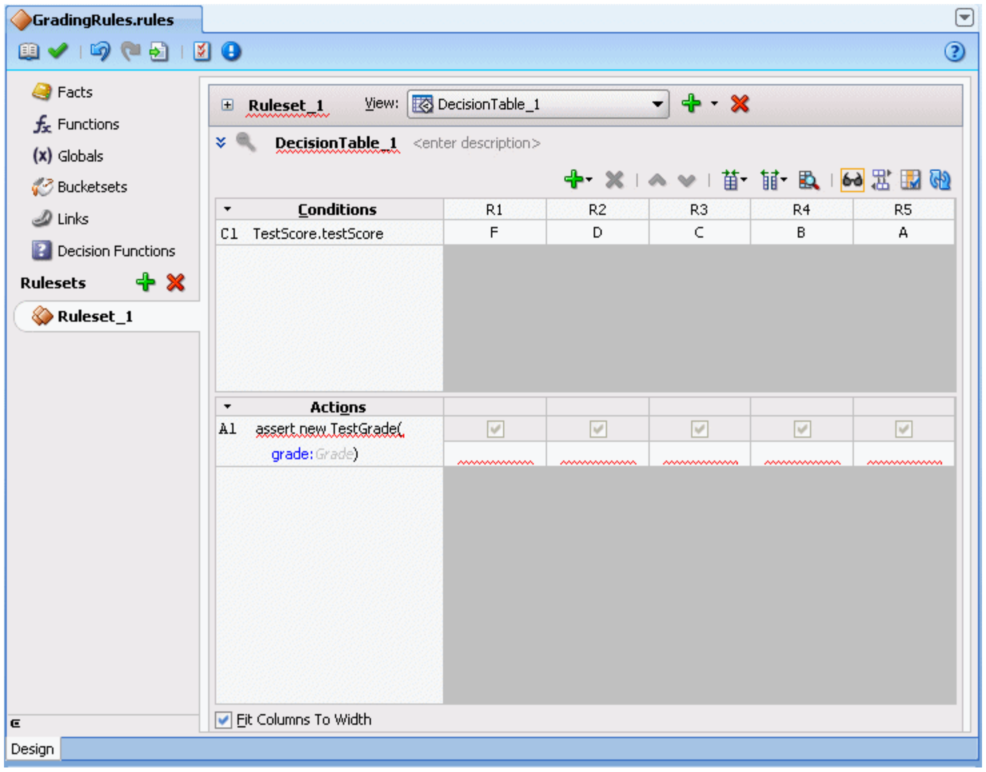
To split the decision table:

1. Select the Decision Table.
2. Click the **Split Table** icon and from the dropdown 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 9–12.

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 9–12 Splitting a Decision Table Using Split Table Operation for Grades



Conditions	R1	R2	R3	R4	R5
C1 TestScore.testScore	F	D	C	B	A

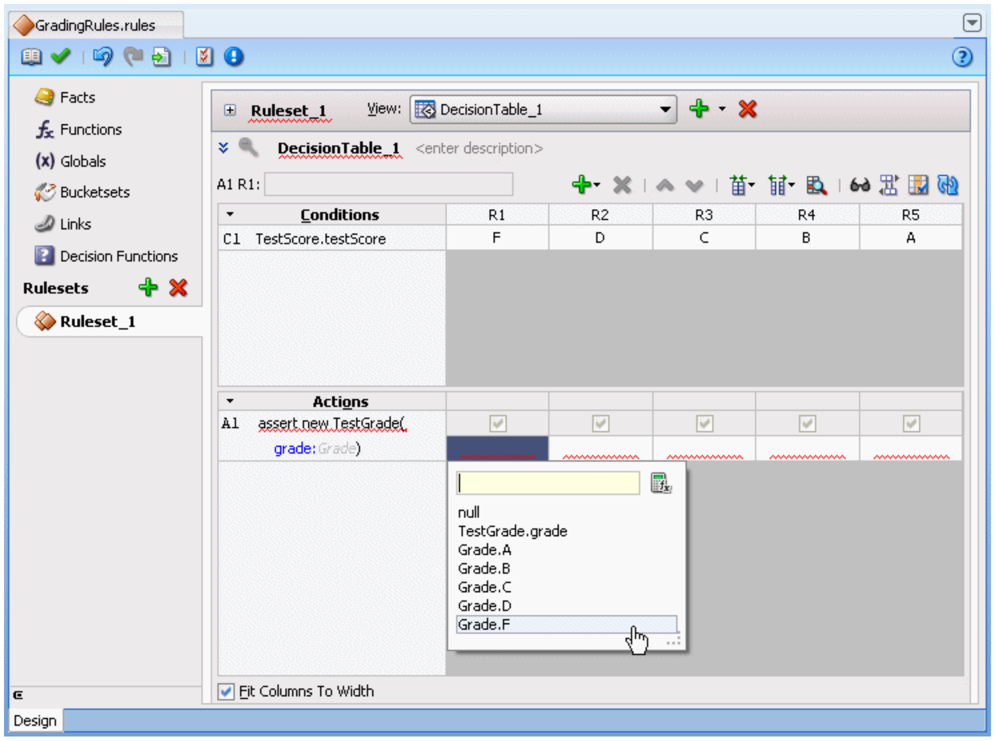
Actions					
A1 assert new TestGrade(grade: Grade)	✓	✓	✓	✓	✓

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 9–13.

Figure 9–13 Adding Action Cell Values to Grades Decision Table



Conditions	R1	R2	R3	R4	R5
C1 TestScore.testScore	F	D	C	B	A

Actions	R1	R2	R3	R4	R5
A1 assert new TestGrade(grade: Grade)	✓	✓	✓	✓	✓

```
null
TestGrade.grade
Grade.A
Grade.B
Grade.C
Grade.D
Grade.F
```

[✓] Fit Columns To Width

2. In the dropdown 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 the buckets for **TestScore**: D, C, B, and A.

9.3.4 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`.

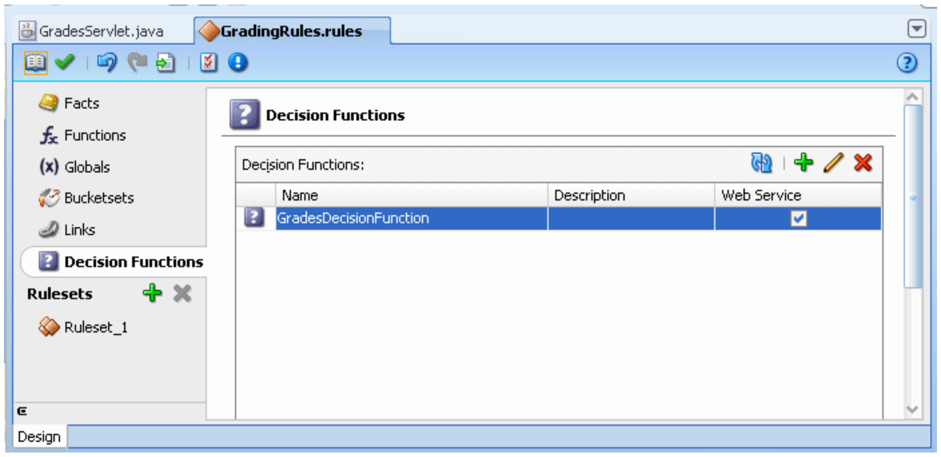
To rename the decision function:

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 9–14.

Figure 9–14 Renaming Decision Function in Rules Designer



Decision Functions

Decision Functions:

Name	Description	Web Service
GradesDecisionFunction		✓

9.4 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 Chapter 7, "Working with Rules SDK Decision Point API".

9.4.1 How to Add a Servlet to the Grades Project

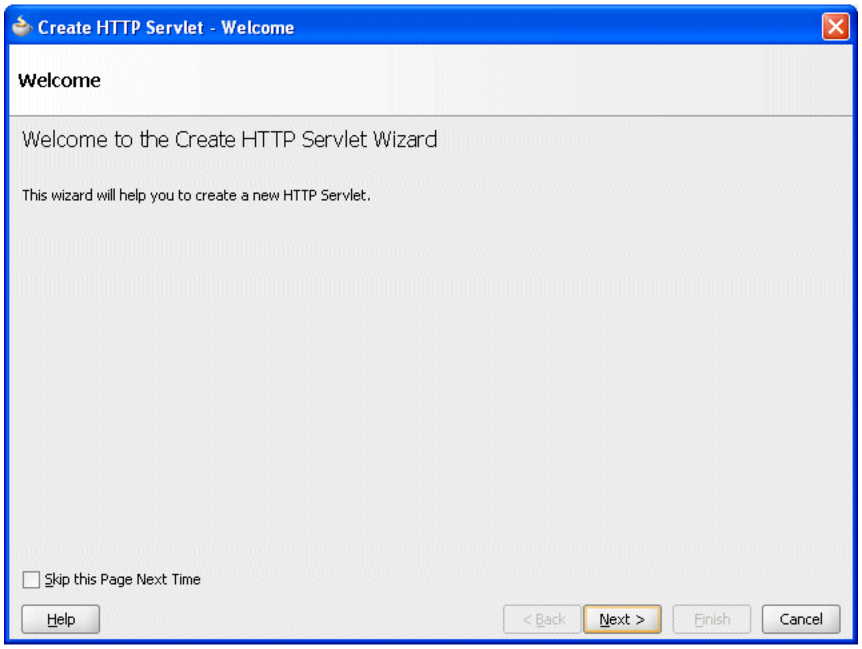
You add a servlet to the grades project using the Create HTTP Servlet wizard.

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**.
6. Click **OK**.

   Oracle JDeveloper displays the Create HTTP Servlet Welcome page, as shown in Figure 9–15.

Figure 9–15 Create HTTP Servlet Wizard - Welcome



Welcome to the Create HTTP Servlet Wizard

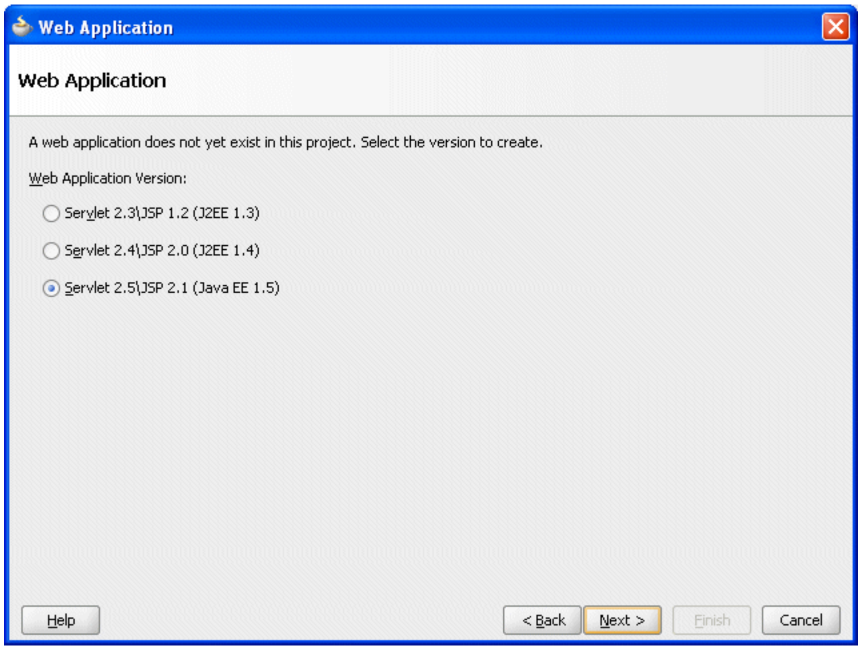
This wizard will help you to create a new HTTP Servlet.

☐ Skip this Page Next Time

7. Click **Next**.

   This displays the Web Application page, as shown in Figure 9–16.

Figure 9–16 Create HTTP Servlet Wizard - Web Application



A web application does not yet exist in this project. Select the version to create.

Web Application Version:

○ Servlet 2.3\JSP 1.2 (J2EE 1.3)

○ Servlet 2.4\JSP 2.0 (J2EE 1.4)

● Servlet 2.5\JSP 2.1 (Java EE 1.5)

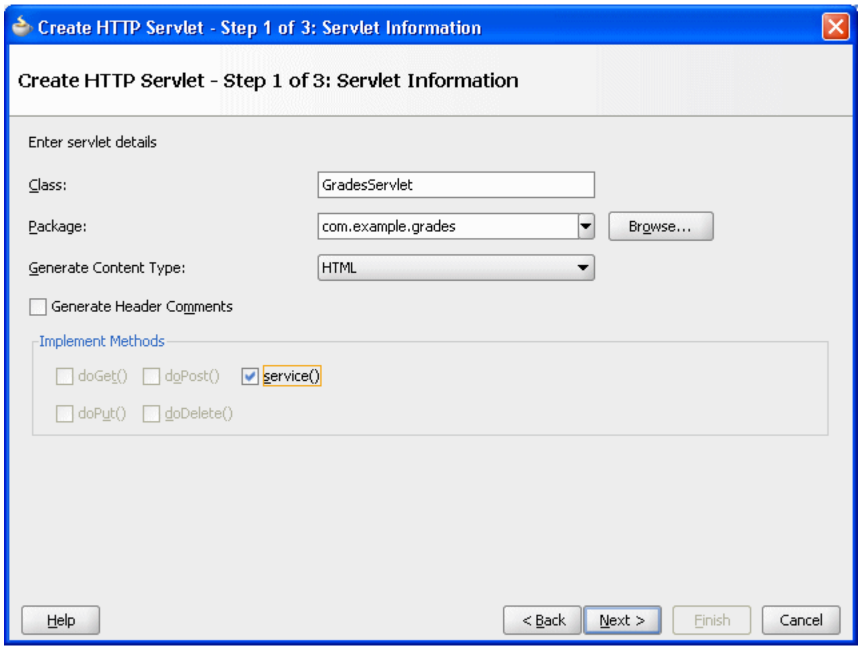
8. Select **Servlet 2.5\JSP 2.1 (Java EE 1.5)** and click **Next**.

   This displays the Create HTTP Servlet - Step 1 of 3: Servlet Information page.

9. Enter values in Create HTTP Servlet - Step 1 of 3: Servlet Information page, as follows, and as shown in Figure 9–17.

- **Class**: `GradesServlet`
- **Package**: `com.example.grades`
- **Generate Content Type**: `HTML`
- **Generate Header Comments**: `unchecked`
- **Implement Methods**: **service()** checked and all other checkboxes unchecked

Figure 9–17 Create HTTP Servlet Wizard - Step 1 of 3: Servlet Information


Create HTTP Servlet - Step 1 of 3: Servlet Information

Create HTTP Servlet - Step 1 of 3: Servlet Information

Enter servlet details

Class: GradesServlet

Package: com.example.grades

Generate Content Type: HTML

☐ Generate Header Comments

Implement Methods

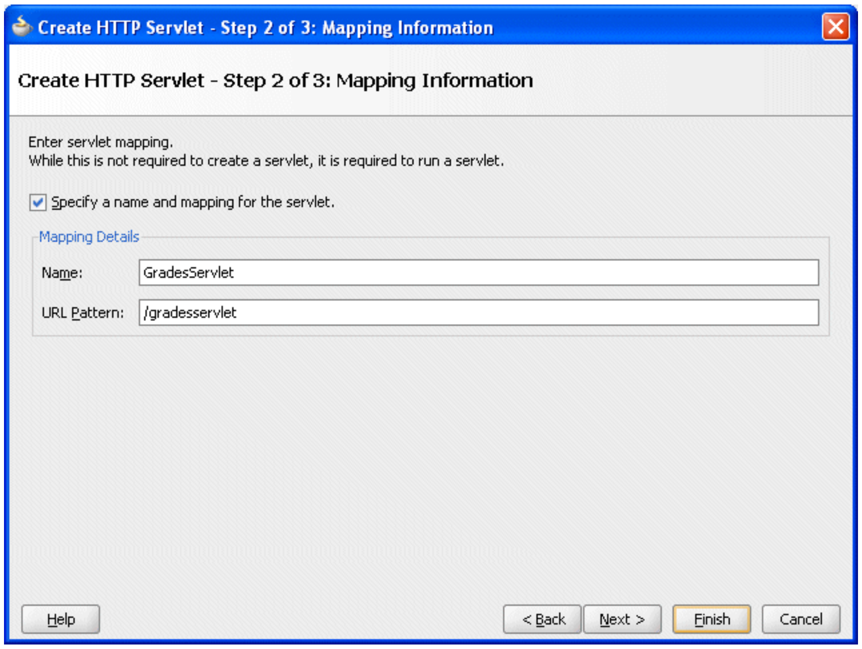
☐ doGet() ☐ doPost() ☑ service()

☐ doPut() ☐ doDelete()

10. Click **Next**.

This displays the Create HTTP Servlet: Step 2 of 3: Mapping Information dialog as shown in Figure 9–18.

Figure 9–18 Create HTTP Servlet Wizard - Step 2 of 3: Mapping Information


Create HTTP Servlet - Step 2 of 3: Mapping Information

Enter servlet mapping.
While this is not required to create a servlet, it is required to run a servlet.

[✓] Specify a name and mapping for the servlet.

Mapping Details

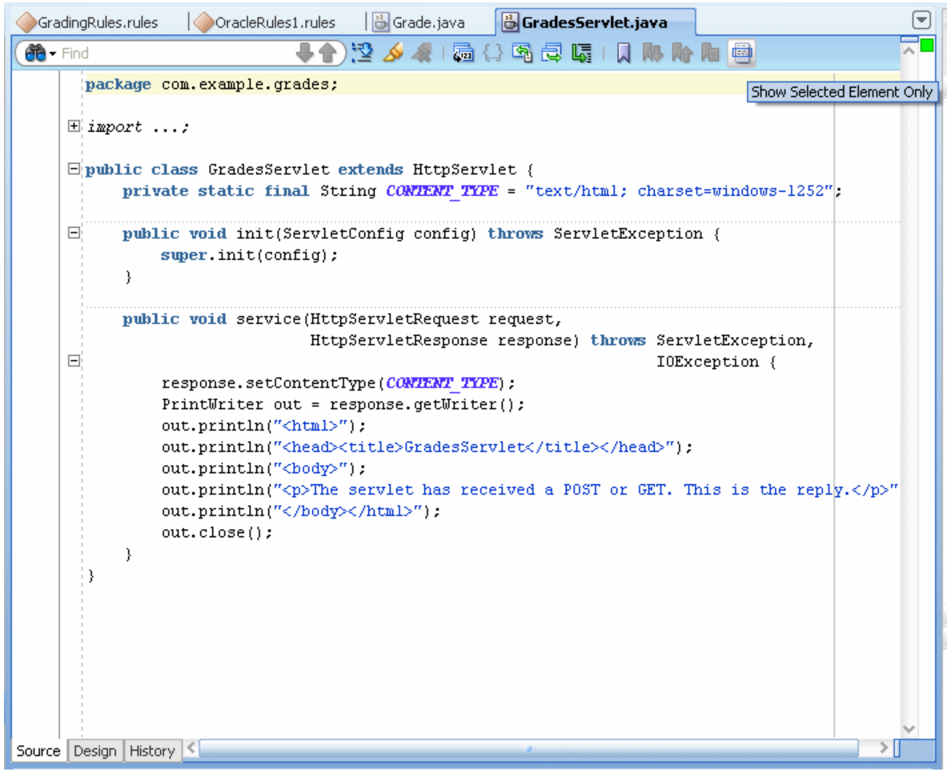
Name: GradesServlet

URL Pattern: /gradesservlet

11. Configure this dialog as follows:
    - **Name**: `GradesServlet`
    - **URL Pattern**: `/gradesservlet`
12. Click **Finish**.

    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 9–19.

Figure 9–19 Generated GradesServlet.java



```
package com.example.grades;

import ...;

public class GradesServlet extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html; charset=windows-1252";

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void service(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException,
                                                             IOException {
        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>GradesServlet</title></head>");
        out.println("<body>");
        out.println("<p>The servlet has received a POST or GET. This is the reply.</p>"
        out.println("</body></html>");
        out.close();
    }
}
```

13. Replace the generated servlet with the source shown in Example 9–2.

Example 9–2 Business Rules Using Servlet for Grades Application

```
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>");
    }
}
```

Example 9–2 includes a Oracle Business Rules Decision Point, that uses an MDS repository to access the dictionary. For more information, see Section 7.5, "What You Need to Know About Using Decision Point in a Production Environment".

When you add the Servlet shown in Example 9–2, 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 Chapter 7, "Working with Rules SDK Decision Point API".
- 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 Chapter 7, "Working with Rules SDK Decision Point API".

9.5 Adding an HTML Test Page for Grades Sample Application

The Grades sample application includes an HTML test page that you use to invoke the servlet you created in Section 9.4, "Adding a Servlet with Rules SDK Calls for Grades Sample Application".

9.5.1 How to Add an HTML Test Page to the Grades Project

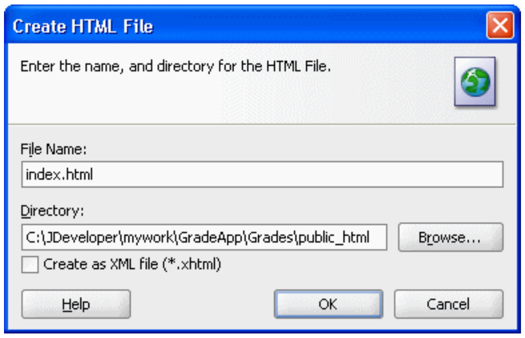
To add an HTML page to the servlet you use the Create HTML File wizard.

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**.
6. In the New Gallery click **OK**.

   Oracle JDeveloper displays the Create HTML File dialog.
7. Configure this dialog as follows and as shown in Figure 9–20:
   - **File Name**: `index.html`
   - **Directory**: `C:\JDeveloper\mywork\GradeApp\Grades\public_html`

Figure 9–20 Create HTML File Dialog



Create HTML File

Enter the name, and directory for the HTML File.

File Name: index.html

Directory: C:\JDeveloper\mywork\GradeApp\Grades\public_html

Create as XML file (*.xhtml)

8. Click **OK**.

   JDeveloper adds `index.html` to the Web Content folder and opens the editor.
9. In the editor for `index.html`, select the **Source** tab.
10. Copy and paste the HTML code from Example 9–3 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 9–18.

Example 9–3 HTML Test Page

```
<!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>
```

11. Select **Save All** from the **File** main menu to save your work.

9.6 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.

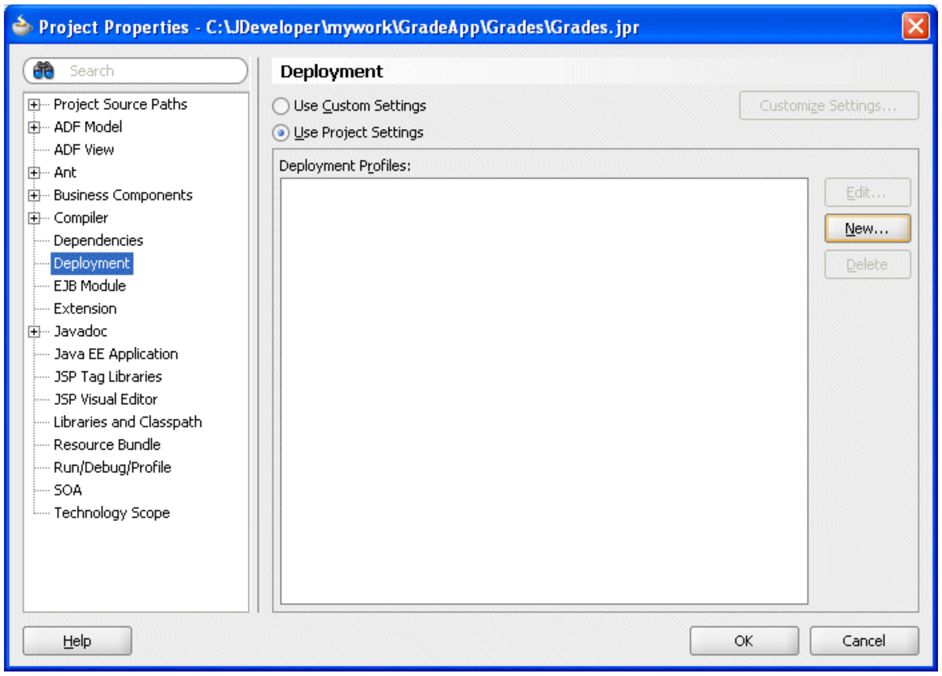
9.6.1 How to Create the WAR File for the Grades Sample Application

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 9–21.

Figure 9–21 Project Properties - Deployment



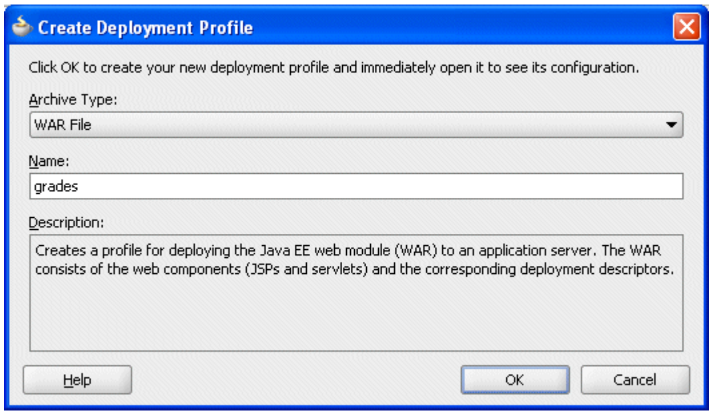
4. In the Project Properties dialog, click **New...**.

   This displays the Create Deployment Profile dialog.

5. In the Create Deployment Profile dialog, in the **Archive Type** dropdown list, select **WAR File**.

6. In the Create Deployment Profile dialog, in the **Name** field enter `grades`, as shown in Figure 9–22. Note the **Name** value uses the package value that you specified in the `form` element `action` attribute in Example 9–3.

Figure 9–22 Create Deployment Profile Dialog for WAR File



Click OK to create your new deployment profile and immediately open it to see its configuration.

Archive Type: WAR File

Name: grades

Description: Creates a profile for deploying the Java EE web module (WAR) to an application server. The WAR consists of the web components (JSPs and servlets) and the corresponding deployment descriptors.

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 9–23:

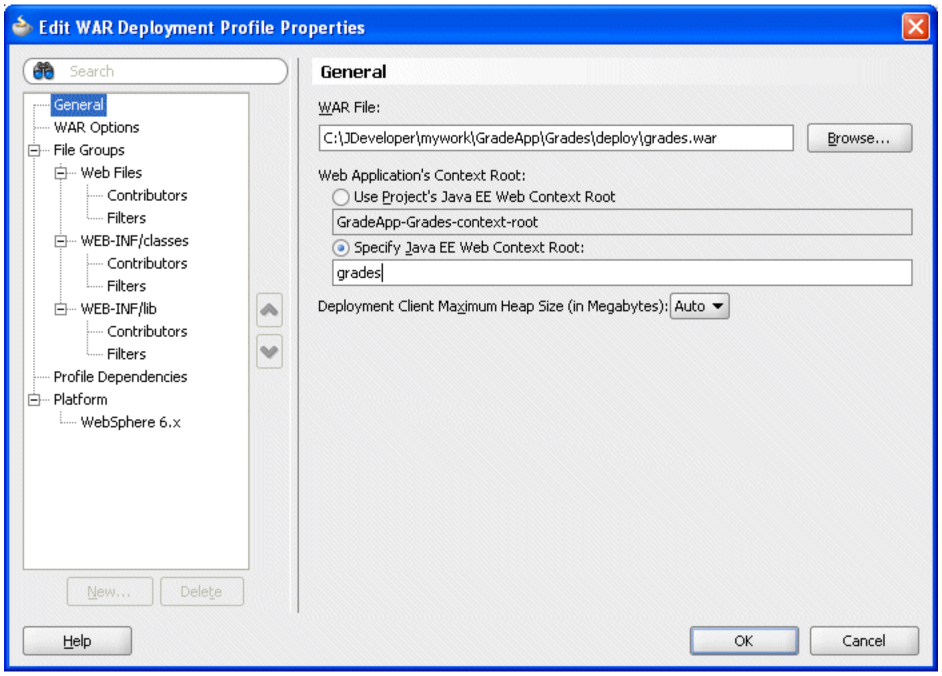
a. Set the **WAR File**: `C:\JDeveloper\mywork\GradeApp\Grades\deploy\grades.war`

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)**: dropdown list select **Auto**

Figure 9–23 Edit WAR Deployment Properties - General



General

WAR File: C:\JDeveloper\mywork\GradeApp\Grades\deploy\grades.war

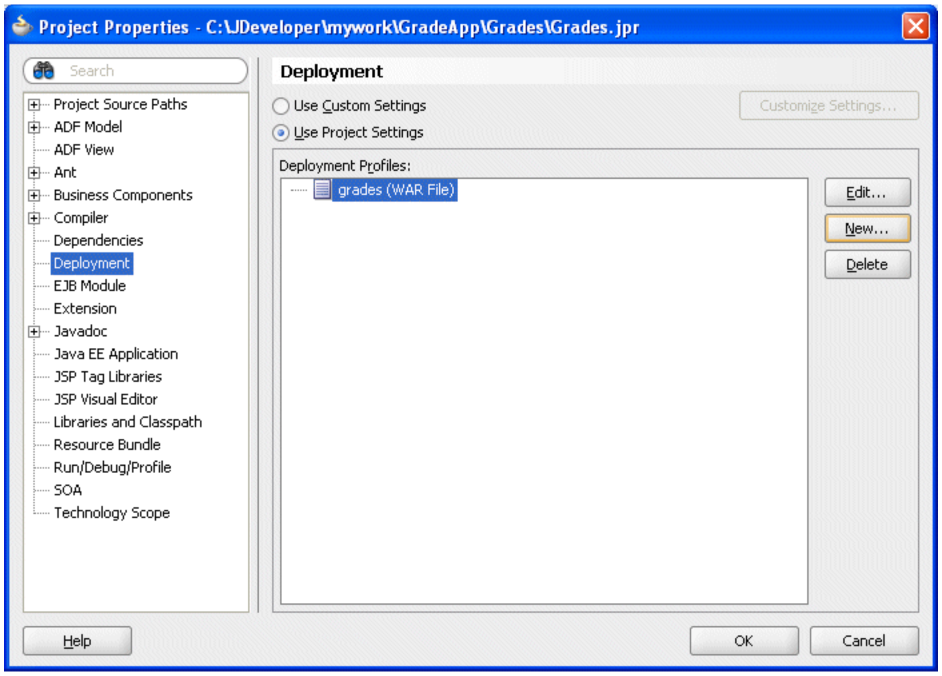
Web Application's Context Root:
( ) Use Project's Java EE Web Context Root
GradeApp-Grades-context-root
(●) Specify Java EE Web Context Root:
grades

Deployment Client Maximum Heap Size (in Megabytes): Auto

9. In the Edit WAR Deployment Profile Properties dialog, click **OK**.

   JDeveloper creates a deployment profile named `grades (WAR File)` as shown in Figure 9–24.

Figure 9–24 Project Properties - Deployment Profile Created



10. In the Project Properties dialog, click **OK**.

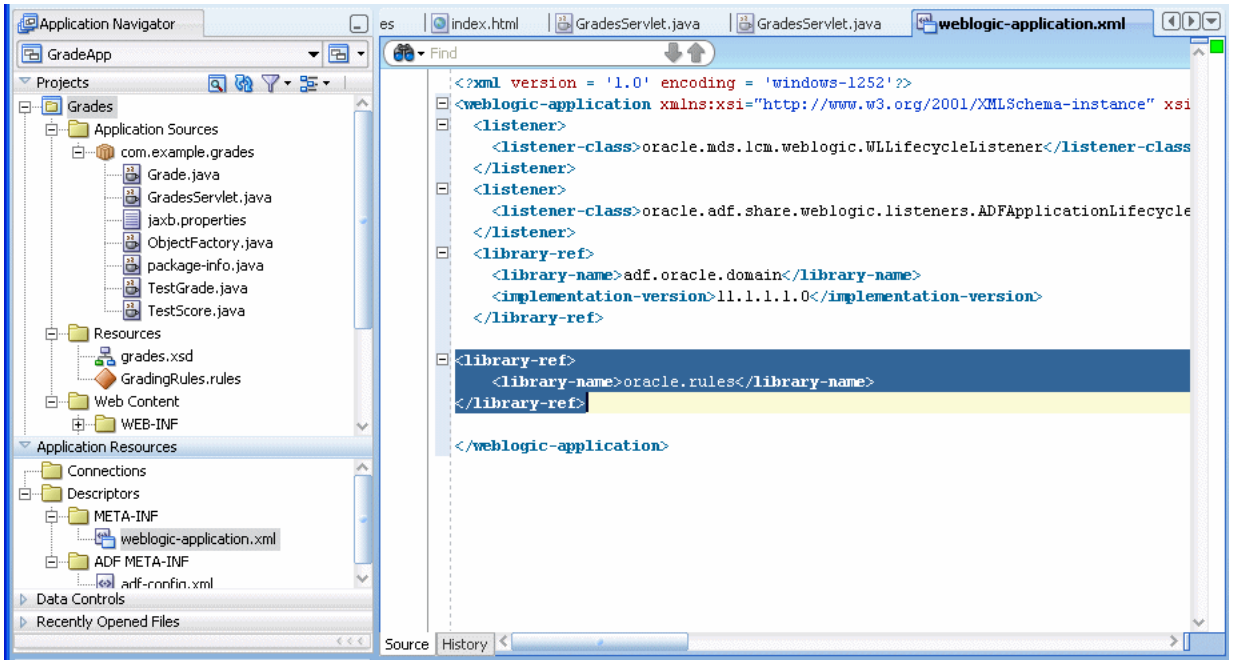
9.6.2 How to Add the Rules Library to the Grades Sample Application

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 9–25.

```
<library-ref>
    <library-name>oracle.rules</library-name>
</library-ref>
```

Figure 9–25 Adding Oracle Rules Library Reference to WebLogic Descriptor



```
<?xml version = '1.0' encoding = 'windows-1252'?>
<weblogic-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi
  <listener>
    <listener-class>oracle.mds.lcm.weblogic.WLLifecycleListener</listener-class
  </listener>
  <listener>
    <listener-class>oracle.adf.share.weblogic.listeners.ADFApplicationLifecycle
  </listener>
  <library-ref>
    <library-name>adf.oracle.domain</library-name>
    <implementation-version>11.1.1.1.0</implementation-version>
  </library-ref>

<library-ref>
    <library-name>oracle.rules</library-name>
</library-ref>

</weblogic-application>
```

4. Save the `weblogic-application.xml` file.

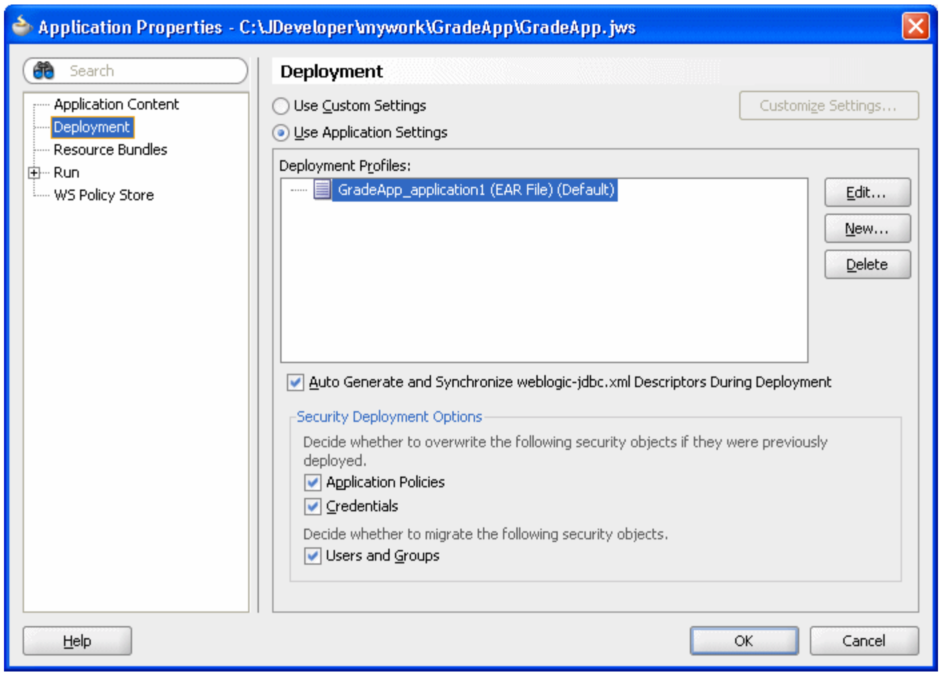
9.6.3 How to Add the MDS Deployment File to the Grades Sample Application

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 9–26.

Figure 9–26 Application Properties - Deployment


Application Properties - C:\JDeveloper\mywork\GradeApp\GradeApp.jws

Deployment

Use Custom Settings
Use Application Settings [✓]

Deployment Profiles:
GradeApp_application1 (EAR File) (Default)

[✓] Auto Generate and Synchronize weblogic-jdbc.xml Descriptors During Deployment

Security Deployment Options

Decide whether to overwrite the following security objects if they were previously deployed.
[✓] Application Policies
[✓] Credentials

Decide whether to migrate the following security objects.
[✓] Users and Groups

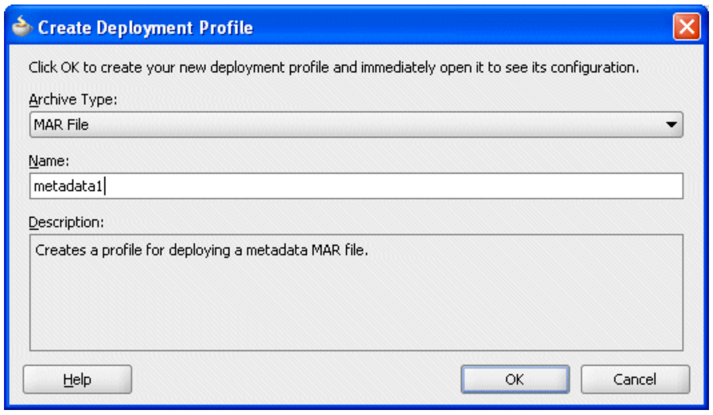
4. In the Application Properties dialog, click **New...**.

   This displays the Create Deployment Profile dialog.

5. Configure this dialog as follows, as shown in Figure 9–27:

   - **Archive Type**: `MAR File`
   - **Name**: `metadata1`

Figure 9–27 Create Deployment Profile Dialog for MAR File


Create Deployment Profile

Click OK to create your new deployment profile and immediately open it to see its configuration.

Archive Type: MAR File

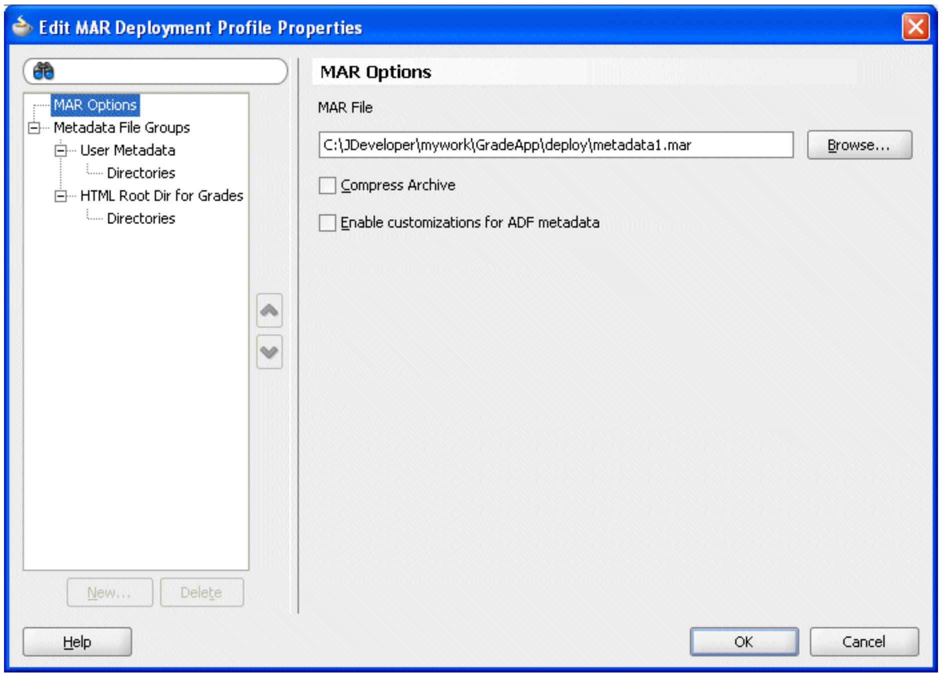
Name: metadata1

Description: Creates a profile for deploying a metadata MAR file.

6. Click **OK**.

   This displays the Edit MAR Deployment Properties dialog as shown in Figure 9–28.

Figure 9–28 Edit MAR Deployment Profile Properties - MAR Options



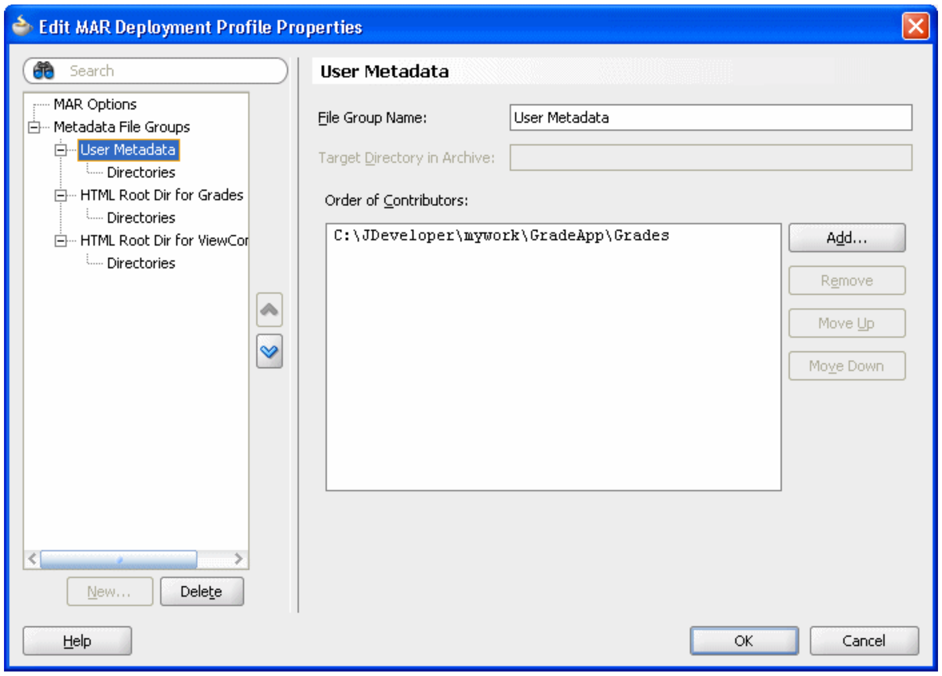
MAR File: C:\JDeveloper\mywork\GradeApp\deploy\metadata1.mar

7. Expand the **Metadata File Groups** item and select the **User Metadata** item.
8. Click **Add....**

   This displays the Add Contributor dialog.
9. 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**.
10. In the Add Contributor dialog, click **OK** to close the dialog. This displays the Edit MAR Deployment Properties dialog as shown in Figure 9–29

Figure 9–29 Edit MAR Deployment Profile Properties - User Metadata



User Metadata

File Group Name: User Metadata

Target Directory in Archive:

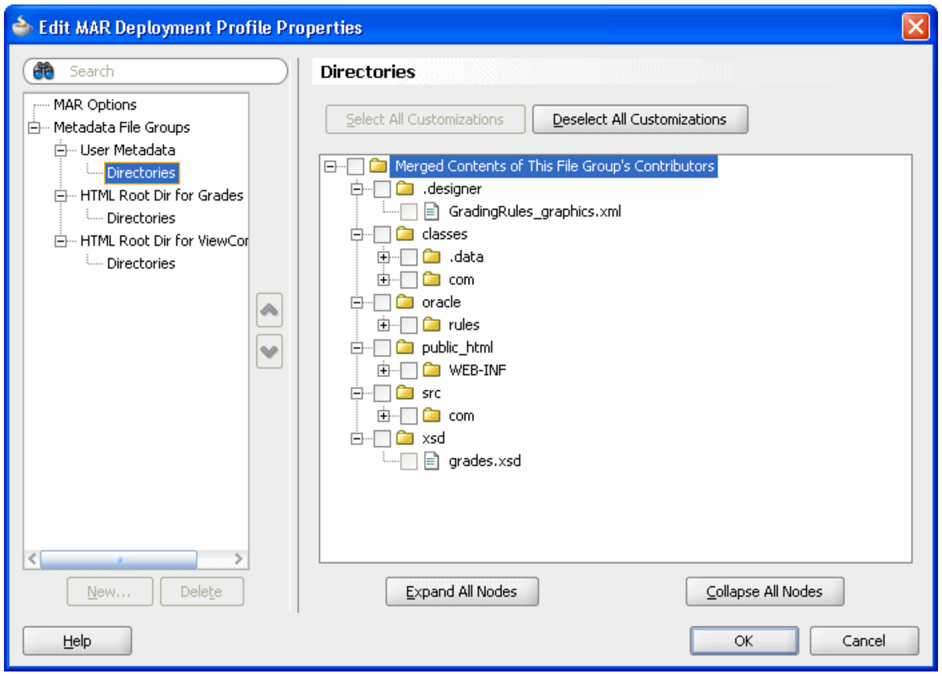
Order of Contributors:

```
C:\JDeveloper\mywork\GradeApp\Grades
```

11. 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 9–30.

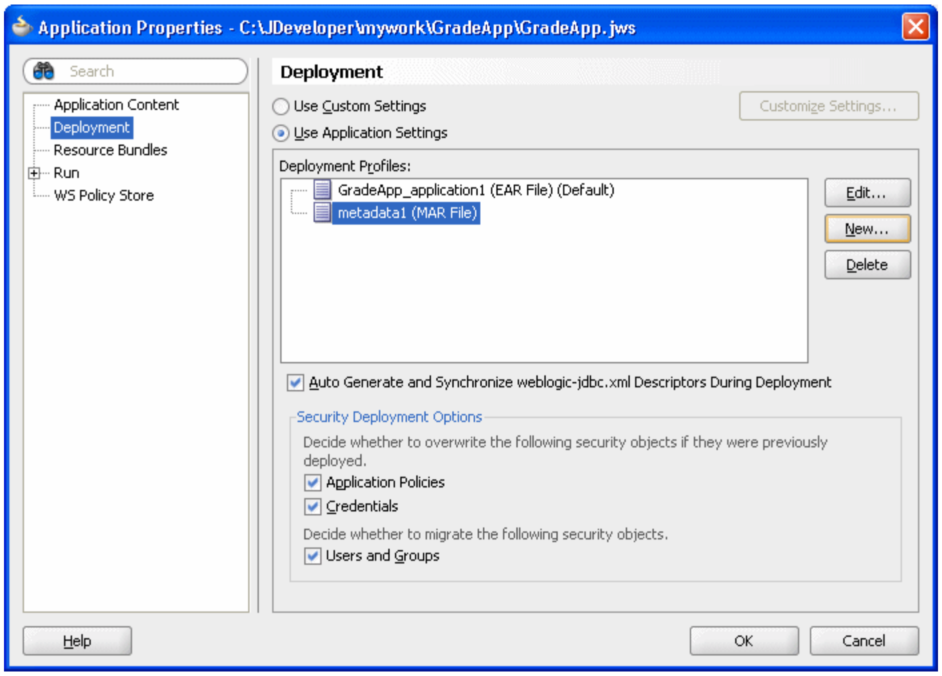
Figure 9–30 Edit MAR Deployment Profile Properties - Directories



12. Select the **oracle** directory checkbox. This selects the **GradingRules.rules** dictionary to be included in the MAR.
13. Click **OK**.

JDeveloper creates an application deployment profile named `metadata1 (MAR File)` as shown in Figure 9–31.

Figure 9–31 Application Properties - Deployment - MAR



Use Custom Settings [✓ Use Application Settings]

Deployment Profiles:
- GradeApp_application1 (EAR File) (Default)
- metadata1 (MAR File)

✓ Auto Generate and Synchronize weblogic-jdbc.xml Descriptors During Deployment

Security Deployment Options

Decide whether to overwrite the following security objects if they were previously deployed.

✓ Application Policies

✓ Credentials

Decide whether to migrate the following security objects.

✓ Users and Groups

14. In the Application Properties dialog, click **OK**.

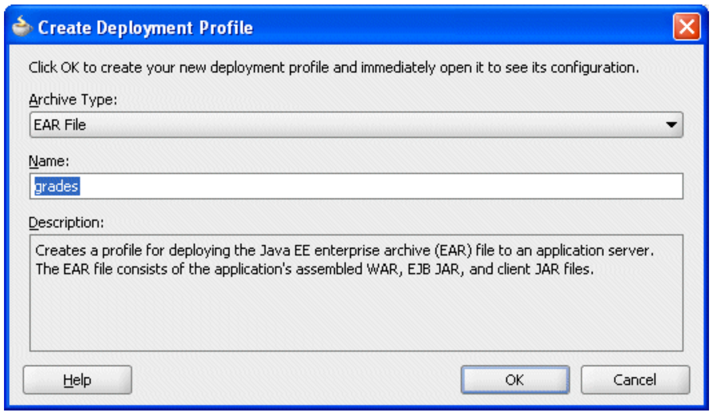
9.6.4 How to Add the EAR File to the Grades Sample Application

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 9–32.
   - **Archive Type**: `EAR`
   - **Name**: `grades`

Figure 9–32 Create Deployment Profile Dialog for EAR File



Click OK to create your new deployment profile and immediately open it to see its configuration.

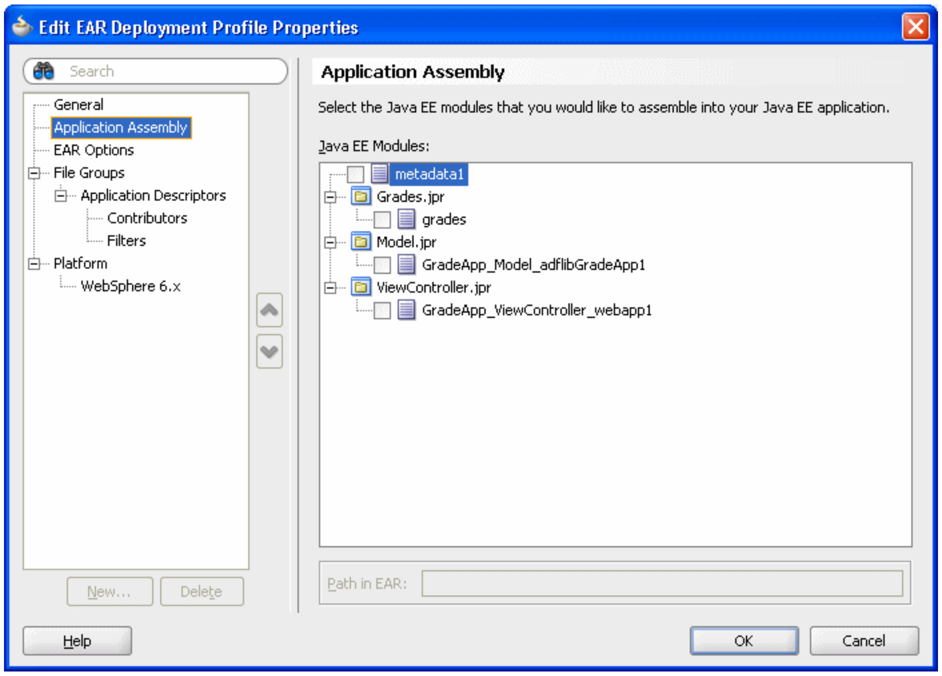
Archive Type: EAR File

Name: grades

Description: Creates a profile for deploying the Java EE enterprise archive (EAR) file to an application server. The EAR file consists of the application's assembled WAR, EJB JAR, and client JAR files.

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 9–33.

Figure 9–33 Edit EAR Deployment Profile Properties - Application Assembly



Application Assembly

Select the Java EE modules that you would like to assemble into your Java EE application.

Java EE Modules:

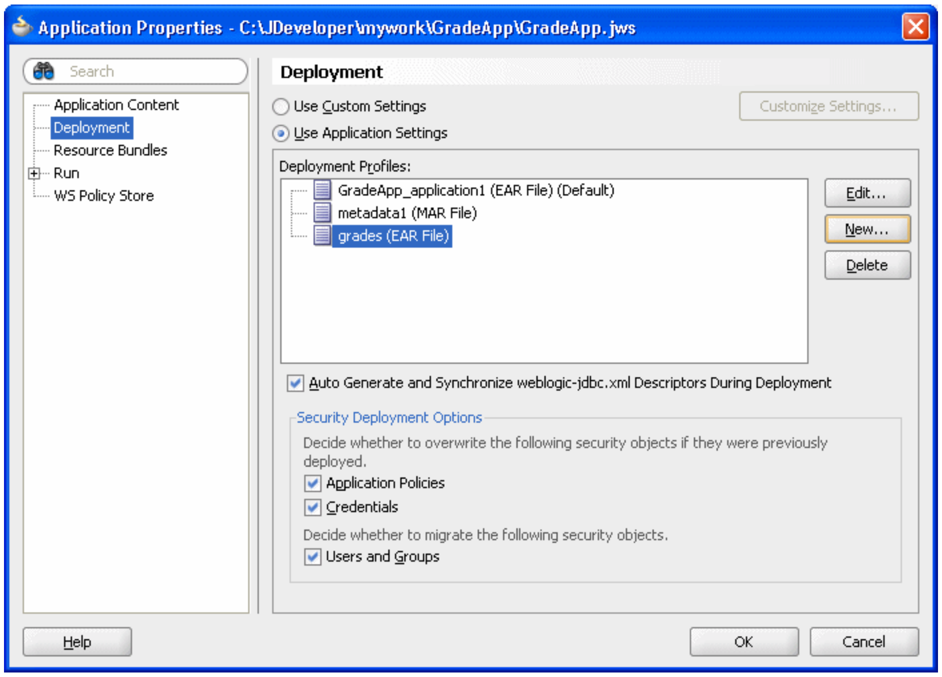
- metadata1
- Grades.jpr
  - grades
- Model.jpr
  - GradeApp_Model_adflibGradeApp1
- ViewController.jpr
  - GradeApp_ViewController_webapp1

Path in EAR:

7. Configure this dialog as follows:
   - Select the **metadata1** checkbox.
   - Expand the **Grades.jpr** item and select the **grades** checkbox.
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 9–34.

Figure 9–34 Application Properties - Deployment - EAR


Application Properties - C:\JDeveloper\mywork\GradeApp\GradeApp.jws
Search
Application Content
Deployment
Resource Bundles
Run
WS Policy Store
Deployment
Use Custom Settings
Customize Settings...
Use Application Settings
Deployment Profiles:
GradeApp_application1 (EAR File) (Default)
metadata1 (MAR File)
grades (EAR File)
Edit...
New...
Delete
Auto Generate and Synchronize weblogic-jdbc.xml Descriptors During Deployment
Security Deployment Options
Decide whether to overwrite the following security objects if they were previously deployed.
Application Policies
Credentials
Decide whether to migrate the following security objects.
Users and Groups
Help
OK
Cancel

9. Click **OK** to close the Application Properties dialog.
10. Select **Save All** from the **File** main menu to save your work.

9.7 Deploying and Running the Grades Sample Application

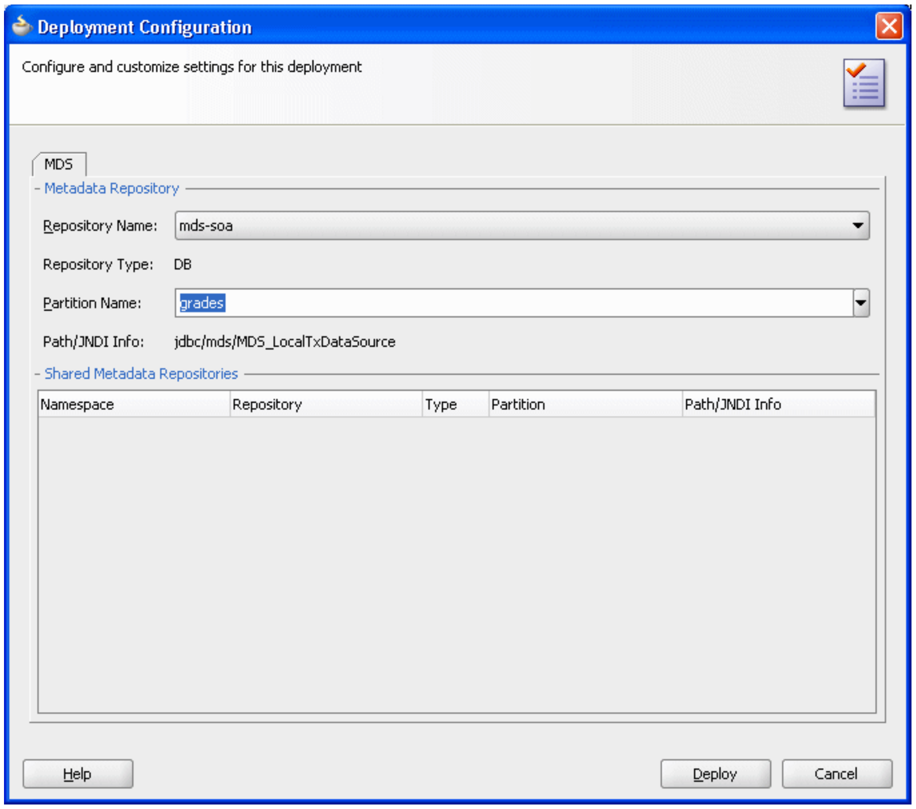
You can now deploy and run the grades sample application on Oracle WebLogic Server.

9.7.1 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 9–32:
   - In the **Repository Name** field, from the dropdown list, select: **mds-soa**
   - In the **Partition Name** field, enter **grades**

Figure 9–35 Deployment Configuration Dialog for MDS with Repository and Partition


Deployment Configuration
Configure and customize settings for this deployment
MDS
Metadata Repository
Shared Metadata Repositories
Help
Deploy
Cancel

Repository Name: mds-soa
Repository Type: DB
Partition Name: grades
Path/JNDI Info: jdbc/mds/MDS_LocalTxDataSource

Namespace	Repository	Type	Partition	Path/JNDI Info

5. In the Deployment Configuration dialog, click **Deploy**.

9.7.2 How to Run the Grades Sample Application

After you deploy the grades sample application, you can run the application.

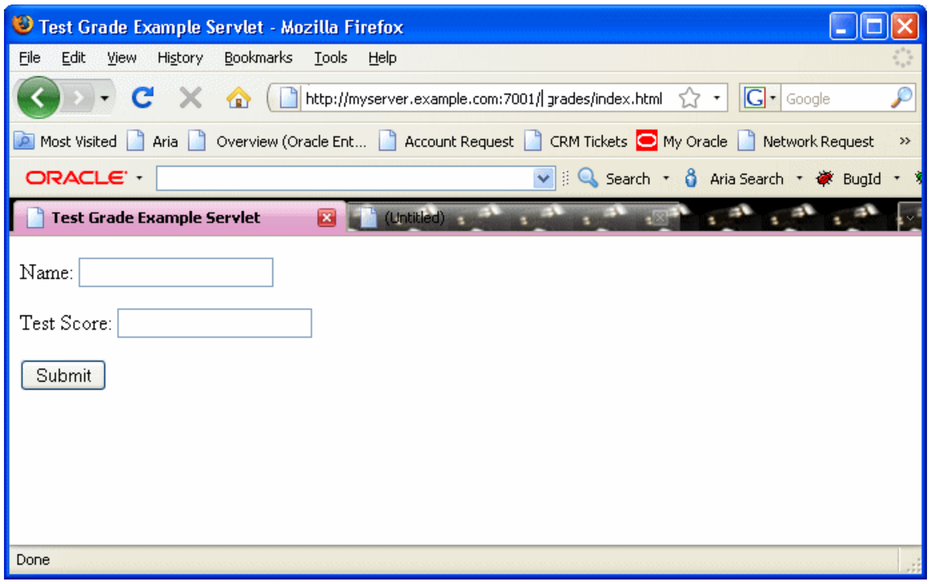
To run the grades sample application:

1. Point a web browser at,

   http://*yourServerName*:*port*/grades/

   This displays the test servlet as shown in Figure 9–36.

Figure 9–36 Grades Sample Application Servlet


Test Grade Example Servlet - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://myserver.example.com:7001/grades/index.html
Most Visited Aria Overview (Oracle Ent... Account Request CRM Tickets My Oracle Network Request
Search Aria Search BugId
Test Grade Example Servlet
(Untitled)
Submit
Done

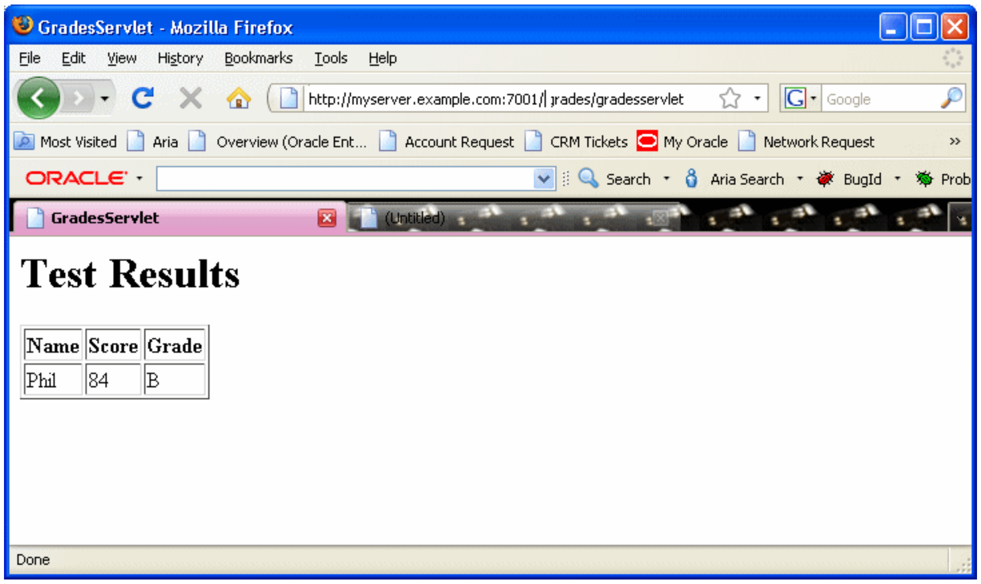
Name:

Test Score:

2. Enter a name and test score and click **Submit**. This returns results as shown in Figure 9–37.

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 9–37 Grades Sample Application Servlet with Results


GradesServlet - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://myserver.example.com:7001/grades/gradesservlet
Most Visited Aria Overview (Oracle Ent... Account Request CRM Tickets My Oracle Network Request
Search Aria Search BugId Prob
GradesServlet
(Untitled)
Done

Test Results

Name	Score	Grade
Phil	84	B

10 Working with Oracle Business Rules and ADF Business Components

Oracle Business Rules allows you to use Oracle ADF Business Components view objects as facts. By using ADF Business Components facts you can assert trees of view object graphs representing the business objects upon which rules should be based, and let Oracle Business Rules handle the complexities of managing the relationships between the various related view objects in the main view object's tree.

This chapter includes the following sections:

10.1 Introduction to Using Business Rules with ADF Business Components

The ADF Business Components rule development process can be summarized as follows:

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.

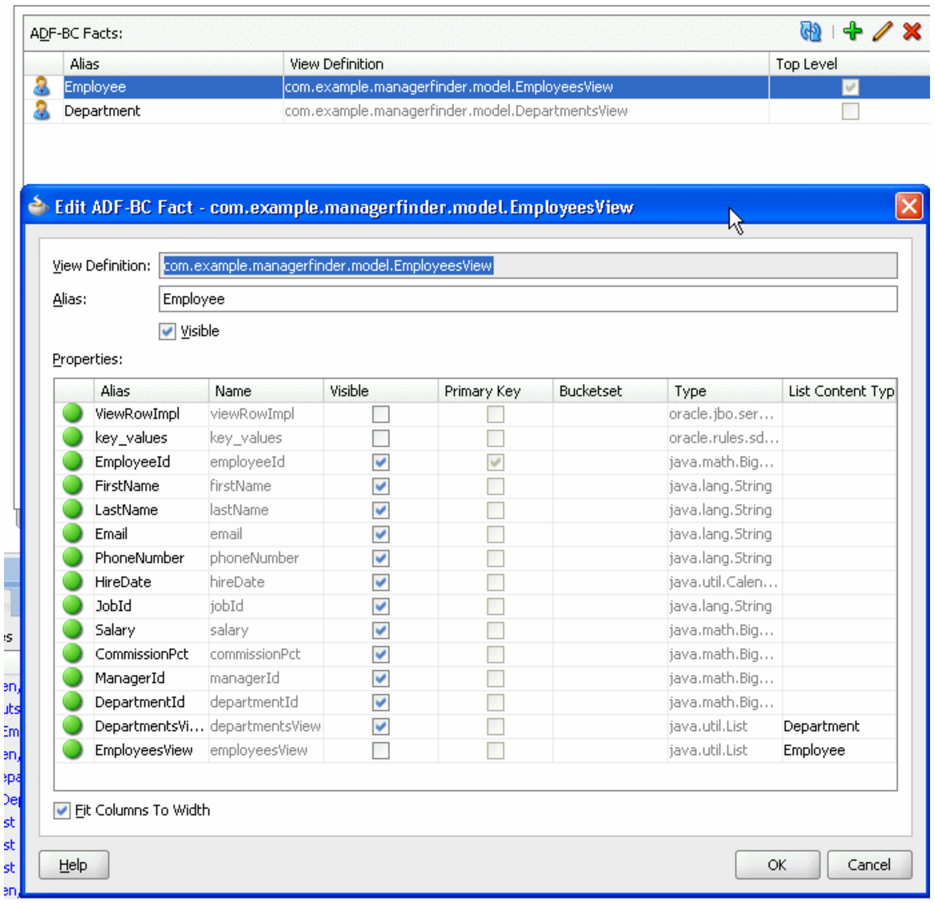
10.1.1 Understanding Oracle Business Rules ADF Business Components Fact Types

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, as shown in Figure 10–1.

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.

Figure 10–1 ADF Business Components Sample Fact Type



ADF-BC Facts:

Alias	View Definition	Top Level
Employee	com.example.managerfinder.model.EmployeesView	✓
Department	com.example.managerfinder.model.DepartmentsView	☐

Edit ADF-BC Fact - com.example.managerfinder.model.EmployeesView

View Definition:	com.example.managerfinder.model.EmployeesView
Alias:	Employee
	✓ Visible

Properties:

Alias	Name	Visible	Primary Key	Bucketset	Type	List Content Typ
ViewRowImpl	viewRowImpl	☐	☐		oracle.jbo.ser...	
key_values	key_values	☐	☐		oracle.rules.sd...	
EmployeeId	employeeId	✓	✓		java.math.Big...	
FirstName	firstName	✓	☐		java.lang.String	
LastName	lastName	✓	☐		java.lang.String	
Email	email	✓	☐		java.lang.String	
PhoneNumber	phoneNumber	✓	☐		java.lang.String	
HireDate	hireDate	✓	☐		java.util.Calen...	
JobId	jobId	✓	☐		java.lang.String	
Salary	salary	✓	☐		java.math.Big...	
CommissionPct	commissionPct	✓	☐		java.math.Big...	
ManagerId	managerId	✓	☐		java.math.Big...	
DepartmentId	departmentId	✓	☐		java.math.Big...	
DepartmentsVi...	departmentsView	✓	☐		java.util.List	Department
EmployeesView	employeesView	☐	☐		java.util.List	Employee

✓ Fit Columns To Width

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 *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*.

10.1.2 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).

Example 10–1 shows a sample `ActionType` implementation.

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;
    }
}
```

In Example 10–1, there is an `oracle.rules.sdk2.decisionpoint.DecisionPointInstance` as a parameter to the `exec` method. 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.

Using Rules Designer you can select parameters appropriate for the `ActionType` you are configuring.

10.2 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 Chapter 6, "Working with Decision Functions".

10.2.1 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 Section 7.3.1, "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 Section 10.2.1.3, "Calling the Invoke Method for an ADF Business Components Rule"

4. Set the transaction to be used by the `DecisionPointInstance`.

   For more information, see Section 10.2.1.1, "Setting the Decision Point Transaction".

5. Set any runtime properties the consequent application actions may expect.

   For more information, see Section 10.2.1.2, "Setting Runtime Properties".

6. Call the `DecisionPointInstance` method `invoke`.

   For more information, see:

   - Section 10.2.1.3, "Calling the Invoke Method for an ADF Business Components Rule"
   - Section 10.2.1.4, "What You Need to Know About Decision Point Invocation"

10.2.1.1 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.

10.2.1.2 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.

10.2.1.3 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. Section 10.3.9.3, "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.

Example 10–2 shows how to invoke a Decision Point with a `ViewObject` instance using the `setInputs` method. For the complete example, see Example 10–5.

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);
        }
    }
```

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);
        }
    }
```

10.2.1.4 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.

10.2.2 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 Chapter 6, "Working with Decision Functions".

To define a decision function using the Java Decision Point interface:

1. Double-click the decision function icon to the left of the decision function item or select this item and click the **Edit** icon. The Edit Decision Function dialog appears.
2. In the Edit Decision Function dialog, configure the decision function:
   - **Input Fact Types**: 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 unselected. 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 Section 10.2.1, "How to Call a Decision Point with ADF Business Components Facts".

- **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 ADF Business Components fact types, **Output Fact Types** should be left empty. The view object is updated using an `ActionType`. For more information, see Section 10.1.2, "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.

10.2.3 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". 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).

10.3 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 is available online at

`http://www.oracle.com/technology/sample_code/products/rules`

For SOA samples online visit

`http://www.oracle.com/technology/sample_code/products/soa`

10.3.1 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`.

   d. In the **Application Template** field, select **Fusion Web Application (ADF)**.

4. Click **Finish**.

10.3.2 How to Add the Chapter10 Generic Project

You need to add a new project named **Chapter10**.

Add a new project:

1. In the **Chapter10** application, select the Application Menu.
2. In the Application Menu dropdown list, select **New Project...**.
3. In the New Gallery, in the **Items** area select **Generic Project**.
4. Click **OK**.
5. On the Name your project page, in the **Project Name** field enter `Chapter10`.
6. Click **Finish**.

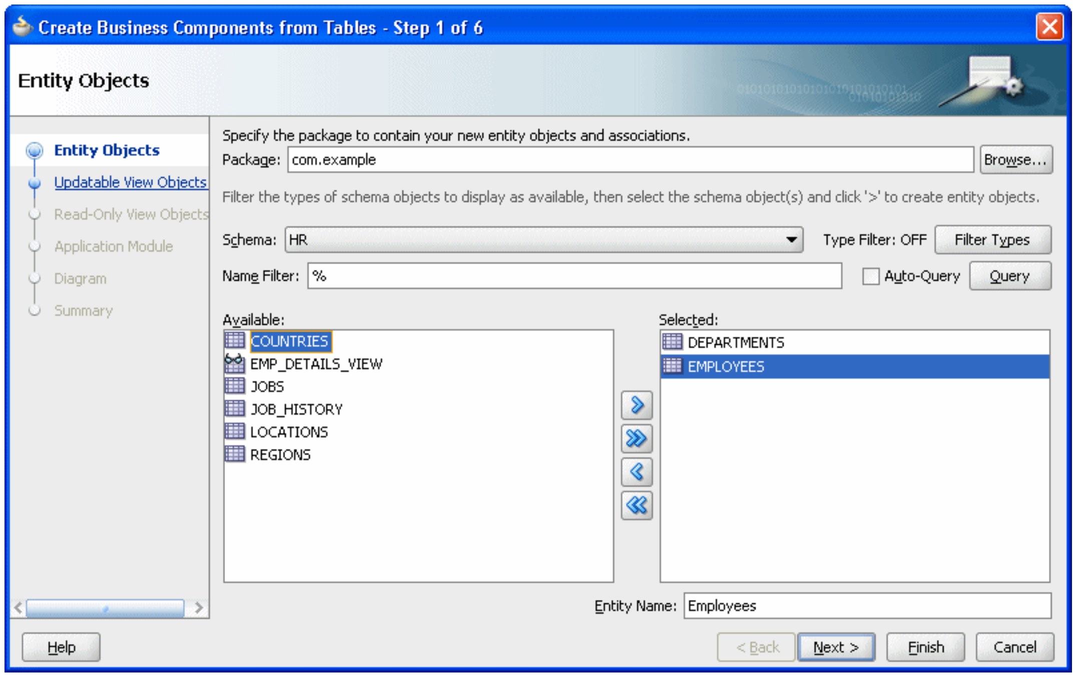
10.3.3 How to Create ADF Business Components Application for Business Rules

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**.
5. Click **OK**.
6. In the Initialize Business Components Project dialog, enter the required connection information to add a connection.
7. Click **OK**. This displays the Create Business Components from Tables wizard.
8. 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**, as shown in Figure 10–2.

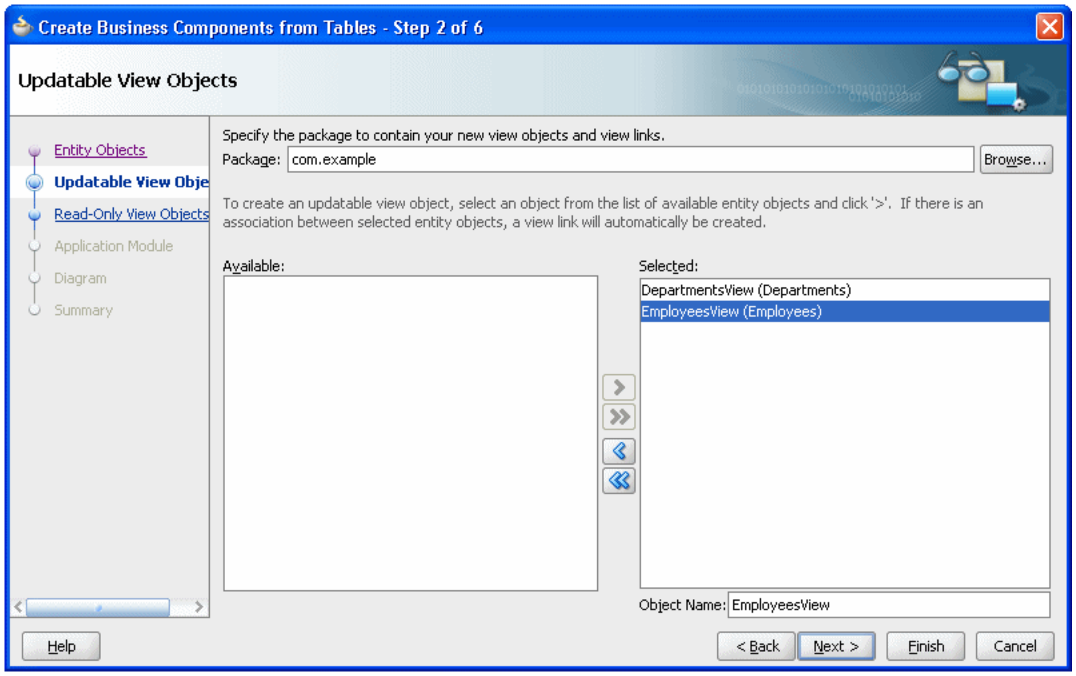
Figure 10–2 Selecting Entity Objects for Sample Application



Specify the package to contain your new entity objects and associations.
Package: com.example
Filter the types of schema objects to display as available, then select the schema object(s) and click '>' to create entity objects.
Schema: HR
Type Filter: OFF
Name Filter: %
Auto-Query
Available: COUNTRIES, EMP_DETAILS_VIEW, JOBS, JOB_HISTORY, LOCATIONS, REGIONS
Selected: DEPARTMENTS, EMPLOYEES
Entity Name: Employees

9. Click **Next**. This displays the Updatable View Objects page.
10. In the Updatable View Objects page select **Departments** and **Employees**, as shown in Figure 10–3.

Figure 10–3 Adding Updatable View Objects for Sample Application



Specify the package to contain your new view objects and view links.
Package: com.example
To create an updatable view object, select an object from the list of available entity objects and click '>'. If there is an association between selected entity objects, a view link will automatically be created.
Available:
Selected: DepartmentsView (Departments), EmployeesView (Employees)
Object Name: EmployeesView

11. Click **Next**. This displays the Read-Only View Objects page.
12. Click **Next**. This displays the Application Module page.

13. Click **Finish**.

10.3.4 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:

1. In the Application Navigator, double-click **EmployeesView**.
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**.

To set tuning options for DepartmentsView:

1. In the Application Navigator, double-click DepartmentsView.
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**.

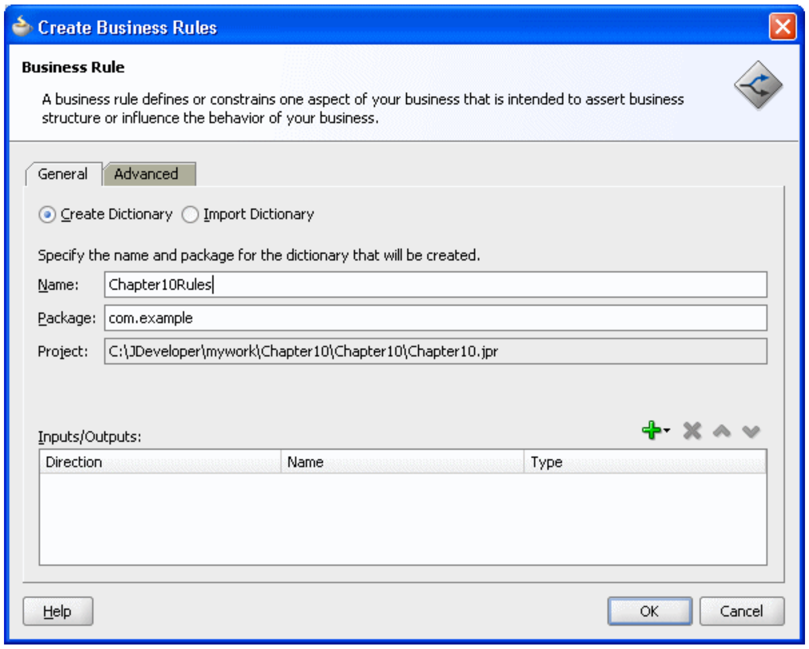
10.3.5 How to Create a Dictionary for Oracle Business Rules

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 dropdown 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, as shown in Figure 10–4:
   - For example, in the **Name** field enter `Chapter10Rules`.
   - For example, in the **Package** field enter `com.example`.

Figure 10–4 Create Business Rules for Chapter10Rules Dictionary


Create Business Rules

Business Rule

A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business.

General | Advanced

Create Dictionary [✓] | Import Dictionary

Specify the name and package for the dictionary that will be created.

Name: Chapter10Rules

Package: com.example

Project: C:\JDeveloper\mywork\Chapter10\Chapter10\Chapter10.jpr

Inputs/Outputs:

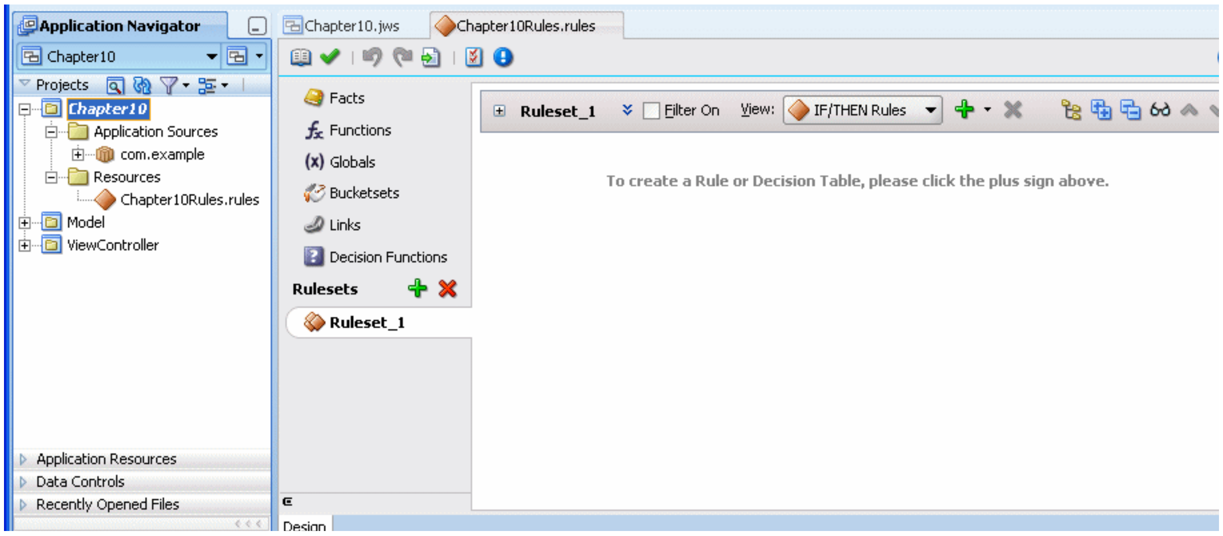
Direction	Name	Type

Help | OK | Cancel

7. Click **OK**.

   JDeveloper creates the dictionary and opens the `Chapter10Rules.rules` file in Rules Designer, as shown in Figure 10–5.

Figure 10–5 Adding the Rules Dictionary



To create a Rule or Decision Table, please click the plus sign above.

10.3.6 How to Add Decision Point Dictionary Links

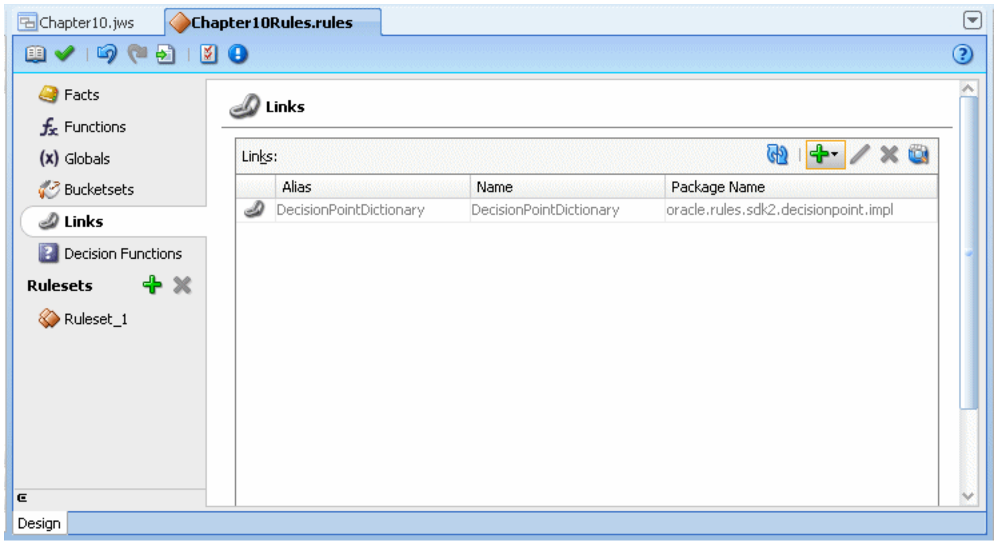
You need to add a dictionary links to the Oracle Business Rules supplied Decision Point Dictionary. This dictionary supports features for working with the Decision Point interface with ADF Business Components objects.

Add decision point dictionary links:

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

2. From the dropdown menu next to the **Create** icon, select **Decision Point Dictionary**. This operation can take awhile to complete. After waiting, Rules Designer adds a link to the Decision Point Dictionary as shown in Figure 10–6.

Figure 10–6 Adding a Dictionary Link to Decision Point Dictionary



Alias	Name	Package Name
DecisionPointDictionary	DecisionPointDictionary	oracle.rules.sdk2.decisionpoint.impl

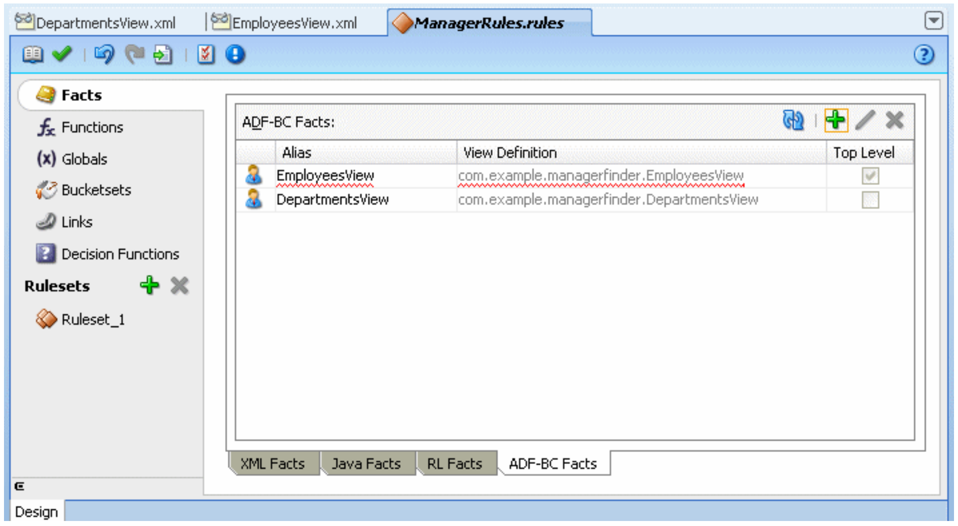
10.3.7 How to Import the ADF Business Components Facts

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...** icon. This displays the ADF Business Components Fact page.
4. In the **Connection** field, from the dropdown 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, as shown in Figure 10–7.

Figure 10–7 ADF Business Components Facts in Rules Designer



Alias	View Definition	Top Level
EmployeesView	com.example.managerfinder.EmployeesView	✓
DepartmentsView	com.example.managerfinder.DepartmentsView	

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 deselect the **Visible** checkbox for one of the properties that is involved in the circular reference.

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 deselect the **Visible** checkbox.
4. Click **OK**.

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.**

10.3.8 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 Section 10.2.1, "How to Call a Decision Point with ADF Business Components Facts".

10.3.8.1 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:

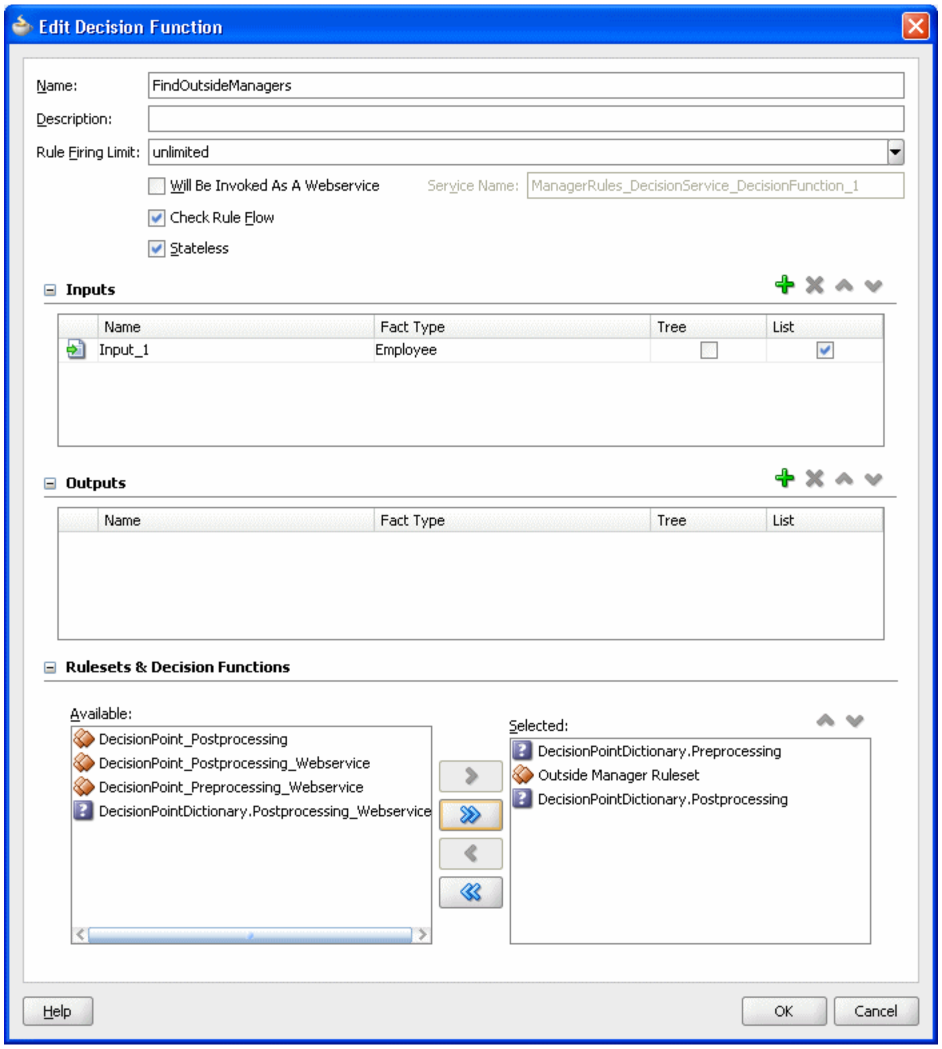
1. In Rules Designer, select the **Ruleset_1** navigation tab.
2. Select the ruleset name and enter `Outside Manager Ruleset` to rename the ruleset.

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, as shown in Figure 10–8.
   - Enter **Name** value `FindOutsideManagers`.
   - In the **Inputs** area, click the **Add Input** icon and edit the input information as follows:
     - Click the **Fact Type** field and select **Employee** from the dropdown list.
     - Select the **List** checkbox.

     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 Section 10.1.2, "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**

Figure 10–8 Adding the Find Outside Managers Decision Function


Edit Decision Function

Name: FindOutsideManagers
Description:
Rule Firing Limit: unlimited
☐ Will Be Invoked As A Webservice Service Name: ManagerRules_DecisionService_DecisionFunction_1
☑ Check Rule Flow
☑ Stateless

Inputs

Name	Fact Type	Tree	List
Input_1	Employee	☐	☑

Outputs

Name	Fact Type	Tree	List

Rulesets & Decision Functions

Available:
DecisionPoint_Postprocessing
DecisionPoint_Postprocessing_Webservice
DecisionPoint_Preprocessing_Webservice
DecisionPointDictionary.Postprocessing_Webservice

Selected:
DecisionPointDictionary.Preprocessing
Outside Manager Ruleset
DecisionPointDictionary.Postprocessing

4. Ensure that the items in the Selected area are in the order shown in Figure 10–8.

   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.

10.3.8.2 How to Create the ActionType Java Implementation Class

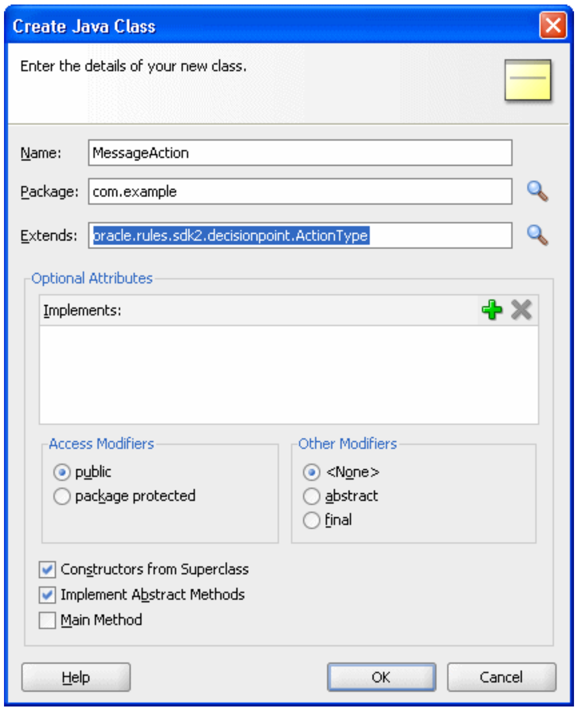
To create the sample application and to modify the view object in a rule, you need to create a Java implementation class for abstract class `oracle.rules.sdk2.decisionpoint.ActionType`. All subclasses of `ActionType` must implement the abstract `exec` method.

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 dropdown 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 as shown in Figure 10–9:
   - Enter the **Name** value `MessageAction`.
   - Enter the **Package** value `com.example`.
   - Enter the **Extends** value `oracle.rules.sdk2.decisionpoint.ActionType`.

Figure 10–9 Creating the Message Action Type Java Class


Create Java Class

Enter the details of your new class.

Name: MessageAction

Package: com.example

Extends: oracle.rules.sdk2.decisionpoint.ActionType

Optional Attributes

Implements:

Access Modifiers: [✓ public] package protected

Other Modifiers: [✓ <None>] abstract final

[✓] Constructors from Superclass

[✓] Implement Abstract Methods

[ ] Main Method

8. Click **OK**.

   Oracle JDeveloper displays the Java Class.
9. Replace this code with the code shown in Example 10–4.

Example 10–4 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 dropdown list select **Make**.

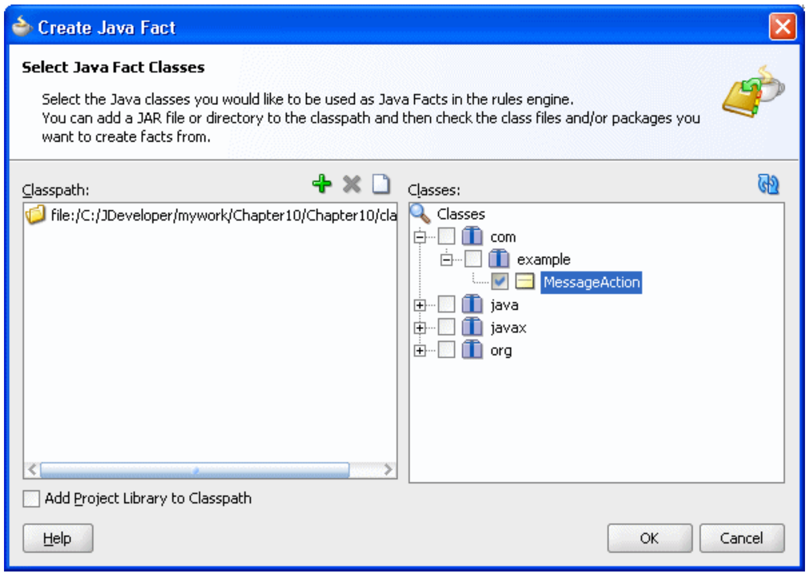
10.3.8.3 How to Import the Message Action Java Fact

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** checkbox.
5. Select the **MessageAction** checkbox, as shown in Figure 10–10.

Figure 10–10 Create Java Fact with Message Action Type



Create Java Fact

Select Java Fact Classes

Select the Java classes you would like to be used as Java Facts in the rules engine.
You can add a JAR file or directory to the classpath and then check the class files and/or packages you want to create facts from.

Classpath: file:/C:/JDeveloper/mywork/Chapter10/Chapter10/cla

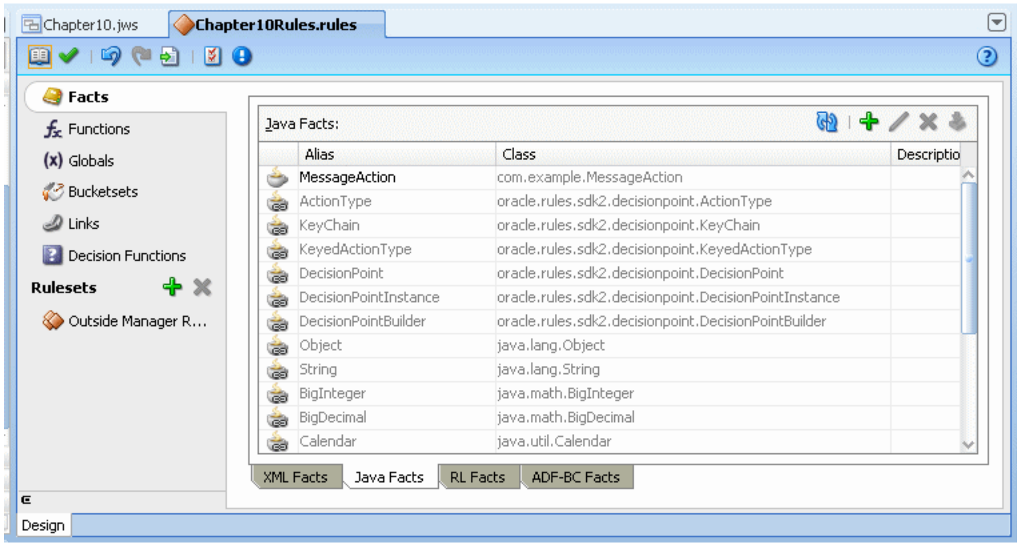
Classes:
- Classes
  - [ ] com
    - [ ] example
      - [✓] MessageAction
  - [ ] java
  - [ ] javax
  - [ ] org

[ ] Add Project Library to Classpath

6. Click **OK**.

   This adds the fact to the table, as shown in Figure 10–11.

Figure 10–11 Adding the Message Action Type Java Fact



Java Facts:

Alias	Class	Descriptio
MessageAction	com.example.MessageAction	
ActionType	oracle.rules.sdk2.decisionpoint.ActionType	
KeyChain	oracle.rules.sdk2.decisionpoint.KeyChain	
KeyedActionType	oracle.rules.sdk2.decisionpoint.KeyedActionType	
DecisionPoint	oracle.rules.sdk2.decisionpoint.DecisionPoint	
DecisionPointInstance	oracle.rules.sdk2.decisionpoint.DecisionPointInstance	
DecisionPointBuilder	oracle.rules.sdk2.decisionpoint.DecisionPointBuilder	
Object	java.lang.Object	
String	java.lang.String	
BigInteger	java.math.BigInteger	
BigDecimal	java.math.BigDecimal	
Calendar	java.util.Calendar	

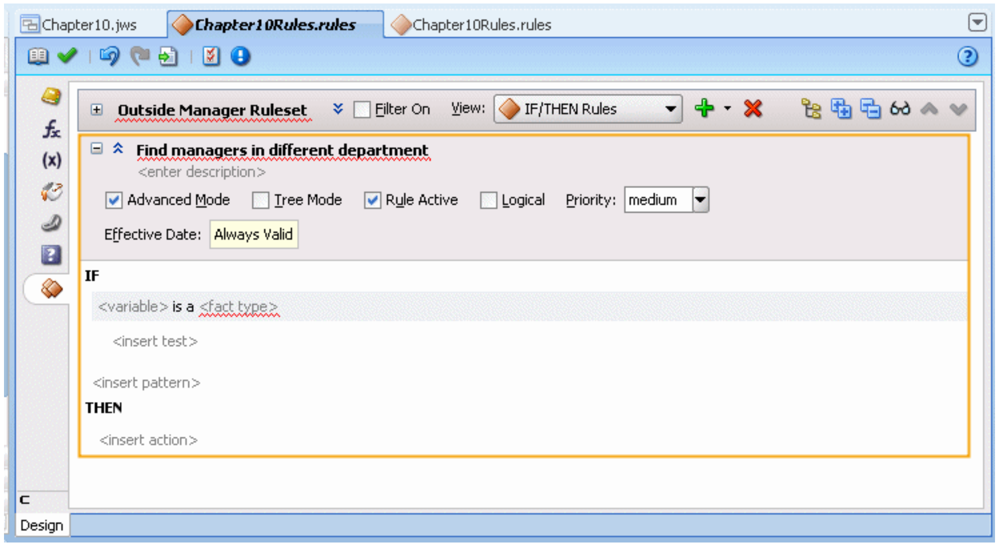
10.3.8.4 How to Add the Find Managers Rule

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 dropdown 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 Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table".
5. In the rule select **Advanced Mode**, as shown in Figure 10–12.

Figure 10–12 Adding the Find Managers in Different Departments Rule



Outside Manager Ruleset ☐ Filter On View: IF/THEN Rules

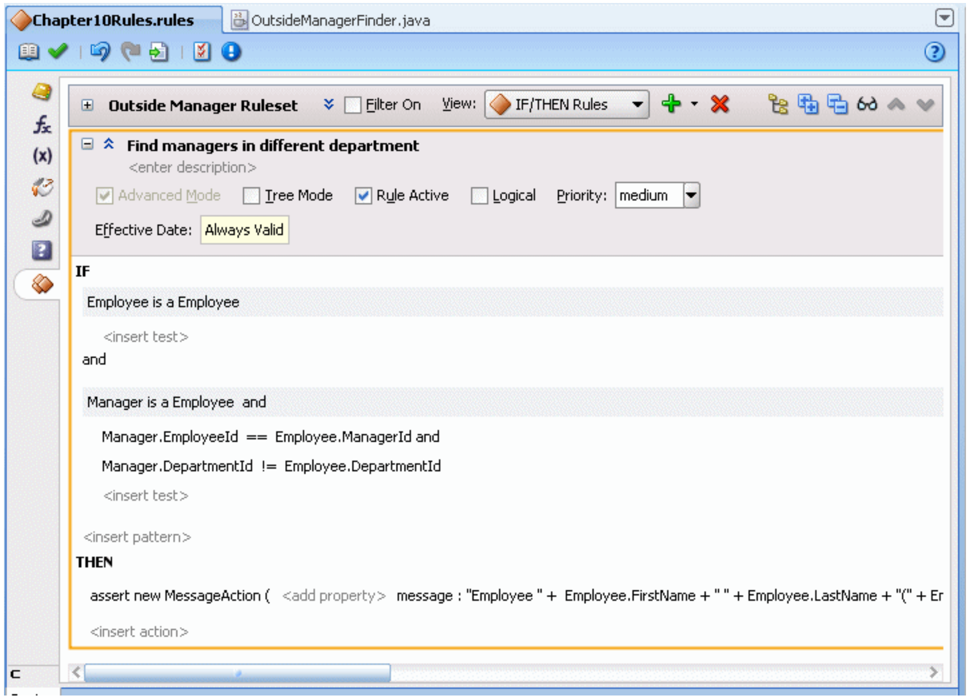
Find managers in different department
<enter description>
☑ Advanced Mode ☐ Tree Mode ☑ Rule Active ☐ Logical Priority: medium
Effective Date: Always Valid

IF
<variable> is a <fact type>
<insert test>
<insert pattern>
THEN
<insert action>

6. Enter the rule as shown in Figure 10–13. 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:

```
"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 10–13 Find Managers in Different Departments Rule



Outside Manager Ruleset ☐ Filter On View: IF/THEN Rules

Find managers in different department
<enter description>
☑ Advanced Mode ☐ Tree Mode ☑ Rule Active ☐ Logical Priority: medium
Effective Date: Always Valid

IF
Employee is a Employee
<insert test>
and
Manager is a Employee and
Manager.EmployeeId == Employee.ManagerId and
Manager.DepartmentId != Employee.DepartmentId
<insert test>
<insert pattern>
THEN
assert new MessageAction (<add property> message : "Employee " + Employee.FirstName + " " + Employee.LastName + "(" + Er
<insert action>

10.3.8.5 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**.
5. Click **OK**.
6. In the **Name** field, enter `OutsideManagerFinder`.
7. Click **OK**.
8. Replace the contents of this class with the code shown in Example 10–5.

Example 10–5 Outside Manager Finder Java Class with Decision Point

```
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();
    }

}
```

10.3.8.6 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 Section 7.5, "What You Need to Know About Using Decision Point in a Production Environment".

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 Example 10–6.

Example 10–6 Adding MDS Elements to adf-config.xml for Local Dictionary Access

```
   <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.

Copy definitions 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.
2. In the application directory for **Chapter10**, above the **Chapter10** project, navigate to the **.adf** directory.
3. Copy the **com** folder to this directory.

Copy 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, 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 **oracle** folder to this directory.

10.3.8.7 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** icon, 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**.

Run the project:

1. In the dropdown menu next to the **Run** project icon, select **OutsideManagerFinder**.
2. Running this configuration generates output, as shown in Example 10–7.

Example 10–7 Running the OutsideManagerFinder Ruleset

```
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
```

10.3.9 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, as shown in Figure 10–15. For more information, see Section 10.2.1, "How to Call a Decision Point with ADF Business Components Facts".

10.3.9.1 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**.

10.3.9.2 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** icon, click **Create Rule**.
3. Change the rule name by selecting the name **Rule_1**, and entering `Find`.

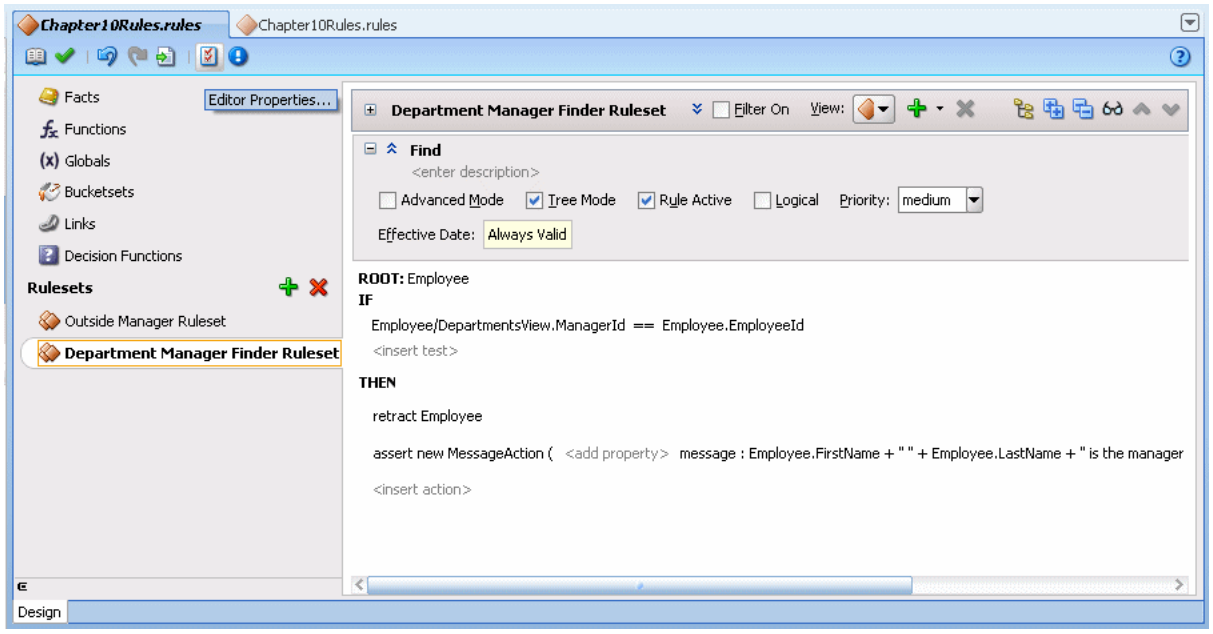
4. Click **Show Advanced Settings**. For more information, see Section 4.5.1, "How to Show and Hide Advanced Settings in a Rule or Decision Table".

5. In the rule, select **Tree Mode**.

6. Enter the **Find** rule tests and actions, as shown in Figure 10–14. The **THEN** area includes the assert that is too wide for the figure. The following shows the complete text of this rule, which is missing in Figure 10–14:

```
Employee.FirstName + " " + Employee.LastName + " is the manager of dept " +
Employee/DepartmentsView.DepartmentName
```

Figure 10–14   Adding the Find Rule to the Department Manager Finder Ruleset



Advanced Mode ☐ Tree Mode ☑ Rule Active ☑ Logical ☐ Priority: medium

Effective Date: Always Valid

```
ROOT: Employee
IF
  Employee/DepartmentsView.ManagerId == Employee.EmployeeId
  <insert test>
THEN
  retract Employee
  assert new MessageAction ( <add property> message : Employee.FirstName + " " + Employee.LastName + " is the manager
  <insert action>
```

10.3.9.3 How to Add Retract Employees Ruleset

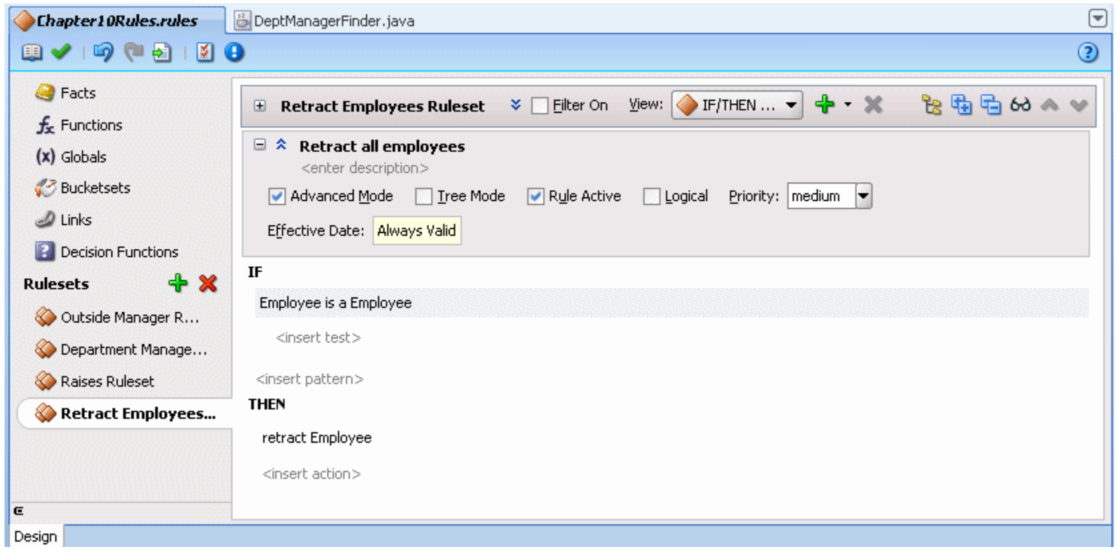
You add a ruleset to retract the employee fact type instances. This ensures that the Employee fact type is removed between invocations of the decision function.

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**, as shown in Figure 10–15.

Figure 10–15 Adding the Retract All Employees Rule



Retract all employees
<enter description>
Advanced Mode ✓ · Tree Mode · Rule Active ✓ · Logical · Priority: medium
Effective Date: Always Valid

IF
Employee is a Employee
<insert test>
<insert pattern>
THEN
retract Employee
<insert action>

10.3.9.4 How to Add the Find Department Managers Decision Function

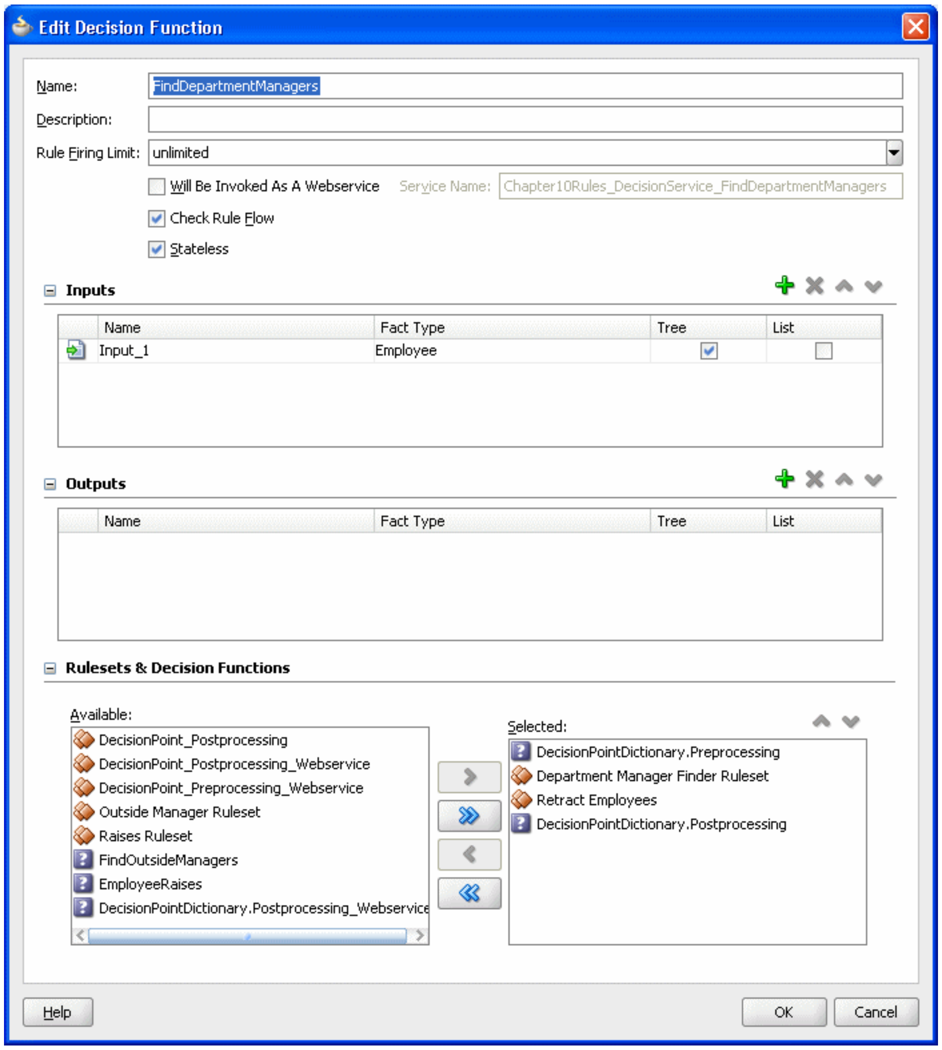
Now you create the decision function for the department manager finder ruleset. You use this decision function to execute the ruleset from a Decision Point.

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, as shown in Figure 10–16.
   - 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 dropdown list.
     – Select the **Tree** checkbox.

     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**

Figure 10–16 Adding the Find Department Managers Decision Function



Edit Decision Function

Name: FindDepartmentManagers

Description:

Rule Firing Limit: unlimited

☐ Will Be Invoked As A Webservice   Service Name: Chapter10Rules_DecisionService_FindDepartmentManagers

☑ Check Rule Flow

☑ Stateless

Inputs

Name	Fact Type	Tree	List
Input_1	Employee	☑	☐

Outputs

Name	Fact Type	Tree	List

Rulesets & Decision Functions

Available:
- DecisionPoint_Postprocessing
- DecisionPoint_Postprocessing_Webservice
- DecisionPoint_Preprocessing_Webservice
- Outside Manager Ruleset
- Raises Ruleset
- FindOutsideManagers
- EmployeeRaises
- DecisionPointDictionary.Postprocessing_Webservice

Selected:
- DecisionPointDictionary.Preprocessing
- Department Manager Finder Ruleset
- Retract Employees
- DecisionPointDictionary.Postprocessing

4. Ensure that the items in the **Selected** area are in the order shown in Figure 10–16.

   If they are not, select an item and use the **Move Up** and **Move Down** buttons to correct the order.

5. Click **OK**.

10.3.9.5 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**.
5. Click **OK**.
6. In the **Name** field, enter `DeptManagerFinder`.

7. Click **OK**.
8. Replace the contents of this class with the code shown in Example 10–8.

Example 10–8 Department Manager Finder Class

```
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();
    }
}
```

10.3.9.6 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.

10.3.9.7 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** icon, 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.
9. Click **Open**.
10. In the Edit Run Configuration dialog, click **OK**.
11. In the Project Properties dialog, click **OK**.

Run the project:

1. In the dropdown menu next to the **Run** project icon, select **DeptManager Finder**.
2. Running the decision point generates output, as shown in Example 10–9.

Example 10–9 Output from Department Manager Finder Ruleset

```
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.

10.3.10 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 Example 10–10 shows how to change the `ViewRowImpl` attribute values with a `ActionType`. For more information, see Section 10.2.1, "How to Call a Decision Point with ADF Business Components Facts".

10.3.10.1 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**.

10.3.10.2 How to Create the Raise ActionType Java Implementation Class

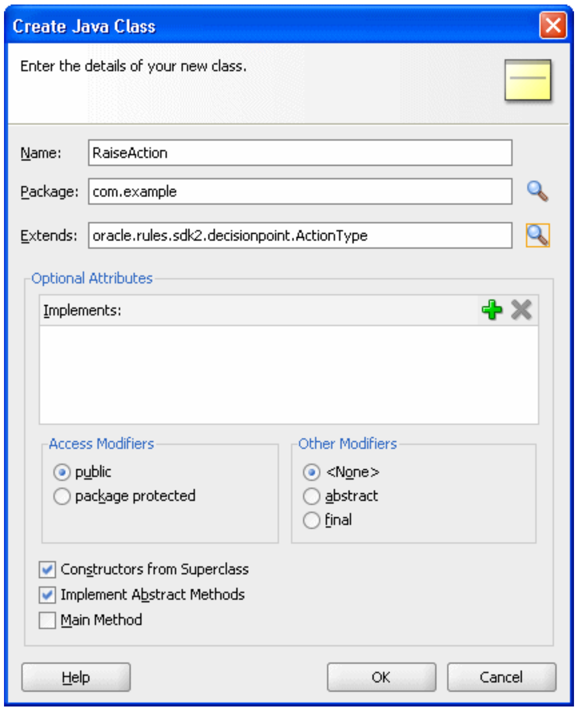
To create this part of the sample application and to modify the view object in the raises rule, you need to create a Java implementation class for the abstract class `oracle.rules.sdk2.decisionpoint.ActionType`. All subclasses of `ActionType` must implement the abstract `exec` method.

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 dropdown 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 as shown in Figure 10–17:
   - Enter the **Name** value `RaiseAction`.
   - Enter the **Package** value `com.example`.
   - Enter the **Extends** value `oracle.rules.sdk2.decisionpoint.ActionType`.

Figure 10–17 Creating the Raise ActionType Java Class



Create Java Class

Enter the details of your new class.

Name: RaiseAction

Package: com.example

Extends: oracle.rules.sdk2.decisionpoint.ActionType

Optional Attributes

Implements:

Access Modifiers: [✓ public] package protected

Other Modifiers: [✓ <None>] abstract final

[✓] Constructors from Superclass

[✓] Implement Abstract Methods

[ ] Main Method

8. Click **OK**.

   Oracle JDeveloper displays the Java Class.
9. Replace this code with the code shown in Example 10–10.

Example 10–10 ActionType Java Implementation

```
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;
    }
}
```

10. In the Application Navigator, right click the `RaiseAction.java` and from the dropdown list select **Make**.

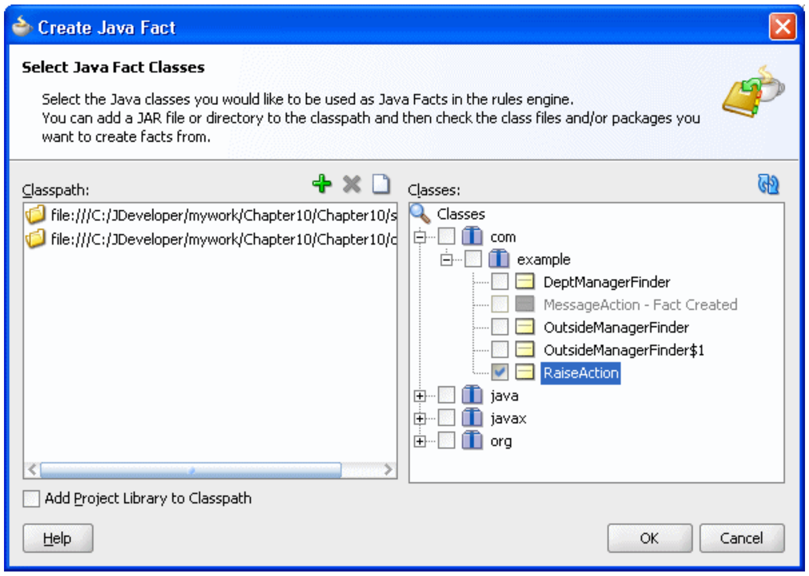
10.3.10.3 How to Import the Raise Action Java Fact

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** checkbox.
5. Select the **RaiseAction** checkbox as shown in Figure 10–18.

Figure 10–18 Create Java Fact from Raise Action Class


Create Java Fact

Select Java Fact Classes

Select the Java classes you would like to be used as Java Facts in the rules engine.
You can add a JAR file or directory to the classpath and then check the class files and/or packages you want to create facts from.

Classpath:
file:///C:/JDeveloper/mywork/Chapter10/s
file:///C:/JDeveloper/mywork/Chapter10/c

Classes:
- Classes
  - [ ] com
    - [ ] example
      - [ ] DeptManagerFinder
      - [ ] MessageAction - Fact Created
      - [ ] OutsideManagerFinder
      - [ ] OutsideManagerFinder$1
      - [✓] RaiseAction
  - [ ] java
  - [ ] javax
  - [ ] org

[ ] Add Project Library to Classpath

6. Click **OK**.

   This adds the Raise Action fact type to the **Java Facts** table.

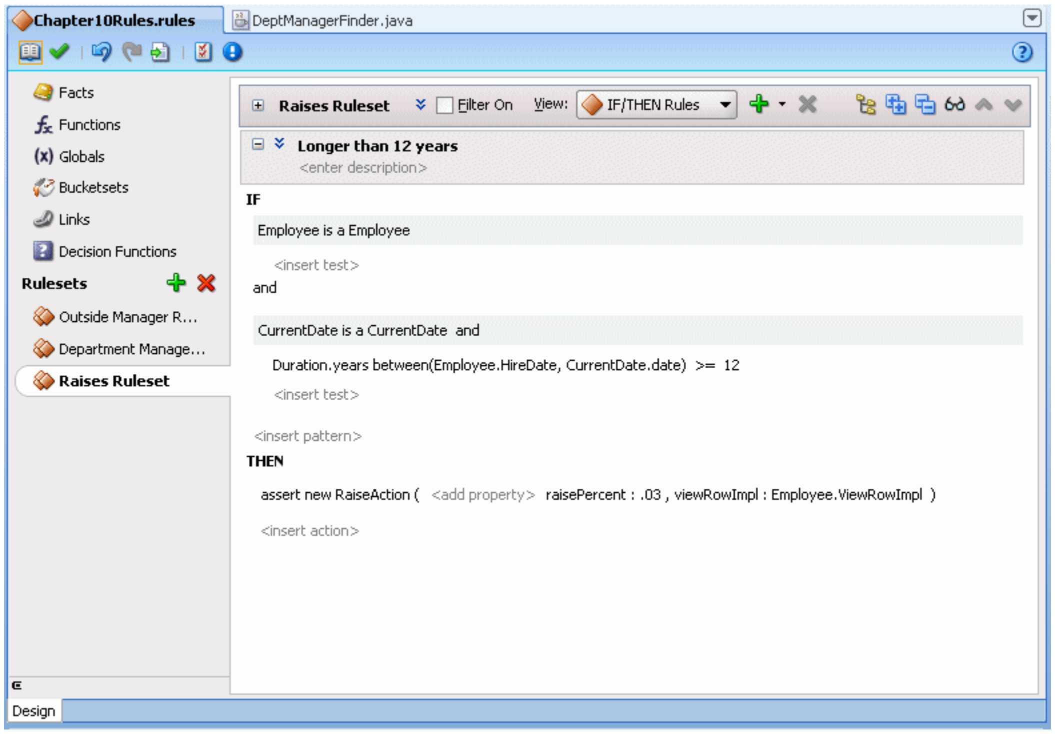
10.3.10.4 How to Add the 12 Year Raise Rule

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 Section 4.5.1, "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 Figure 10–19.

Figure 10–19 Adding the Longer Than 12 Years Rule to the Raises Ruleset



Longer than 12 years
<enter description>

```
IF
  Employee is a Employee
    <insert test>
  and
  CurrentDate is a CurrentDate  and
    Duration.years between(Employee.HireDate, CurrentDate.date)  >=  12
    <insert test>
  <insert pattern>
THEN
  assert new RaiseAction (  <add property>  raisePercent : .03 , viewRowImpl : Employee.ViewRowImpl  )
  <insert action>
```

10.3.10.5 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:

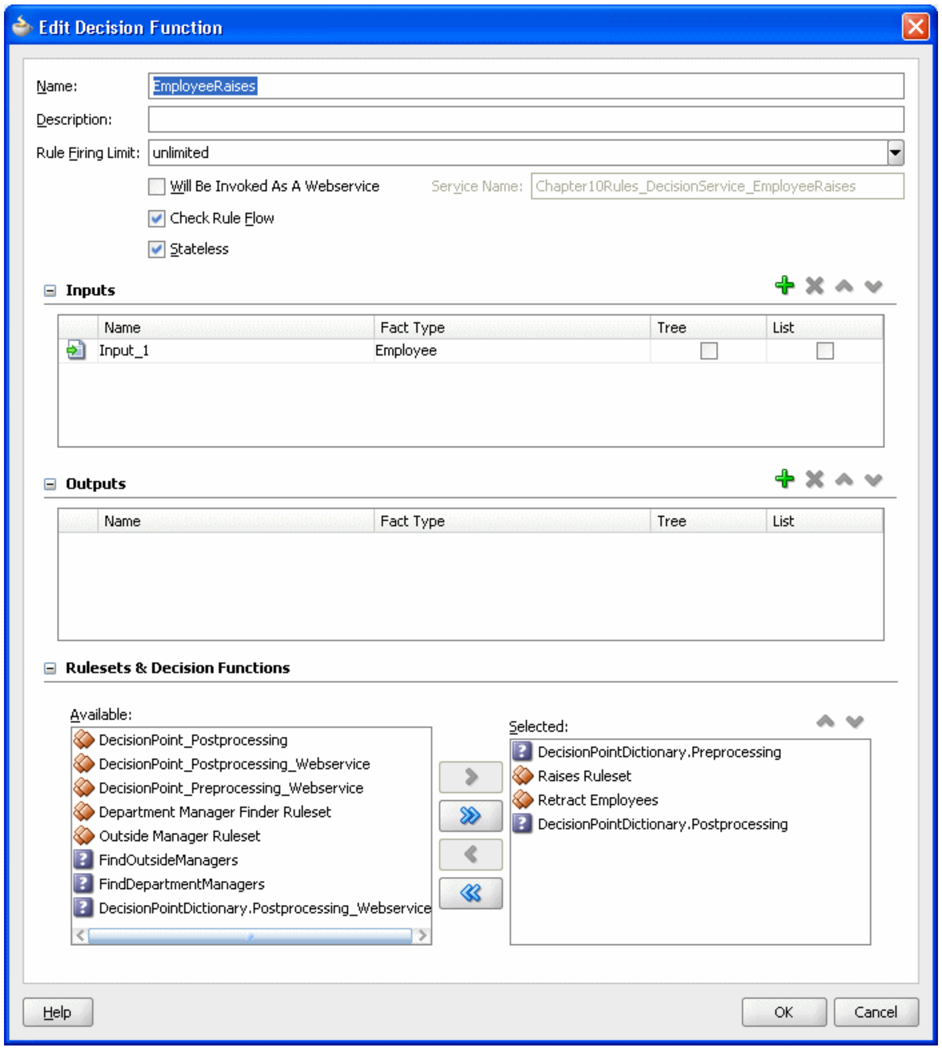
1. Click the **Decision Functions** navigation tab.
2. In the Decision Functions area, click **Create...**. This displays the Edit Decision Function dialog.
3. Update the decision function fields as shown in Figure 10–20.
   - Enter **Name** value `EmployeeRaises`.
   - In the **Inputs** area, click the **Add Input** and edit the input information as follows:

  - Click the **Fact Type** field and select **Employee** from the dropdown 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**

Figure 10–20 Adding the Employee Raises Decision Function



Edit Decision Function

Name: EmployeeRaises
Description:
Rule Firing Limit: unlimited
[ ] Will Be Invoked As A Webservice   Service Name: Chapter10Rules_DecisionService_EmployeeRaises
[✓] Check Rule Flow
[✓] Stateless

Inputs

Name	Fact Type	Tree	List
Input_1	Employee	☐	☐

Outputs

Name	Fact Type	Tree	List

Rulesets & Decision Functions

Available:
- DecisionPoint_Postprocessing
- DecisionPoint_Postprocessing_Webservice
- DecisionPoint_Preprocessing_Webservice
- Department Manager Finder Ruleset
- Outside Manager Ruleset
- FindOutsideManagers
- FindDepartmentManagers
- DecisionPointDictionary.Postprocessing_Webservice

Selected:
- DecisionPointDictionary.Preprocessing
- Raises Ruleset
- Retract Employees
- DecisionPointDictionary.Postprocessing

4. Ensure that the items in the **Selected** area are in the order shown in Figure 10–20.

   If they are not, select an item and use the **Move Up** and **Move Down** buttons to correct the order.

5. Click **OK**.

10.3.10.6 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**.
5. Click **OK**.
6. In the **Name** field, enter `EmployeeRaises`.
7. Click **OK**.
8. Replace the contents of this class with the code shown in Example 10–11.

Example 10–11 DeptManagerFinder Class

```
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();
    }
}
```

10.3.10.7 How to Copy Dictionary

Copy the updated dictionary to the MDS accessible location.

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.

10.3.10.8 How to Build and Run the Project to Check the Raises Rule

You can build and test the project by running employee raises ruleset.

Build the project:

1. From the dropdown menu next to **Run** icon, 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**.
5. Click **OK**.
6. With **EmployeeRaises** selected, click **Edit...**.
7. In the **Default Run Target** field, click **Browse...**.
8. Select **EmployeeRaises.java** from the `src\com\example` folder.

9. Click **Open**.
10. In the Edit Run Configuration dialog, click **OK**.
11. In the Project Properties dialog, click **OK**.

Run the project:

1. In the dropdown menu next to the **Run** project icon, select **EmployeeRaises**.
2. Oracle JDeveloper displays the output as shown in Example 10–12.

Example 10–12 Output from Raises Ruleset

```
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

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 a 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.

This chapter includes the following sections:

- Section 11.1, "Introduction to Decision Components"
- Section 11.2, "Working with a Decision Component"
- Section 11.3, "Decision Service Architecture"

11.1 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. Each ruleset that contains rules or Decision Tables is 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 Section 11.2.1, "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 *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

11.2 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 an 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 "Using the Business Rule Service Component" in the *Oracle Fusion Middleware Developer's Guide for 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 "Using the Business Rule Service Component" in the *Oracle Fusion Middleware Developer's Guide for 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.

11.2.1 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.

11.2.1.1 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.

In Oracle Fusion Middleware 11*g* Release 1 (11.1.1), the Decision Service metadata comprises the decision function name that is being 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.
`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 11*g* Release 1 (11.1.1), 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.

11.2.1.2 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>
```

11.2.1.3 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".

11.2.2 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 that is configured declaratively, without using RL Language programming that you use to call rules from a Java EE application or from a BPEL process.

Example 11–1 shows a *`business_rule_name`*`.decs` file `decisionServices` element that defines the metadata for an Oracle Business Rules Decision Function exposed as a service.

Example 11–1 decisionService for Decision Function Execution

```
<?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, Chapter 6, "Working with Decision Functions".

11.2.3 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 you deselect the **Stateless** checkbox 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

11.2.4 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) checkbox in the Create Business Rules dialog provides support for these two modes of operation.

When the **Reset Session** (stateless) checkbox selected, this indicates stateless operation.

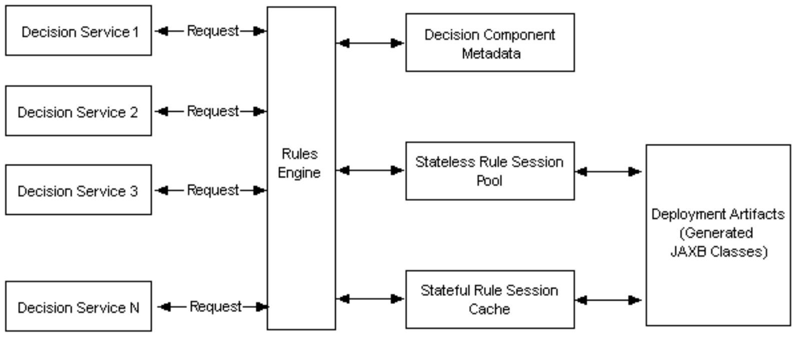
When **Reset Session** (stateless) checkbox is unselected, the underlying Oracle Business Rules object is kept in memory of the Business Rules service engine at a separate location (so that it is not given back to the Rule Session Pool when the operation is finished). Only use stateful operation if you 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) checkbox is unselected, 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.

11.3 Decision Service Architecture

A Decision service consists only of the service description. All other artifacts are shared within a decision component as shown in Figure 11–1.

Figure 11–1 Decision Service Architecture


Decision Service 1
Decision Service 2
Decision Service 3
Decision Service N
Request
Request
Request
Request
Rules Engine
Decision Component Metadata
Stateless Rule Session Pool
Stateful Rule Session Cache
Deployment Artifacts (Generated JAXB Classes)

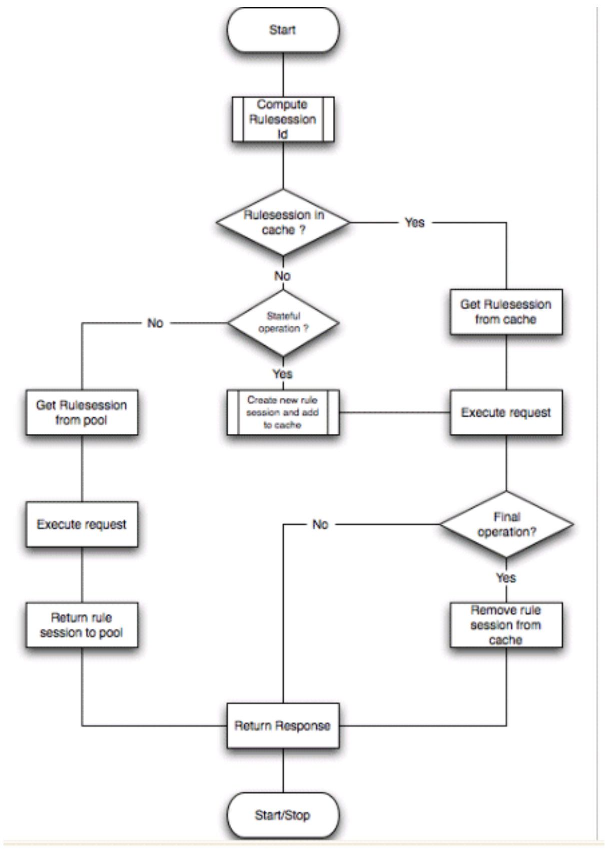
The heart of runtime is the Decision service cache, which is organized in a tree structure. Every decision component owns a subtree of that cache (depending on the composite distinguished name (DN), component, and service name). In this regard, Decision services of a decision component share the following 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 Section 11.2.3, "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



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:

- Section A.1, "Rules Designer Naming Conventions"

A.1 Rules Designer Naming Conventions

This section covers Rules Designer naming conventions.

A.1.1 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`).

A.1.2 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`.

A.1.3 Alias Naming

Rules Designer alias names must begin with a letter and contain only letters, numbers, `"."`, `"-"`, `"_",""`, `":"`, `"/"`, and single spaces.

A.1.4 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 Rules Extension Methods

This appendix lists the extension methods. This appendix includes the following sections:

B.1 Duration Extension Methods (oracle.rules.rl.extensions.Duration)

Table B–1 lists the Duration methods.

Use the Duration methods to calculate a duration between two dates. The Duration methods all take two date arguments. The Duration methods are overloaded so that the first argument is one of, `java.util.Calendar` or `javax.xml.datatype.XMLGregorianCalendar`, and the second argument is one of `java.util.Calendar` or `javax.xml.datatype.XMLGregorianCalendar`.

Rules Designer lists the arguments and valid types for each method in the list.

Table B–1 Oracle Business Rules Duration Methods

Method	Returns	Description
`compare`	`int`	Compares two dates and returns an `int` value of 0 if they are equal, less than 0 if date1 represents a point in time before date2, greater than 0 if date1 represents a point in time after date2.
`daysBetween`	`int`	Returns the difference between the two dates in days.
`hoursBetween`	`long`	Returns the difference between the two dates in hours.
`millisecondsBetween`	`long`	Returns the difference between the two dates in milliseconds.
`minutesBetween`	`long`	Returns the difference between the two dates in minutes.
`monthsBetween`	`int`	Returns the difference between the two dates in months.
`secondsBetween`	`long`	Returns the difference between the two dates in seconds.
`weeksBetween`	`int`	Returns the difference between the two dates in weeks.
`yearsBetween`	`int`	Returns the difference between the two dates in years.

B.2 JavaDate Extension Methods (oracle.rules.rl.extensions.JavaDate)

Table B–2 lists the JavaDate add and subtract methods. Table B–3 lists the JavaDate to and from string methods. For details see the ISO 8601 numeric representation of dates and times, at

`http://www.iso.org/iso/support/faqs/faqs_widely_used_standards/widely_used_standards_other/date_and_time_format.htm`.

Use the JavaDate add and subtract methods to return an updated calendar. Each method in Table B–2 returns a `java.util.Calendar`. Each method takes a first argument of `java.util.Calendar`, and a second argument that is either an `int` or a `long`, depending on the method.

Table B–2 JavaDate Add To and Subtract From Calendar Methods

Method	Arguments	Description
`addDaysTo`	`Calendar,int`	Returns a new Calendar that is the result of adding the specified number of days to the specified date.
`addHoursTo`	`Calendar,long`	Returns a new Calendar that is the result of adding the specified number of hours to the specified date.
`addMillisecondsTo`	`Calendar,long`	Returns a new Calendar that is the result of adding the specified number of milliseconds to the specified date.
`addMinutesTo`	`Calendar,long`	Returns a new Calendar that is the result of adding the specified number of minutes to the specified date.
`addMonthsTo`	`Calendar,int`	Returns a new Calendar that is the result of adding the specified number of months to the specified date.
`addSecondsTo`	`Calendar,long`	Returns a new Calendar that is the result of adding the specified number of seconds to the specified date.
`addWeeksTo`	`Calendar,int`	Returns a new Calendar that is the result of adding the specified number of weeks to the specified date.
`addYearsTo`	`Calendar,int`	Returns a new Calendar that is the result of adding the specified number of years to the specified date.
`subtractDaysFrom`	`Calendar,int`	Returns a new Calendar that is the result of subtracting the specified number of days from the specified date.
`subtractHoursFrom`	`Calendar,long`	Returns a new Calendar that is the result of subtracting the specified number of hours from the specified date.
`subtractMillisecondsFrom`	`Calendar,long`	Returns a new Calendar that is the result of subtracting the specified number of milliseconds from the specified date.
`subtractMinutesFrom`	`Calendar,long`	Returns a new Calendar that is the result of subtracting the specified number of minutes from the specified date.
`subtractMonthsFrom`	`Calendar,int`	Returns a new Calendar that is the result of subtracting the specified number of months from the date.
`subtractSecondsFrom`	`Calendar,long`	Returns a new Calendar that is the result of subtracting the specified number of seconds from the specified date.
`subtractWeeksFrom`	`Calendar,int`	Returns a new Calendar that is the result of subtracting the specified number of weeks from the specified date.
`subtractYearsFrom`	`Calendar,int`	Returns a new Calendar that is the result of subtracting the specified number of years from the specified date.

Table B–3 JavaDate Date and Time String Methods

Method	Returns	Description
`fromDateString`	`Calendar`	Creates a Calendar instance for the specified ISO 8601 date. The argument is a String.
`fromDateTimeString`	`Calendar`	Creates a Calendar instance for the specified ISO 8601 date and time. The argument is a String.
`fromTimeString`	`Calendar`	Creates a Calendar instance for the specified ISO 8601 time. The argument is a String.
`toDateString`	`String`	Return the ISO 8601 representation of the specified date. The argument is a `Calendar`.
`toDateTimeString`	`String`	Return the ISO 8601 representation of the specified date and time. The argument is a `Calendar`.
`toTimeString`	`String`	Return the ISO 8601 representation of the specified time. The argument is a `Calendar`.

B.3 XMLDate Extension Methods (oracle.rules.rl.extensions.XMLDate)

Table B–4 lists the XMLDate add and subtract methods. Table B–5 lists the XMLDate to and from string methods. For details see the ISO 8601 numeric representation of dates and times, at

`http://www.iso.org/iso/support/faqs/faqs_widely_used_standards/widely_used_standards_other/date_and_time_format.htm`.

Use the XMLDate add and subtract methods to return an updated XMLGregorianCalendar. Each method in Table B–4 returns a `javax.xml.datatype.XMLGregorianCalendar`. Each method takes a first argument of `javax.xml.datatype.XMLGregorianCalendar`, and a second argument that is either an `int` or a `long`, depending on the method.

Table B–4 XMLDate Add To and Subtract From Methods

Method	Arguments	Description
`addDaysTo`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of adding the specified number of days to the specified XMLGregorianCalendar.
`addHoursTo`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of adding the specified number of hours to the specified XMLGregorianCalendar.
`addMillisecondsTo`	`XMLGregorianCalendar, long`	Returns a new XMLGregorianCalendar that is the result of adding the specified number of milliseconds to the specified XMLGregorianCalendar.
`addMinutesTo`	`XMLGregorianCalendar, long`	Returns a new XMLGregorianCalendar that is the result of adding the specified number of minutes to the specified XMLGregorianCalendar.
`addMonthsTo`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of adding the specified number of months to the specified XMLGregorianCalendar.
`addSecondsTo`	`XMLGregorianCalendar, long`	Returns a new XMLGregorianCalendar that is the result of adding the specified number of seconds to the specified XMLGregorianCalendar.
`addWeeksTo`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of adding the specified number of weeks to the specified XMLGregorianCalendar.

Table B–4 (Cont.) XMLDate Add To and Subtract From Methods

Method	Arguments	Description
`addYearsTo`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of adding the specified number of years to the specified XMLGregorianCalendar.
`subtractDaysFrom`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of subtracting the specified number of days from the specified XMLGregorianCalendar.
`subtractHoursFrom`	`XMLGregorianCalendar, long`	Returns a new XMLGregorianCalendar that is the result of subtracting the specified number of hours from the specified XMLGregorianCalendar.
`subtractMillisecondsFrom`	`XMLGregorianCalendar, long`	Returns a new XMLGregorianCalendar that is the result of subtracting the specified number of milliseconds from the specified XMLGregorianCalendar.
`subtractMinutesFrom`	`XMLGregorianCalendar, long`	Returns a new XMLGregorianCalendar that is the result of subtracting the specified number of minutes from the specified XMLGregorianCalendar.
`subtractMonthsFrom`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of subtracting the specified number of months from the specified XMLGregorianCalendar.
`subtractSecondsFrom`	`XMLGregorianCalendar, long`	Returns a new XMLGregorianCalendar that is the result of subtracting the specified number of seconds from the specified XMLGregorianCalendar.
`subtractWeeksFrom`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of subtracting the specified number of weeks from the specified XMLGregorianCalendar.
`subtractYearsFrom`	`XMLGregorianCalendar, int`	Returns a new XMLGregorianCalendar that is the result of subtracting the specified number of years from the specified XMLGregorianCalendar.

Table B–5 XMLDate To and From String Methods

Method	Returns	Description
`fromString`	`XMLGregorianCalendar`	Creates an XMLGregorianCalendar instance from specified ISO 8601 date, datetime or time. Takes a single `String` argument.
`toString`	`String`	Return the ISO 8601 representation of the specified XMLGregorianCalendar.Takes a single `XMLGregorianCalendar` argument.

B.4 OracleDate Methods (oracle.rules.sdk2.extensions.OracleDate)

Table B–6 lists the OracleDate add and subtract methods. Table B–7 lists the OracleDate to and from string methods. For details see the ISO 8601 numeric representation of dates and times, at `http://www.iso.org/iso/support/faqs/faqs_widely_used_standards/widely_used_standards_other/date_and_time_format.htm`.

Use the OracleDate add and subtract methods to return an updated Timestamp. Each method in Table B–6 returns an `oracle.jbo.domain.Timestamp`. Each method takes a first argument of `oracle.jbo.domain.Timestamp`, and a second argument that is either an `int` or a `long`, depending on the method.

Table B–6 OracleDate Add To and Subtract From Methods

Method/Arguments	Description
`addDaysTo` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of adding the specified number of days to the specified Timestamp instance.
`addHoursTo` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of adding the specified number of hours to the specified Timestamp instance.
`addMillisecondsTo` oracle.jbo.domain.Timestamp, int	Returns a Timestamp instance that is the result of adding the specified number of milliseconds to the Timestamp instance.
`addMinutesTo` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of adding the specified number of minutes to the specified Timestamp instance.
`addMonthsTo` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of adding the specified number of months to the specified Timestamp instance.
`addSecondsTo` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of adding the specified number of seconds to the specified Timestamp instance.
`addWeeksTo` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of adding the specified number of weeks to the specified Timestamp instance.
`addYearsTo` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of adding the specified number of years to the specified Timestamp instance.
`subtractDaysFrom` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of subtracting the specified number of days from the specified Timestamp instance.
`subtractHoursFrom` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of subtracting the specified number of hours from the specified Timestamp instance.
`subtractMillisecondsFrom` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of subtracting the specified number of milliseconds from the Timestamp instance.
`subtractMinutesFrom` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of subtracting the specified number of minutes from the specified Timestamp instance.
`subtractMonthsFrom` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of subtracting the specified number of months from the date.
`subtractSecondsFrom` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of subtracting the specified number of seconds from the specified Timestamp instance.
`subtractWeeksFrom` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of subtracting the specified number of weeks from the specified Timestamp instance.
`subtractYearsFrom` oracle.jbo.domain.Timestamp, int	Returns a new Timestamp instance that is the result of subtracting the specified number of years from the specified Timestamp instance.

Table B–7 OracleDate To String and From String Methods

Method	Returns	Description
`fromString`	`oracle.jbo.domain.Timestamp`	Creates a Timestamp instance for the specified ISO 8601 date and time string. Takes a single `String` argument.
`toGregorianCalendar`	`java.util.GregorianCalendar`	Return a GregorianCalendar representing the specified Timestamp. Takes a single oracle.jbo.domain.Timestamp argument.
`toString`	`java.lang.String`	Return the ISO 8601 representation of the specified Timestamp instance. Takes a single oracle.jbo.domain.Timestamp argument.

B.5 OracleDuration Methods (oracle.rules.sdk2.extensions.OracleDuration)

Use the Duration methods to calculate a duration between two dates. The Duration methods all take two date arguments. The Duration methods are overloaded so that the first argument is one of, `java.util.Calendar`, `oracle.jbo.domain.Timestamp`, `javax.xml.datatype.XMLGregorianCalendar`, and the second argument is one of java.util.Calendar, oracle.jbo.domain.Timestamp, or `javax.xml.datatype.XMLGregorianCalendar`. Note that one argument for the methods must be a Timestamp (`oracle.jbo.domain.Timestamp`).

Rules Designer lists the arguments and valid types for each method in the list.

Table B–8 lists the OracleDuration methods.

Table B–8 OracleDuration Methods

Method	Returns	Description
`compare()`	`int`	Compares two dates and returns an `int` value of 0 if they are equal, less than 0 if date1 represents a point in time before date2, greater than 0 if date1 represents a point in time after date2.
`daysBetween()`	`int`	Returns the difference between the two dates in days.
`hoursBetween()`	`long`	Returns the difference between the two dates in hours.
`millisecondsBetween()`	`long`	Returns the difference between the two dates in milliseconds.
`minutesBetween()`	`long`	Returns the difference between the two dates in minutes
`monthsBetween()`	`int`	Returns the difference between the two dates in months.
`secondsBetween()`	`long`	Returns the difference between the two dates in seconds.
`weeksBetween()`	`int`	Returns the difference between the two dates in weeks.
`yearsBetween()`	`int`	Returns the difference between the two dates in years.

C Oracle Business Rules Frequently Asked Questions

This appendix contains frequently asked questions about Oracle Business Rules.

C.1 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 re-evaluate 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.

C.2 What are the 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 *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*.

C.3 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, the execution of the two threads might proceed as shown in Example C–1.

Example C–1 Using 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 Example C–1, 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–2, Example C–3, and Example C–4.

Example C–2 RuleSession Shared Objects

```
RuleSession ruleSession;
Object ruleSessionLock = new Object();
```

Example C–3 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;
}
```

Example C–4 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.

C.4 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 Example C–5.

Example C–5 Self-Join Using Fact F

```
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 Example C–5 output? The answer is 4 lines because the same fact instance can match for both `f1` and `f2`.

Thus, Example C–5 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.

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–6.

Example C–6 Find All Combinations of Fact F

```
rule r1 {
  if (fact F F1 && fact F F2 && F1 != F2) {
    println("Results: " + F1.i + ", " + F2.i);
  }
}
```

The code in Example C–6 gives the following output:

```
Results: 2, 1
Results: 1, 2
```

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–7.

Example C–7 Finding Combinations of Fact F

```
rule r1 {
  if (fact F F1 && fact F F2 && id(F1) < id(F2)) {
    println("Results: " + F1.i + ", " + F2.i);
  }
}
```

Because the function `id()` shown in Example C–7 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–8 shows the same rule using the oid value.

Example C–8 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 Example C–9.

Example C–9 Retracting Duplicate Facts Incorrect Sample

```
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. Example C–10 shows the correct version of this rule.

Example C–10 Retracting Duplicate Facts Corrected Sample

```
rule rRemoveDups {
  if (fact F F1 && fact F F2 && F1 != F2 && F1.i == F2.i) {
    retract(F2);
  }
}
```

C.5 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.

C.6 What Are the Limitations on a Decision Service with Oracle Business Rules?

There are some limitations for using Business Rules with a BPEL process, including 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 Appendix D.9, "How Are Decision Service Inputs and Outputs Restricted?".

For information on setting XML fact type visible option, see Section 3.2, "Working with XML Facts".

C.7 How Do I Change the Name of a Dictionary or Dictionary Package?

To change the name of a dictionary or a dictionary package name, you use JDeveloper option **File > Rename**. Do not use **Refactor > Rename**. The refactor option does not apply to dictionaries or dictionary packages.

For more information, see Section 2.2.5, "How to Rename a Dictionary or Rename a Dictionary Package" and Section D.3, "Renaming a Dictionary or Dictionary Package".

C.8 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.

C.9 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.

C.10 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 *Oracle Fusion Middleware Administrator's Guide for Oracle SOA Suite*.

C.11 How Do I Support Versioning with Oracle Business Rules?

Versioning is supported in Oracle Business Rules in two ways:

- 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.

C.12 What is the Priority Order Using Priorities with Rules and Rulesets?

The priority is highest to lowest, with the higher priority rule or ruleset 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.

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 rules to use guard clauses for representing dependencies between rules and divide the rules into multiple rulesets if they are intended to be run in a certain order. For more information, see Section 4.5.5, "How to Set a Priority for a Rule".

C.13 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 toplevel `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.

C.14 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 datamodel when they change.

C.15 How Do I Use Rules SDK to Include a null in an Expression?

You can use the following Rules SDK code to include a null value:

```
SimpleTest test = pattern.getSimpleTestTable().add();
test.getLeft().setValue(attr);
test.setOperator(Util.TESTOP_NE);
test.getRight().setValue("null");
```

C.16 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.

D Oracle Business Rules Troubleshooting

This appendix contains workarounds and solutions for issues you may encounter when using Oracle Business Rules. The following topics are covered:

D.1 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.

D.2 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)
```

D.3 Renaming a Dictionary or Dictionary Package

When renaming a dictionary or dictionary package, using JDeveloper **Refactor > Rename** has no affect on the dictionary or dictionary package name. To change the name of a dictionary or a dictionary package name, you use **File > Rename**. Do not use **Refactor > Rename**. The refactor option does not apply to dictionaries or dictionary packages.

Note that this the rename operation changes the name of the dictionary but not the alias. The alias can be changed through the Dictionary Settings dialog. In general, these should be set to the same value. For more information, see Section 2.2.4, "How to View and Edit Dictionary Settings".

For more information, see Section 2.2.5, "How to Rename a Dictionary or Rename a Dictionary Package".

To rename a dictionary that is part of a composite application that uses a Business Rules component, as part of a non-BPEL composite, you can rename a dictionary with the following steps.

To rename a dictionary in a composite application that does not use BPEL, do the following:

To rename a dictionary that is in a composite application with business rules:

1. Select the dictionary file and then select File > Rename to rename the dictionary file.
2. Open the dictionary and change the alias to match the dictionary name, using the Dictionary Settings dialog. For more information, see Section 2.2.4, "How to View and Edit Dictionary Settings".
3. In the project, rename the *dictionaryName*`.decs` file to correspond with the new dictionary name.
4. In the project, open the *dictionaryName*`.decs` file and change the <path> element value to match the renamed dictionary name.
5. In the project, rename the *dictionaryName*`.componentType` file to correspond with the new dictionary name.
6. In the project, open `composite.xml`, select the **Source** tab and edit the `<component>` element `name` attribute to match the renamed dictionary.
7. In the `composite.xml` file, edit the `<implementation.decision>` element `src` attribute to match the renamed dictionary name, with a `.decs` extension.

D.4 Runtime NoClassDefFound Error

Sometimes when working with XML facts, you might 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.

D.5 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. For more information, see *Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules*.

D.6 java.lang.IllegalAccessError from Business Rules Service Runtime

Problem: I receive an error such as the following:

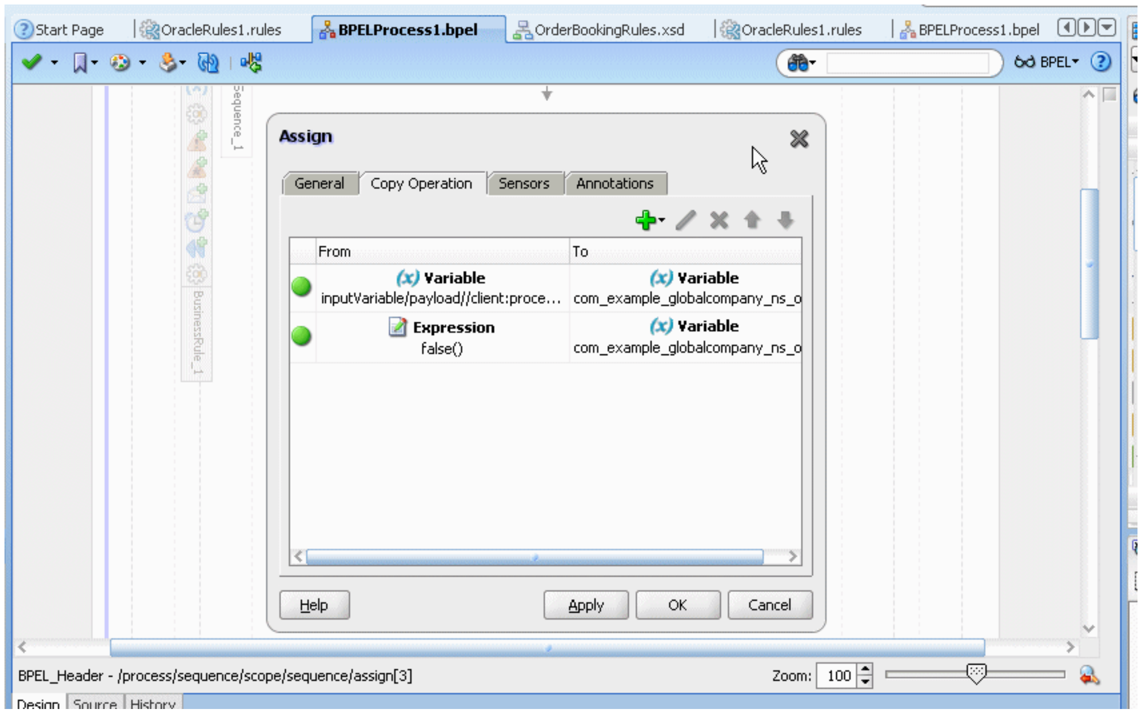
```
java.lang.IllegalAccessError: tried to access class
com.sun.xml.bind.v2.runtime.reflect.opt.Const from class:...
```

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,

https://jaxb.dev.java.net/issues/show_bug.cgi?id=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()`.

Figure D–1 Adding an Expression to Initialize a Value for a Business Rules Service Input

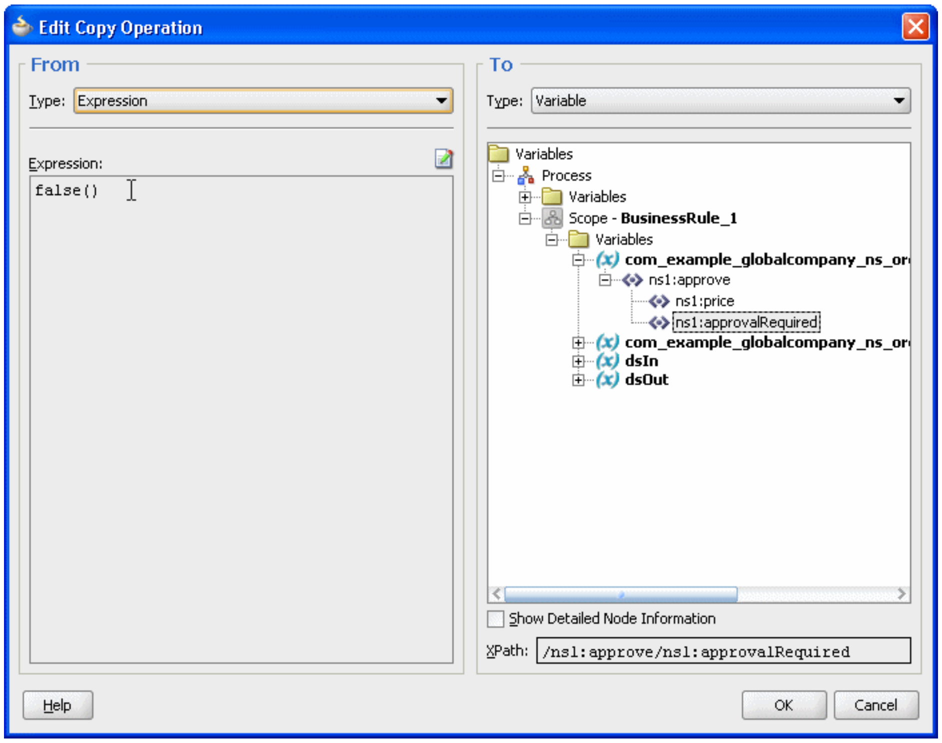


From	To
Variable inputVariable/payload//client:proce...	Variable com_example_globalcompany_ns_o
Expression false()	Variable com_example_globalcompany_ns_o

BPEL_Header - /process/sequence/scope/sequence/assign[3]

Zoom: 100

Figure D–2 Expression Assigned to Input Variable for Business Rules Service



```
false()
```

XPath: `/ns1:approve/ns1:approvalRequired`

D.7 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.

D.8 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`.

D.9 How Are Decision Service Inputs and Outputs 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*.

E

Working with Oracle Business Rules and JSR-94 Execution Sets

The specification for the Java Rule Engine API (JSR-94) defines a standard Java runtime API to access a rule engine from a Java SE or Java EE client. You can access Oracle Business Rules using JSR-94.

This chapter includes the following sections:

- Section E.1, "Introduction to Oracle Business Rules and JSR-94 Execution Sets"
- Section E.2, "Creating JSR-94 Rule Execution Sets from Oracle Business Rules Rulesets"
- Section E.3, "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`

E.1 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 Section E.2, "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 Section E.3, "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 Section E.3.4, "Using Oracle Business Rules JSR-94 Extensions".

E.2 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 Section E.3.1, "Creating a Rule Execution Set with createRuleExecutionSet".

E.2.1 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" on page E-6 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. Example E–1 shows a sample XML document for mapping RL Language text to a JSR-94 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 '\'.

Example E–1 XML Mapping File for Rulesets Defined in an Oracle RL Program

```
<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 = &quot;Rental declined &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( &quot;Rental decision is &quot; + decision.type +
                              &quot; for driver &quot; + decision.driverName +
                              &quot; for reason &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;
                        (v0_Driver.age &lt; 19))) {
                            DM.println( &quot;Rental declined: &quot; + v0_Driver.name +
                            &quot; Under age, age is: &quot; + v0_Driver.age);
                            retract(v0_Driver);
                    }
                }
            }
        </rl-text>
    </rule-source>
    <ruleset-stack>
        <ruleset-name>vehicleRent</ruleset-name>
    </ruleset-stack>
</rule-execution-set>
```

E.2.2 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" on page E-6 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. Example E–2 shows a sample XML document for mapping Oracle RL text to a JSR-94 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.

Example E–2 XMP Mapping File for Rulesets Defined in a URL

```
<?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>
```

E.2.3 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.

E.3 Using the JSR-94 Interface with Oracle Business Rules

This section describes some Oracle Business Rules specific details for JSR-94 interfaces.

E.3.1 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.

E.3.2 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`.

E.3.3 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.

E.3.4 Using Oracle Business Rules JSR-94 Extensions

This section covers the following extensions provided in the JSR-94 implementation classes.

- Using the Extended createRuleExecutionSet to Create a Rule Execution Set
- Invoking an RL Language Function in JSR-94

E.3.4.1 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.

E.3.4.2 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.

Example E–3 shows how you can to invoke an RL Language function with no arguments in a JSR-94 `StatefulRuleSession`.

Example E–3 Using CallFunction with a StatefulRuleSession

```
import javax.rules.*;
...
StatefulRuleSession session;
...
((oracle.rules.jsr94.RLStatefulRuleSession) session).callFunction("myFunction");
```

F

Working with Rule Reporter

This appendix includes the following sections:

F.1 Introduction to Working with Rule Reporter

As the size and complexity of an Oracle Business Rules dictionary grows, documenting the dictionary and communicating with others about the contents of the rules dictionary can be important. Using the `RuleReporter` class you can create lists or reports of the contents of a rules dictionary. You can use these reports to document your design and to communicate about the dictionary contents.

There are two ways to use Rule Reporter:

- Execute `RuleReporter` on the command line
- Create custom reports using the `RuleReporter` API in a Java program

Rule Reporter is written in the Groovy programming language using the MarkupBuilder class, making it easy to create custom reporters whether you simply want to have differently formatted HTML or use an entirely different markup language. Groovy is a Java-like dynamic language which runs on the JVM and interacts seamlessly with Java objects.

F.1.1 What You Need to Know About Rule Reporter HTML Style Sheets

The `*JDEV_INSTALL*/jdeveloper/soa/modules/oracle.rules_11.1.1/reporter.jar` file contains style sheet `oracle/rules/tools/reporter/style.css`. When you place this file in the same directory as the HTML output file that Rule Reporter generates, this provides definitions to render the page. You can modify the style sheet to change the HTML layout.

F.1.2 What You Need to Know About RuleReporter API

For complete details on the `RuleReporter` API, see the *Oracle Fusion Middleware Java API Reference for Oracle Business Rules*.

F.1.3 What You Need to Know About Rule Reporter Dependent Jar Files

The command-line or Java API use of Rule Reporter needs to have the classpath include all required JAR files.

F.2 Using Rule Reporter Command Line Interface

You can execute a command line script to use Rule Report to list the contents of a dictionary.

F.2.1 How to List the Contents of a Dictionary with Rule Reporter Command Line

You can execute a command line script to use Rule Report to list the contents of a dictionary.

To list the contents of a dictionary with Rule Reporter using the command line:

1. Open a terminal shell window on your system.
2. Update your classpath to include `RuleReporter` dependencies as Example F–1 shows.

   For more information, see Section F.1.3, "What You Need to Know About Rule Reporter Dependent Jar Files".
3. Run `RuleReporter` with the following command line as Example F–1 shows:

   `java oracle.rules.tools.reporter.RuleReporter DICT-NAME DEST-FILE LINK-PATHS`

   Where:

   - *`DICT-NAME`*: the name of the rules dictionary you want to generate a report on.

     For example: `C:\JDeveloper\mywork\GradeApp\Grades\oracle\rules\grades\OracleRules1.rules`.
   - *`DEST-FILE`*: the name of the destination file for the generated Rule Reporter output, usually suffixed with `.html`.

     For example: `C:\Temp\report.html`.
   - *`LINK-PATHS`*: a list of the locations on the file system which may contain dictionaries that *`DICT-NAME`* links to.

     For example: `C:\Temp`.

     If *`DICT-NAME`* does not link to any dictionaries, you must still specify at least one path.

   Example F–1 shows how to generate a report for a dictionary.

Example F–1 Executing RuleReporter on the Command Line

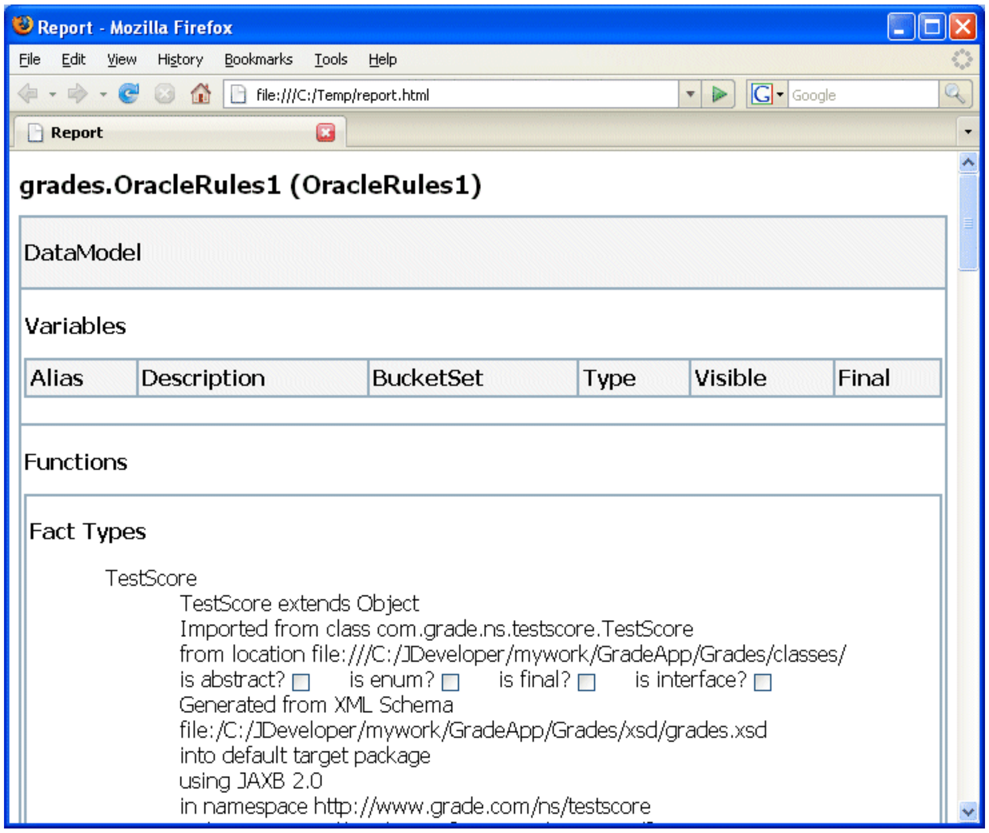
```
C:\> set CLASSPATH=%CLASSPATH%;C:\Oracle\Middleware\jdeveloper\modules\oracle.adf.model_
11.1.1\adfm.jar;C:\Oracle\Middleware\jdeveloper\modules\oracle.adf.model_
11.1.1\groovy-all-1.5.4.jar;C:\Oracle\Middleware\wlserver_
10.3\server\lib\ojdbc6.jar;C:\Oracle\Middleware\jdeveloper\soa\modules\oracle.rules_
11.1.1\rules.jar;C:\Oracle\Middleware\jdeveloper\modules\oracle.xdk_11.1.1\xmlparserv2.jar

C:\> java oracle.rules.tools.reporter.RuleReporter
C:\JDeveloper\mywork\GradeApp\Grades\oracle\rules\grades\OracleRules1.rules
C:\Temp\report.html C:\Temp
```

4. Optionally, copy the *JDEV_INSTALL*/jdeveloper/soa/modules/oracle.rules_11.1.1\reporter.jar file oracle/rules/tools/reporter/style.css

   to the same directory as the HTML output file. In this example, copy the style.css file to C:/Temp.

   This causes a web browser to use the definitions to render the page. You can modify the style sheet to change the visual layout of the HTML as shown in Figure F–1.

Figure F–1 RuleReporter report.html with style.css


Report - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Temp/report.html
Google
Report

grades.OracleRules1 (OracleRules1)

DataModel

Variables

Alias	Description	BucketSet	Type	Visible	Final

Functions

Fact Types

TestScore
TestScore extends Object
Imported from class com.grade.ns.testscore.TestScore
from location file:///C:/JDeveloper/mywork/GradeApp/Grades/classes/
is abstract? ☐ is enum? ☐ is final? ☐ is interface? ☐
Generated from XML Schema
file:/C:/JDeveloper/mywork/GradeApp/Grades/xsd/grades.xsd
into default target package
using JAXB 2.0
in namespace http://www.grade.com/ns/testscore

F.3 Using Rule Reporter with Java

You can quickly and easily create a basic report of the contents of a dictionary using a Java application with the `oracle.rules.tools.reporter.RuleReporter` class.

F.3.1 How to List the Contents of a Dictionary Using Rule Reporter with Java

You can use the `RuleReporter` class to list the contents of a dictionary. This class, `oracle.rules.tools.reporter.RuleReporter` takes several arguments, as shown:

```
RuleReporter ruleReporter = new RuleReporter(
 DICT-NAME,
 DEST-FILE,
 LINK-PATHS
);
```

Where:

- `DICT-NAME`: the name of the rules dictionary you want to generate a report on.

  For example: `C:\\JDeveloper\\mywork\\GradeApp\\Grades\\oracle\\rules\\grades\\OracleRules1.rules`.

- `DEST-FILE`: the name of the destination file for the generated Rule Reporter output, usually suffixed with `.html`.

  For example: `C:\\Temp\\report.html`.

- `LINK-PATHS`: a list of the locations on the file system which may contain dictionaries that `DICT-NAME` links to.

  For example: `new ArrayList<String>(Arrays.asList("C:\\Temp"))`.

  If `DICT-NAME` does not link to any dictionaries, you must still specify at least one path.

When you supply these arguments and call the `RuleReporter.report()` method, this produces a dictionary report for the specified dictionary.

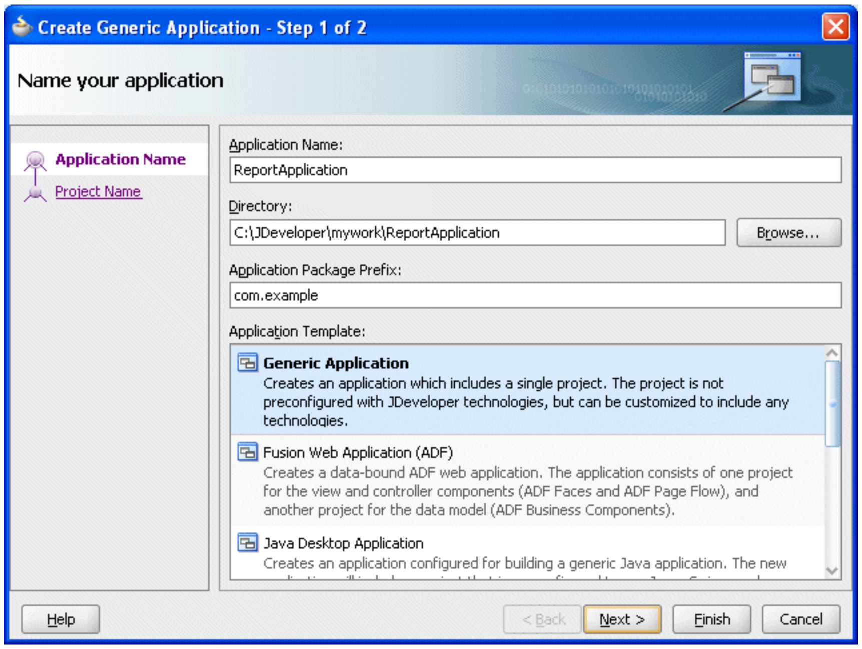
To list the contents of a dictionary using rule reporter with Java:

1. Start Oracle JDeveloper, this displays the Oracle JDeveloper start page.
2. In the Application Navigator, click **New Application** if no applications have been created, or if applications have been created, click **Applications** and from the dropdown list choose **New Application**.
3. In the Create Application wizard, enter the name and location for the application:
   a. In the **Application Name** field, enter an application name. For example, enter `ReportApplication`.
   b. Enter or browse for a directory name, or accept the default.
   c. Enter an application package prefix or accept the default, no prefix.

      This should be a globally unique prefix and commonly uses 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, you use the prefix `com.example`.
   d. For this Oracle Business Rules project, select **Generic Application** for the application template, as shown in Figure F–2.

Figure F–2 Adding the Report Application



Application Name: ReportApplication

Directory: C:\JDeveloper\mywork\ReportApplication

Application Package Prefix: com.example

Application Template:

Generic Application
Creates an application which includes a single project. The project is not preconfigured with JDeveloper technologies, but can be customized to include any technologies.

Fusion Web Application (ADF)
Creates a data-bound ADF web application. The application consists of one project for the view and controller components (ADF Faces and ADF Page Flow), and another project for the data model (ADF Business Components).

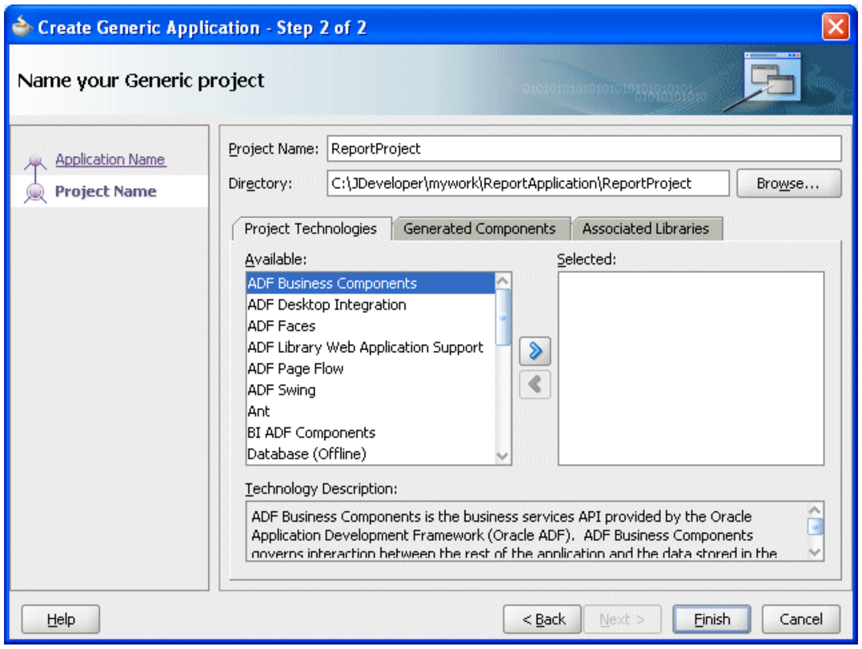
Java Desktop Application
Creates an application configured for building a generic Java application. The new

4. Click **Next**.

5. In the Create Generic Application wizard - Name your Generic project page, enter the name and location for the project as shown in Figure F–3:

   - In the **Project Name** field, enter an application name. For example, enter `ReportProject`.
   - Enter or browse for a directory name, or accept the default.
   - On the Project Technologies tab, in the Available list, select **Java** and click **Add** to add it to the Selected area.

Figure F–3 Specifying Technologies in a Project



Project Name: ReportProject

Directory: C:\JDeveloper\mywork\ReportApplication\ReportProject

Available:

ADF Business Components
ADF Desktop Integration
ADF Faces
ADF Library Web Application Support
ADF Page Flow
ADF Swing
Ant
BI ADF Components
Database (Offline)

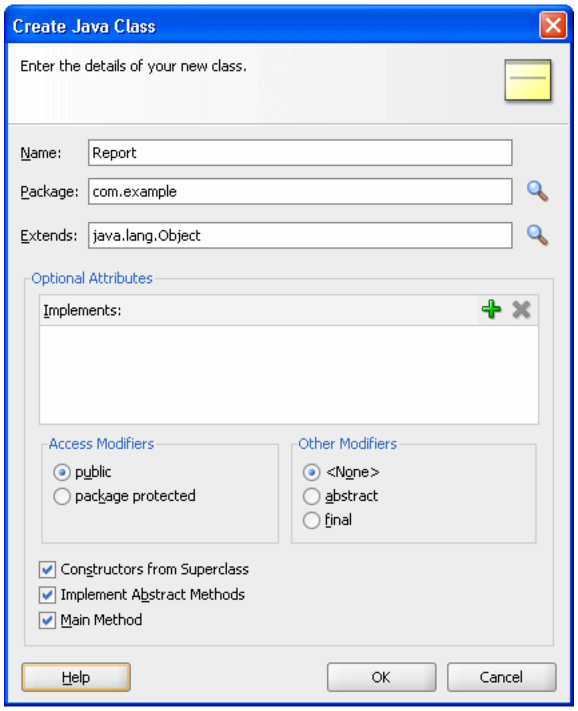
Selected:

Technology Description:

ADF Business Components is the business services API provided by the Oracle Application Development Framework (Oracle ADF). ADF Business Components governs interaction between the rest of the application and the data stored in the

6. Click **Finish**.
7. In Oracle JDeveloper, select the project named **ReportProject**.
8. Right-click and from the dropdown list select **Project Properties**.
9. Select the **Libraries and Classpath** item.
10. Add the libraries **Adfm Designtime API**, **JAXB**, **ADF Model Runtime**, **Oracle XML Parser v2**, **Oracle JDBC**, and **Oracle Rules**.
11. Click **OK**.
12. In Oracle JDeveloper, select the project named **ReportProject**.
13. Right-click and from the dropdown list select **New**.
14. In the New Gallery, in the **Categories** area, select **General**.
15. In the New Gallery, in the **Items** area, select **Java Class**.
16. Click **OK**.
17. In the Create Java Class window, configure the following properties as shown in Figure F–4:
    - Enter the **Name** value `Report`.
    - Check the following checkboxes:
      – **Public**
      – **Main Method**

Figure F–4 Creating the Report.java Class



Enter the details of your new class.

Name: Report

Package: com.example

Extends: java.lang.Object

Access Modifiers: [✓ public] package protected

Other Modifiers: [✓ <None>] abstract final

✓ Constructors from Superclass
✓ Implement Abstract Methods
✓ Main Method

18. Click **OK**.

Oracle JDeveloper displays the Java Class, as shown in Example F–2.

Example F–2 Code Created for New Report.java Class

```
package com.example;

public class Report {
    public static void main(String[] args) {
        Report report = new Report();
    }
}
```

19. Use the `RuleReporter` class as shown in Example F–3. Replace the first argument to the `RuleReporter` constructor with the location of your dictionary.

Example F–3 Report.java Completed

```
package com.example;

import java.util.List;
import java.util.Arrays;
import java.util.ArrayList;

import oracle.rules.sdk2.exception.SDKException;
import oracle.rules.tools.reporter.RuleReporter;

public class Report {
  public Report() throws SDKException {
    try {

      RuleReporter ruleReporter = new RuleReporter(
       "C:\\JDeveloper\\mywork\\GradeApp\\Grades\\oracle\\rules\\grades\\OracleRules1.rules",
       "C:\\Temp\\report.html",
       Arrays.asList("C:\\Temp")
```

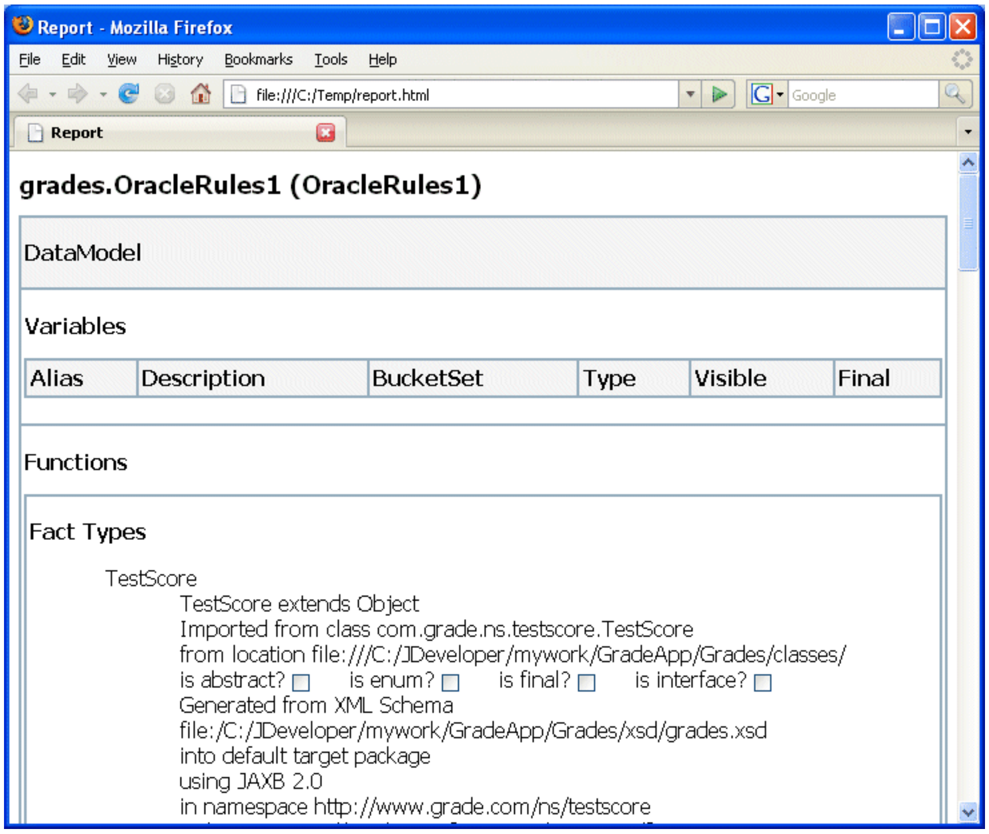
```
        );

       ruleReporter.report();

      } catch (Exception e) {
        System.out.println(e);
      }
    }

  public static void main(String[] args) throws SDKException {
      Report report = new Report();
  }
}
```

20. In the Application Navigator, right-click `ReportProject` and select **Make**.
21. In the Application Navigator, right-click `Report.java` and select **Run**.

    In this example, the `Report.java` class generates the report in `C:\Temp\report.html`
22. Optionally, copy the `*JDEV_INSTALL*/jdeveloper/soa/modules/oracle.rules_11.1.1\reporter.jar` file `oracle/rules/tools/reporter/style.css` style sheet to the same directory as the HTML output file. In this example, copy the `style.css` file to `C:/Temp`.

    This causes a web browser to use the definitions to render the page. You can modify the style sheet to change the visual layout of the HTML as shown in Figure F–5.

Figure F–5 RuleReporter report.html with style.css


Report - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Temp/report.html
Google
Report

grades.OracleRules1 (OracleRules1)

DataModel

Variables

Alias	Description	BucketSet	Type	Visible	Final

Functions

Fact Types

TestScore
TestScore extends Object
Imported from class com.grade.ns.testscore.TestScore
from location file:///C:/JDeveloper/mywork/GradeApp/Grades/classes/
is abstract? ☐ is enum? ☐ is final? ☐ is interface? ☐
Generated from XML Schema
file:/C:/JDeveloper/mywork/GradeApp/Grades/xsd/grades.xsd
into default target package
using JAXB 2.0
in namespace http://www.grade.com/ns/testscore

Index