WebSphere Application Server Enterprise Services

Business Rule Beans (BRBeans)

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices”on page .

Contents

Part I: Concepts and Architecture

Business Rule Beans (BRBeans)

The **Business Rule Beans** (BRBeans) framework extends the scope of Websphere Application Server Enterprise Edition to support business applications that **externalize their business rules**. You can use BRBeans to create and modify rules to keep pace with your complex business practices so that your application's core behavior, and user interface objects remain intact and untouched.

Rule externalization is accomplished by extending the application analysis and design processes to identify the points of variability in application behavior. These are called **trigger points**: small pieces of code that interface with the BRB trigger point framework to run business rules during application execution.

The user can also employ standard Java development tools to attach BRBeans to either **Enterprise Java Beans (EJBs)** or ordinary **Java** objects. Programming a unique new rule implementation in Java is usually a simple process, made easier by the set of pre-defined rule implementors provided that can be used to create your own business rules.

What is a business rule?

A business rule is a statement that defines or constrains some aspect of a business by asserting control over some behavior of that business. A business rule officiates over frequently changing business practices, and can come from within the company, or be mandated from outside, typically by regulatory agencies.

At its simplest level, a business rule is little more than a well placed **IF/THEN** statement that compares a variable against a determined value, and then issues a command when they match.

For example, consider the following business rules:

Example 1:

IF *a person is a senior citizen,* ***THEN*** *provide a 5% discount.*

In this case, the variable information includes:

- the age at which a person is considered to be a senior citizen
- the amount of the discount.

Example 2:

IF *a person drives more than 150 miles a week to and from work,* ***THEN*** *add $25 to their auto insurance premium.*

In this case, the values that might change are:

- the number of miles
- the amount of money to add to the premium.

Example 3:

A bank wants to classify its customers based on their investment (checking, savings, CDs, loans) according to the following business rules:

- ***IF** a customer invests less than $5000 **THEN** place the customer in the bronze level.*
- ***IF** a customer invests $5000 or more but less than $10,000, **THEN** classify the customer as silver.*
- ***IF** a customer invests more than $10,000, **THEN** consider the customer as gold.*

In this case, the bank would run into long-term problems if these categories were coded into their application data. What would happen if the bank wanted to change the boundaries between each level? Or, what if they decided to add another level in the future, such as platinum? If the rules were internal, they would have to be modified at their every occurrence in the code. This is just one advantage to externalizing rules. For more, see "Why externalize rules?" on page 4 .

Different types of business rules

The two types of rules are:

- **base** rules (of which there are several kinds), and
- **classifier** rules.

Base rules are the most common type of rule used, and are triggered with the `TriggerPoint.trigger` method.

There are a number of kinds of base rules:

- **Derivation rule:** A rule that uses an algorithm to return a value. It can return any type of value that makes sense in the business context in which it is used. For example, a derivation rule may calculate a discount or compute the total price of an order.
- **Constraint rule:** A rule that confirms that an operation has met all of its obligations, and that a particular constraint or edit has been met. For instance, it may check that a value entered by an external user is within legal bounds. BRBeans provides a special return type: `com.ibm.websphere.brb.ConstraintReturn`, which can be returned by a constraint-type rule. A `ConstraintReturn` object contains a boolean value so that if it is false, it can contain information that can be used to produce an external message explaining what constraint was not met.
- **Invariant rules:** A rule that ensures that multiple changes made by an operation are properly related to one another.
- **Script rules:** Scripts implement "micro-workflow" or electronic performance support. They are small, variable pieces of business process which can provide assistance to end-users to get the most from the application.

On the surface, **classifier** rules are much like base rules, however they differ in that they can be used to determine the ways in which variables are classified by a business. Classifier rules are triggered with the `TriggerPoint.triggerClassifier` method.

A classifier rule is used to compute a classification for a particular business situation. The classification returned is required to be a String. For instance, a bank customer may be classified into Gold, Silver, and Bronze categories based on their spending history or the amount of money they have in their account. For more information on the kind of support that BRBeans has for this type of rule, refer to "Situational Trigger Point - an example" on page 26.

BRBeans development and maintenance roles

Business Rule Beans can be used by anyone needing to externalize their business rules. Specifically though, they are created and maintained by the following individuals:

- the **application developer** and
- the **business analyst**.

The application developer is the individual who writes the application that uses the rules, and so is responsible for their initial creation.

The application developer will typically assign the following to a rule:

- a name
- a description
- possibly a classification
- rule implementor class
- a firing location
- a default start date (optional)
- a default set of init parameters (the business rule values) (optional)

In contrast, **the business analyst** determines the values for the business rules. They would provide the following additional information:

- start and end date
- init parameters
- business intent

This kind of working arrangement is advantageous as both individuals may work with the same tool (the Rule Management Application), the developer may create a rule without needing to know the values that are to be contained inside it, and the analyst can modify it armed only with the knowledge of the rule name and the folder it is located in.

Why externalize rules?

Here is a list of some advantages of externalizing business rules:

- **Explicit documentation of business practice decisions.** Externalizing them gets the rules out of people's heads and out of the application code making it available for others to view and understand.
- **Clearer understanding of application behavior.** Externalization makes it possible to inspect the application to see which business rules are being applied, when, and under what circumstances.
- **Reuse of rules across business processes.** Separating rules from the business logic of the application makes it easy to reuse a business practice decision in a consistent fashion.
- **Increased consistency of business practices.** Because externalized rules promote reuse and facilitate clear understanding of business practice decisions, they provide a basis for improving business practice consistency across applications.
- **Decreased maintenance and testing costs.** Externalized rules have a clearly defined scope and are not tightly coupled to the application code. This makes them easy to modify and quick to test, decreasing costs and improving cycle time.
- **Improved manageability of business practice decisions**. Externalization, change history, and inspectability all promote clear ownership and consequently a better definition of who can change rules and under what circumstances.

- **Increased confidence in predicting the business impact of proposed changes.** Because rules are available for inspection, have well-defined scope, and are not tightly coupled to application business logic, they make it easy to understand the likely impact of changes and to predict whether contemplated modifications or additions will have unwanted ripple effects.
- **Easy ability to identify and correct conflicting business rules in different parts of the business.** Externalized rules make it easy to check that rules being used in two different parts of an application, or even two different applications dealing with different parts of the business, are consistent.

Part II: Where to begin

Database considerations

The following relational databases are supported by BRBeans:

- DB2
- Oracle
- Sybase
- Informix

This documentation does not provide you with specific instructions on how to use any of these databases. For help with specific commands, consult the documentation that accompanied your database software . For installation and configuration instructions, refer to the documentation for Advanced Edition (InfoCenter).

Large character data

There are several attributes in the BRBeans Rule EJB that may contain large amounts of data. This would include such fields as: `businessIntent, dependentRules, description, firingParameters, initParameters, originalReq`, and `userDefinedData`. The value for these attributes is stored in a character type column within a database table. When possible, they are stored in large character fields like `LONG VARCHAR` (for DB2) and `TEXT` (for Sybase).

There are several cases where the use of large character fields is problematic, mostly in terms of a lack of query support. Refer to each of the supported database sections for details on the column type used for storing the values in these attributes.

Isolation level

All EJBs accessed in a transaction must specify the same isolation level. If your application contains EJBs that are used in the same transaction as the rules, you must do one of the following:

1. **Change the BRBeans EJBs (`Rule, RuleFolder, and RuleHelper`) to the same isolation level as your beans.**
2. **Change your beans to the same isolation level as the BRBeans EJBs.**

   Place the BRBeans EJBs in a different database than your EJBs, and configure the application to run two phase. This causes the beans to run in a different transactions, thereby removing the restriction that they need to have the same isolation level.

Oracle considerations

Large character data

The preferred Oracle data type for storing large character objects is `CLOB`. However, Oracle does not allow a `CLOB` to be queried. Because of this, a datatype of `VARCHAR2` is used by BRBeans. A specific length must be specified when specifying `VARCHAR2`. The maximum length for a `VARCHAR2` is 4000.

To determine the default size of `VARCHAR2`, look in the table.ddl file that was generated when you deployed the code. If the default size is not acceptable for your application, you can do one of the following:

1. **Increase the size of the columns.**

   Keep in mind that maximum size for a `VARCHAR2` in Oracle is 4000. Increase the column size either by changing the value in the **create table statement**, or by changing the schema mapping and deploying the BRBeans jar file.

2. **Change the schema mapping to specify `CLOB`**

   Do this for any of the attributes that you do not wish to query, and then deploy the BRBeans jar file.

Isolation level

The default isolation level is `REPEATABLE_READ`. Oracle does not support this isolation level. Therefore, the Websphere runtime will convert this to the next highest isolation level, which in this case is `SERIALIZED`. Be aware that this isolation level tends to be overly restrictive, as it prevents two clients from reading data at the same time. The `BRBeansOracle.jar` file specifies an isolation level of `READ_COMMITTED`.

Sybase considerations

Allowing null values

By default, Sybase does not allow null values in string columns (like `VARCHAR`, `TEXT`, etc). You can change this default value for a database using "isql" by issuing the following command:

```
sp_dboption databasename, "allow nulls by default", true
```

In this example "databasename" is your database name.

Large character data

The large character data fields are stored in a column of type `TEXT`. Sybase allows `TEXT` fields to be queried only using the SQL "`LIKE`" operator. Queries against these columns that

perform the SQL "IS NULL" or "IS NOT NULL" operations are not allowed by Sybase. The alternative is to specify a column type of VARCHAR. However, the maximum allowed size for a VARCHAR in Sybase is 255 characters. This is not considered a large enough value for the default size for storing firingParameters, initParameters, descriptions, etc.

If performing "IS NULL" and "IS NOT NULL" type queries is important and the 255 character limitation is acceptable, change these column types to VARCHAR. This should be accomplished by altering the schema mapping for the Rule bean and then deploying the BRBeans jar file.

The query APIs (in the com.ibm.websphere.brb.query package) allow for "IS NULL" and "IS NOT NULL" type queries to be performed on several of these fields. In addition, the "Rule Management Application" on page 14allows the firing parameters to be queried in this manner. These queries will fail on Sybase with the default column type of TEXT.

Informix considerations

Large character data

The preferred Informix data type for storing large character data is CLOB. However, Informix does not allow a CLOB to be queried. Because of this, a datatype of LVARCHAR is used by BRBeans. The maximum length for an Informix LVARCHAR is 2K. If 2K is not acceptable, and your application does not need to query these datatypes, you can change the schema mapping to specify CLOB. Then deploy the BRBeans jar file.

Custom properties for the datasource

When configuring the datasource for your application, you must specify the following properties:

Note: The following configuration values are subject to change. Consult your Informix documentation for updates.

- ifxIFXHOST=Name of the physical machine on which the Informix instance is installed
- serverName=Informix instance name
- portNumber=Port number for which the Informix instance is configured
- informixLockModeWait=500

  A setting of 500 causes a connection to wait for up to 500 seconds for a lock. If you have a busy system, this wait can appear to be a system hang. This setting has the same effect as running SET LOCK MODE TO WAIT 500 on the connection.

Getting Started with BRBeans

To work with BRBeans rules, you must install the EJBs that are used to implement them. To do this, use one of the following two methods:

1. Install one of the 2 sample ear files that are provided in the <WAS-HOME>\Enterprise\samples\BRBeans directory. For detailed installation instructions on how to do this, see BRBeansSimpleSample.html and BRBeansMovieSample.html in this directory.
2. Install one of the BRBeans...jar files in the <WAS-HOME>\Enterprise\BRBeans directory. If you are running AE, then go "here" on page 9 for instructions. If you are running AEd, then go "here" on page 12.

When you are ready to ship your application, you should include a BRBeans jar file in your ear file. See “Including BRBeans in your application” on page 47fo details on how to do this.

When you have completed these steps, you will have installed and configured Business Rules Beans for use with WebSphere. To familiarize yourself with it, you can test it out with the **Samples** provided in “Part VII” on page 49.

Installing a BRBeans jar file on AE - overview

There are several BRBeans jar files in the <*WAS-HOME* \Enterprise\BRBeans directory, each of their names reflecting the database that they support (ex: BRBeansDB2.jar). To install the one for your database, follow these tasks in sequence:

1. “Create a database” on page 9
2. “Create a datasource and jdbc provider” on page 9
3. “Create a server” on page 10
4. “Install the BRBeans <.jar file>” on page 10
5. “Creating the database tables” on page 11

When you have completed these steps, you will have installed and configured Business Rules Beans for use with WebSphere AE. To familiarize yourself with it, you can test it out with the **Samples** provided in “Part VII” on page 49.

Creating a database

The following task applies to both the AE and AEd platforms.

Determine whether you want to use an existing database with the BRBeans tables, or create a new one. If you decide to create a new one, consult your database documentation for instructions on how to do this. (If you're using DB2, you would create a new database called **brbeans**, by typing `db2 create database brbeans` at a DB2 prompt.

You have successfully created a database.

Proceed to “Creating a datasource and jdbc provider” on page 9 .

Creating a datasource and jdbc provider

Follow this procedure when installing on an AE platform, if you are using AEd, then see “Installing a BRBeans jar file on AEd - overview” on page 12

You need to make sure that you have a data source and JDBC provider specified for the database. You may use existing ones or create new ones. To create new ones, here are some steps you may follow:

1. From the **WebSphere Administrative Console**, start the **Create Data Source** Wizard.
   a. In the **Specifying Datasource Resources** window, specify the following:
      - **Name:** brbds
      - **Database name:** brbeans (or the name of the database you are using)
   b. In the **Specifying JDBC Provider** window:
      - Select - **Create a new JDBC provider**
   c. In the **Create New JDBC Provider** window, specify the following:

- Name:
  brbjdbc
- **Implementation class:** - from button on right, select the class corresponding to the platform and database. So, for example, for Windows and DB2, select `COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource`.

d In the **Completing the DataSource wizard** panel, click **Finish**.

2. In the **WS Admin Console**, expand the folders to Resources/JDBC Providers and click on brbjdbc

a Click on the **Nodes** tab in the right panel

b Click on the **Install New** button

c In the **Install Driver** window:

- Click on the node for the driver. For the classpath, specify the classpath directory to the db driver.
- Click **Install**.

d Click **Apply**.

You have successfully created a datasource and jdbc provider.

Proceed to "Creating a server" on page 10.

Creating a server

Follow this procedure when installing on an AE platform, if you are using AEd, then see "Installing a BRBeans jar file on AEd - overview" on page 12

Decide on a server to use for the sample. You may use an existing server or create a new one. To create a new one, follow these directions:

1. From the **Administrative Console**, start the **Create Application Server** wizard
2. In the **Specifying Application Server Properties** window, proceed as follows:

a **Application server**: brbsrv

b Select the node

c Click **Next**

d Click **Next** on the following window

e Click **Finish** on the next window

You have successfully created a server.

Proceed to "Installing the BRBeans <.jar file>" on page 10.

Installing the BRBeans <.jar file>

Follow this procedure when installing on an AE platform, if you are using AEd, then see "Installing a BRBeans jar file on AEd - overview" on page 12

From the **WebSphere Administrative Console**, start the **Install Enterprise Application** Wizard and proceed as follows:

1. In the **Specifying the Application or Module** window, select to **Install**

standalone-module (*.war, *.jar), and then browse for the path to the BRBeans<DBtype>.jar file. Click **Next** to continue.

2. In the **Unprotected Methods** window, click **Yes**.
3. In the **Mapping Users to Roles** window, click **Next** if you have not enabled security. If you have enabled security, click **Select** for each of the Roles and make the appropriate selections. Choose at least 1 user or group for each of the following roles: **RuleManager** and **RuleUser**. Click **Next**.
4. In the **Mapping EJB RunAs Roles to Users** window, click **Next**.
5. In the **Bind EJB to JNDI Name** window, make sure there are 3 EJBs listed with JNDI Names starting with `brbeans/application/...`. Click **Next** to proceed.
6. In the **Mapping EJB References to EJB** window, make sure there are 6 EJB References listed (3 for Rule, 3 for RuleFolder). Click **Next**.
7. In the **Mapping Resource References to Resources** window, click **Next**.
8. In the **Specifying the Default Datasource for EJBModule** window, make sure the BRBeans<DBtype>.jar file is selected. Then click **Select Datasource...** and choose the datasource for the database where the BRBeans tables were created above. (For, example, select brbds if you created it above.) Click **Next**.
9. In the **Specifying Datasource for Individual CMP Beans** window, click **Next**.
10. In the **Selecting the Virtual Hosts for Web Modules** window, click **Next**.
11. In the **Selecting the Application Server** window, make sure BRBeans<DBtype>.jar is listed. Then click **Select Server...** and select a server to use. Click **Next**.
12. In the **Completing the Application Installation Wizard** window, click **Finish**.
13. In the window asking if the application should be deployed, click **Yes**. In the subsequent window proceed as follows:
    a. For the **Dependent classpath** the following jar files need to be in the classpath: (On a Windows system, use ";" as the separator between jar files, on a Unix based system, use ":" as the separator)
       - `WAS_HOME\lib\brbClient.jar`
       - `WAS_HOME>\lib\brbServer.jar`
    b. For the **Database type**, select the type of database that you are using.
    c. Click **OK**
14. If the install is successful, an info dialog should appear that says **Command "EnterpriseApp.install" completed successfully**.

You have successfully installed the BRBeans <.jar file>.

Proceed to "Creating the database tables" on page 11.

Creating the database tables

Follow this procedure when installing on an AE platform, if you are using AEd, then see "Installing a BRBeans jar file on AEd - overview" on page 12

1. Extract the file of SQL statements that was generated in "Installing the BRBeans <.jar file>" on page 10 . This file can be found in the `WAS_HOME\installedApps\BRBeans<DBtype>.ear\BRBeans<DBtype>.jar` file as `META-INF/Table.ddl`.
2. Use the SQL statements to create the database tables. For example, if you are

using DB2, launch a DB2 command window and type the following:

```
db2 connect to <database-name>
db2 -tf <fully-qualified-path-to-Table.ddl>
```

You have successfully created the database tables.

This is the last of the steps necessary to install Business Rules Beans for use with WebSphere AE. To familiarize yourself with it, you can test it out with the **Samples** provided in "Part VII" on page 49.

Installing a BRBeans jar file on AEd - overview

There are several BRBeans jar files in the `<WAS-HOME>\Enterprise\BRBeans` directory, each of their names reflecting the database that they support (ex: BRBeansDB2.jar). To install the one for your database, follow these tasks in sequence:

1. "Deploy the code" on page 12
2. "Create a database" on page 12
3. "Create the database tables" on page 13
4. "Create a datasource" on page 13
5. "Install the BRBeans <.jar file>" on page 13

When you have completed these steps, you will have installed and configured Business Rules Beans for use with WebSphere AEd. To familiarize yourself with it, you can test it out with the **Samples** provided in "Part VII" on page 49.

Deploying the code

Follow this procedure when installing on an AEd platform, if you are using an AE platform, then see "Installing a BRBeans jar file on AE - overview" on page 9

1. Launch the **Application Assembly Tool** by typing `assembly.bat/sh` on a command line.
2. Open BRBeans<DBType>jar in the *WAS-HOME* \Enterprise\BRBeans directory.
3. From the File menu, select **Generate code for deployment...**.
   a. Specify the following paths for the Dependent classpath. The paths should be separated either by a ";" (on Windows platforms) or by a ":" (on Unix based platforms).
      - `WAS_HOME\lib\brbClient.jar`
      - `WAS_HOME\lib\brbServer.jar`
   b. For Database type, select the type that you will be using.
   c. Click **Generate Now**.

   This will create a new .jar file called Deployed_BRBeans<DBType>.jar in the same directory.

You have successfully deployed the code.

Proceed to "Creating a database" on page 12.

Creating a database

The following task applies to both the AE and AEd platforms.

Determine whether you want to use an existing database with the BRBeans tables, or

create a new one. If you decide to create a new one, consult your database documentation for instructions on how to do this. (If you're using DB2, you would create a new database called **brbeans**, by typing `db2 create database brbeans` at a DB2 prompt.

You have successfully created a database.

Proceed to "Creating the database tables" on page 13.

Creating the database tables

Follow this procedure when installing on an AEd platform, if you are using an AE platform, then see "Installing a BRBeans jar file on AE - overview" on page 9

1. Extract the file of SQL statements that was generated in "Deploying the code" on page 12 . If you opted for the default values in that step, this file can be found in the *WAS_HOME*\Enterprise\BRBeans\Deployed_BRBeans<DBtype>.jar file as META-INF/Table.ddl
2. Use the SQL statements to create the database tables. For example, if you are using DB2, launch a DB2 command window and type the following:
```
db2 connect to <database-name>
db2 -tf <fully-qualified-path-to-Table.ddl>
```

You have successfully created the database tables.

Proceed to "Creating a datasource" on page 13.

Creating a datasource

Follow this procedure when installing on an AEd platform, if you are using an AE platform, then see "Installing a BRBeans jar file on AE - overview" on page 9

1. Start the server (from a Command Prompt, type `startServer.bat/sh`).
2. Launch a browser, and start the Administrative Console (URL - http://localhost:9090/admin).
3. Type in a userid (and password if required) and click **Submit**.
4. In the left panel, expand the folders **Resources > JDBC Drivers** and click on driver that you want to use
5. In the right panel, specify a **Server Class Path** if it doesn't already exist.
6. Expand the driver folder and click on **Data Sources**.
7. To create a new data source, click the **New** button, and define at least the following:
    - **Name** (ex: brbds)
    - **JNDI Name** (ex: jdbc/brbds)
    - **Database Name** (ex: brbeans)

You have successfully created a datasource.

Proceed to "Installing the BRBeans <.jar file>." on page 13.

Installing the BRBeans <.jar file>.

Follow this procedure when installing on an AEd platform, if you are using an AE platform, then see "Installing a BRBeans jar file on AE - overview" on page 9

1. Start the server (from a Command Prompt, type `startServer.bat/sh`).
2. Launch a browser, and start the Administrative Console (URL -

http://localhost:9090/admin).

3. Type in a userid (and password if required) and click **Submit**.
4. In the left panel, expand the folders **Nodes > <node-name>**, and click on **Enterprise Applications**
5. In the right panel (Enterprise Applications), click the **Install** button.
6. In the **Application Installation Wizard**, choose the section which applies, based on where the `Deployed_BRBeans<DBType>.jar` file is located. Browse to the path of this jar file, and type in an Application name. Click **Next** to continue.
7. In **Mapping Roles to Users** window, select at least 1 user or group for each role (RuleManager and RuleUser). Alternatively, select the check-box for either **Everyone** or **All Authenticated**. Click **Next** to proceed.
8. In the **Binding Enterprise Beans to JNDI Names** window, make sure there are 3 EJBs listed with JNDI names.
9. In the **Mapping EJB References to Enterprise Beans** window, make sure there a 6 EJB references.
10. In the **Mapping EJB Jar Default Data Source References to JNDI Names** window, fill in the JNDI name of the datasource for the Deployed_BRBeans<DBType>.jar file (ex: jdbc/brbds).
11. In the **EJB Deploy** window, clear the box under the **Re-Deploy** option. (This is very important.)
12. Click **Finish**.
13. Save the configuration by clicking on **Save** at the top of the right panel.
14. Click **OK**.
15. Stop (`stopServer.bar/sh`) and restart the server.

You have successfully installed the BRBeans <.jar file>.

This is the last of the steps necessary to install Business Rules Beans for use with WebSphere AEd. To familiarize yourself with it, you can test it out with the **Samples** provided in “Part VII” on page 49.

Starting the BRBeans Rule Management Application

Follow the instructions below to start the BRBeans Rule Management Application.

1. Open a command prompt, and browse to `WAS_HOME\AppServer\Enterprise\bin`
2. For Windows platforms, type `rulemgmt.bat <properties-file>`
3. For UNIX platforms, type `rulemgmt.sh <properties-file>`
   Where properties-file is a fully qualified name of a file containing port, host, and the JNDI names used for the BRBeans EJBs. If you're using localhost and port=900, and installed BRBeans according to these instructions, use `WAS_HOME\AppServer\Enterprise\bin\brbeansDefaultProperties`. For a full definition of the contents of this file, refer to “The BRBeans Properties file” on page 47.

You have successfully started the WebSphere BRBeans Rule Management Application.

The Business Rules Beans framework has been installed and configured on your system. You may now begin using it. To familiarize yourself with it, you can test it out with the **Samples** provided in “Part VII” on page 49.

Part III: Rule Administration

Rule administration

In BRBeans, rule administration involves making changes to the set of business rules being used by applications. This can include any of the following activities:

- creating new rules that didn't exist before,
- deleting existing rules,
- creating a new rule with the same name as an existing rule to replace it,
- setting existing rules to expire when a change is to go into effect, or,
- moving rule changes from a development/test system to a production system.

There are two different interfaces that can be used for rule administration:

- “ **Rule Management Application:** ” on page 36An external user interface that allows users to manage rules interactively. It provides a very general purpose interface for managing rules where no assumptions are made about the content or implementation of the rules.
- “ **Rule Management APIs:** ” on page 34A programmatic interface that can be used by programmers writing code to manage rules or to customize an external user interface.

Rules can be administered in any way that makes sense for your application, but the BRBeans framework was designed with the following adminstrative paradigm in mind:

1. Understand the change in business behavior that is desired.
2. Inspect the application documentation (in particular information indicating where trigger point are located) to understand where the changes will need to be made in the system.
3. Inspect the corresponding set of existing business rules using the **Rule Management Application** (or your own custom management application, if you have one) to understand which rules need to change.
4. On a test system, use the **Rule Management Application** to create one or more new rules that implement the required new behavior. Give these rules the correct name so that they will be triggered by the appropriate trigger point. Also make sure that these new rules are currently in effect.
5. On the test system, withdraw (by setting the end date of the rule) all rules that are to be superseded.
6. Test the application to ensure it behaves as expected.
7. Using the Rule Exporter on the test system, export the new rules. Schedule them to become effective at the correct point in time.
8. Using the Rule Exporter on the test system, export the rules to be superseded. Set them to expire at the point in time at which the new rules come into effect.
9. Using the Rule Importer on the production system, import the new rules. This will create the new rules and schedule them to become effective at the date specified when they were exported.
10. Using the Rule Importer on the production system, import the rules to be superseded. This will put the new end date into the existing rules on the production system, thus setting them to expire on the specified date.

Part IV: The BRBeans framework

The BRBeans framework - overview

The BRBeans Framework is comprised of the following components:

- **BRBeans Trigger Point Framework:** The Trigger Point Framework forms the application writer's major interface to BRBeans. Your application invokes trigger point methods that find, filter, fire, and combine results of the appropriate rules at runtime. As provided, the Trigger Point Framework covers all of the common patterns of rule use and is designed so you can easily extend it to support new ones
- **BRBeans Rule Implementors:** Many times the logic contained in business rules occurs repeatedly. For example, a parameterized range check (is "x" between "a" and "b"?) is very common. For that reason, BRBeans provides a set of rule implementors that perform common algorithms that can be used in many of the rules that you create. In addition, since a BRBeans Rule Implementor is simply Java code that implements a small, simple interface, it's easy to write your own. The source for these rule implementors is installed with the samples.

- **BRBeans EJBs:** The WebSphere application server in which the BRBeans EJBs are installed is referred to as the BRBeans rule server. These EJBs provide the underlying implementation for the business rule persistence, the runtime facilities required to find and instantiate rules as necessary, and the management facilities required to inspect, clone, adjust, and otherwise maintain the persistent business rules. They are managed by the Trigger Point Framework and by the Rule Management APIs, so the programmer does not have to deal with them directly in the application. There is a set of Rule Management APIs available to simplify the interaction with the EJBs.
- **BRBeans Rule Management Application:** The BRBeans Rule Management Application is implemented as a Java Application that runs stand-alone, remotely or locally to the BRBeans rule server. It is used to create, update, expire, and delete BRBeans Rules, and can also be used to interactively import and export BRBeans Rules from/to XML.
- **BRBeans Rule Management APIs:** The Rule Management APIs provides a way for the application programmer to develop a rule management utility geared specifically for the domain of the application.
- **BRBeans Rule Importer and Exporter:** When deploying a rule driven application (moving it from a test system to a production system) it is often convenient to update the production rule system in batch mode, along with other artifacts of the application. The stand-alone, batch oriented Rule Importer tool provided with BRBeans does this.

Business Rule Beans - overview

The business rules defined by the BRBeans framework are organized in a straight-forward manner. Each rule is represented by an entity EJB that is used to persistently store information related to that rule. It is assigned an appropriate **rule name**, and stored in an appropriate **rule folder**. It is set up much like the file system on your computer's hard drive, and has many of the same characteristics. For example:

- Rules can be placed in folders based on any criteria the user desires.
- Two different rules can share the same name as long as they are stored in different rule folders.
- A rule folder can contain any number of rules and/or other folders.

In terms a naming scheme for the folders, it is recommended that the Java package naming convention be adhered to. That is, base the names on the domain name of the organization where the rules are developed. So, ACME's `isSeniorCitizen` rule's fully-qualified rule name, or full rule name, might be `com/acme/ageRules/isSeniorCitizen`. In this example, the `com/acme` path would be used by all rules developed by ACME, and the `ageRules` folder would be used to separate "age" rules from rules of other kinds. Note that the root folder has no name meaning that fully-qualified path names never start with a '/'.

A fully-qualified rule name consists the following:

- the full path of the folder followed by a '/'
- the name of the rule.

This fully-qualified rule name is used by a trigger point to identify the rule that is to be triggered. Note that when there is more than one rule with the same fully-qualified name, all the rules with that name that are currently in effect are triggered, and the results are combined using the combining strategy specified on the trigger point.

A business rule has a start date and an end date (see “Rule Attributes” on page 19) that together define the interval during which the rule is in effect (see “Rule States” on page 19). By default, trigger points will only trigger rules that are currently in effect based on the current date and time when the trigger point is called. This can be overridden by specifying a date on the trigger point. This date is referred to as the “As Of date” on page 47. If no start

date is specified, then the rule is not valid, and will not be found by trigger points. Conversely, if no end date is specified, then the rule will never expire. Dates and times with a precision of one second can be assigned using the “ **Rule Management Application** ” on page 36.

Rule States

Rules can be in any one of these four states at any particular time:

- **scheduled:** The rule is scheduled to become effective (its start date is in the future), and will not be found by current trigger points.
- **in effect:** The rule is currently in effect and can be found by trigger points.
- **expired:** The rule is no longer in effect (the end date is in the past), and will not be found by trigger points.
- **invalid:** The rule is not correctly defined and will not be found by trigger points.

Typically, only those rules which are "in effect" are found by the BRBeans runtime. This behavior can be overriden by setting an `asOfDate` on the TriggerPoint object, which will then execute "as if" the current date is the given date.

When a Rule is first created, it is marked as "ready for use" and will be found when firing Rules. If the Rule is not complete and you don't want it to be found by BRBeans, then mark the Rule using one of the following:

- Use the method `setReady(false)` in the **Rule Management APIs**
- Use the **Rule Management Application** to mark the rule as not ready

Rule attributes

Rule Name

Assign a name for the rule that is appropriate to its business context. Two different rules can share the same name as long as they are stored in different rule folders.

Rule Folder

The folder that contains the rule.

Start Date

This is the date and time at which the rule goes into effect. Prior to this time, it will not be found by trigger points. Together with the end date, the start date defines a period of time during which the rule is effective. A rule with no start date specified is not a valid rule and will not be found by trigger points.

End Date

This is the date and time at which the rule is no longer effective. After this date and time the rule is no longer in effect and will not be found by trigger points. Together with the start date, the end date defines a period of time during which the rule is effective. A rule with no end date specified is valid and will never expire.

Ready

Indicates whether or not the rule is ready to be used. Rules which are not marked as ready will not be found by trigger points. This is intended to be an easy way to keep a rule from being used until it is completely defined or to temporarily turn a rule off without having to change the basic rule data such as start and end dates.

Java Rule Implementor Name

This is the fully package-qualified name of a Java class that implements the BRBeans RuleImplementor interface. The `fire` method of the class performs the function of the rule. BRBeans provides several pre-defined rule implementors or you can write your own. Refer to the “Rule Implementor Interface” on page 29for more information on both of these choices.

Initialization Parameters

This is an array of parameters that are passed to the rule implementor to initialize it. Each element in the array can be any Object. This can also be referred to as the rule data, which is the external data that may change over time. The initialization parameters defined for a rule are passed directly to the `init` method of the rule implementor when it is instantiated. Refer to the “Rule Implementor Interface” on page 29for more information on how rule implementors can use initialization parameters.

Firing Parameters

Normally firing parameters are simply the parameters passed on the trigger point when a rule is triggered. However, it is allowed to override these parameters by specifying parameters on the rule itself. This is where these overriding parameters are specified.

Firing Location

This specifies where the rule implementor for this rule will be instantiated and run. Three values are allowed:

1. **Local:** This option instantiates the rule implementor and runs it local to the trigger point (in the same JVM as the trigger point call). This would be on the client machine if the trigger point call is done there, or the server if the server part of an application makes a trigger point call. Use this option for the best performance since, once a rule is cached on the client, the entire triggering process can be performed locally without going to the server at all. The main disadvantage of this option is that the class files for the rule implementors need to be available on every client that can trigger rules.
2. **Remote:** This will instantiate the rule implementor and run it on the application server where the BRBeans EJBs are installed. When using this option at least one remote method call will always be required to trigger a rule since the trigger takes place on the server. The advantage is that the rule implementor class files only need to be available on the server.
3. **Anywhere:** This option will try to instantiate and run the rule implementor locally, and, if the class cannot be found, it will try to trigger it remotely.

Classification

For classified rules, this is the classification to which the rule applies. This is used when performing a situational trigger. Once a classification is computed for the situational trigger point, rules that apply to that classification are found and triggered. For more details see “Situational Trigger Point - an example” on page 26.

Classifier

Indicates whether or not this rule computes a classification. This is used where performing a situational trigger. A classifier rule is used to perform the first step of a situational trigger which is to compute a classification that will be used to find rules to deal with the situation. For more details see “Situational Trigger Point - an example” on page 26.

Dependent Rules

This is an array containing the fully-qualified names of the dependent rules of this rule. Dependent rules are rules that this rule can use when it is triggered. For more details see “Dependent Rules” on page 21.

Business Intent

This is a text description of the intent of this rule from the point of view of the business analyst. Any text string can be stored here.

Description

This is a text description of the rule at the programmer's level. Any text string can be

stored here.

Original Requirement

This is a text description of the initial business analyst requirement of this rule. It can be used to keep track of why this rule was originally created, for example to keep auditing records. Any text string can be stored here.

User-Defined Data

A user-defined text string can be stored here. The format and use of this data is completely determined by the user.

Primary Key

Every rule has a primary key to uniquely identify it in the database where the EJBs are stored. Normally a unique primary key is generated automatically when you create a new rule. However, you can use the rule management APIs to specify your own primary key, if desired.

Precedence

This is the relative priority of this rule. The default finding strategy uses this value to order the rules found in the database, from lowest to highest, when more than one rule is found for a particular trigger point. Rules are sorted numerically by precedence with the numerically lowest precedence first and the numerically highest precedence last.

Returning results from rules

In general, a rule can return any type of result that makes sense for the business purpose of the rule. The return type on the fire method is `java.lang.Object` so any Java object can be returned, including arrays. You cannot return a Java primitive since the results must be an Object, however you can return the Object form of the primitives. For example, you can return a `java.lang.Integer` instead of an `int`.

Dependent Rules

When a business rule triggers other business rules as part of its implementation, then the rules that are triggered are called **dependent** rules of the first rule. An example is the `RuleAND` rule implementor supplied with BRBeans. It uses two or more dependent rules, each of which is assumed to return a true or false value. When a rule with `RuleAND` as its implementor is triggered, each of its dependent rules performs a logical AND operation on all the returned results. The result of this AND operation is returned as the result of the top-level rule.

Dependent rules are specified in the attributes of the top-level rule where the fully-qualified name of each dependent rule is listed. When the top-level rule is triggered, an array of dependent rule names is passed to the rule implementor's `init`method. They are stored here until they are triggered by the `fire` method. Note that the BRBeans framework does not ensure that the dependent rules specified in the EJB are actually triggered.

Dependent rules can be nested within other dependent rules. In other words, a dependent rule of some particular rule can have its own dependent rules which, in turn, can have their own dependent rules, and so on. The BRBeans framework does not place any restriction on the number of levels that dependent rules can be nested. The only practical restriction is the complexity of the rule set that is built up when dependent rules are nested many levels deep.

BRBeans Rule Folders

Rule folders are similar to the directories that divide a computer's hard drive in that they split a large number of files into conceptual units. The rule folder adds its path to the fully qualified rule name, and allows two names with the same name to be stored in separate folders effectively avoiding name collisions. Like the directories on a hard drive, a rule folder

can contain any number of rules of rule folders.

Although you can name the folders whatever you deem appropriate, it is recommended that you follow the Java package naming convention. That is, base the names on the domain name of the organization where the rules are developed. So, ACME's `isSeniorCitizen` rule's fully-qualified rule name, or full rule name, might be `com/acme/ageRules/isSeniorCitizen`. In this example, the `com/acme` path would be used by all rules developed by ACME, and the `ageRules` folder would be used to separate "age" rules from rules of other kinds. Note that the root folder has no name meaning that fully-qualified path names never start with a '/'.

When using the Rule Management APIs, a rule folder contains instances of `IRules`, which are also referred to as "rules". To begin working with rules, get the root rule folder by using the method `getRootFolder` on class `RuleMgmtHelper`. From the root rule folder you can add, delete, and retrieve folders and rules using methods on this interface.

Trigger Point Framework - overview

A trigger point is simply the location in a method of an object at which externalized business rules are invoked. Proper placement of trigger points can add substantially to the flexibility and speed with which a business application adapts to new business practices.

Wherever a trigger point is placed in user-written code, the BRBeans trigger point framework needs to do the following:

1. Assemble the parameter list to send to the rules.
2. Find the potential rules that apply.
3. Optionally, filter out any rules which do not apply.
4. Fire the rules in the filtered rule set.
5. Combine the results of the rule firings is some meaningful way

The application code that contains the trigger point needs to perform the following functions:

1. Establish a value for the target object. Usually the target object is the object in which the trigger point is encountered. The target object is one of the parameters passed to the fire method of the `RuleImplementor`.
2. Build the array of objects containing the runtime parameters needed to satisfy the trigger point's business purpose. This array is normally passed as one of the parameters of the `fire` method of the `RuleImplementor`. If firing parameters are specified on the rule itself, then those firing parameters are passed instead of the ones passed by the caller.
3. Invoke the `trigger(), triggerClassifier()`, or `triggerSituational()` method of the `TriggerPoint` class.
4. Catch and handle any exceptions that might occur as a result of firing the rules, else take action based upon the rule firing results.

The two simple trigger methods, `trigger` and `triggerClassifier`, perform their function in four steps:

1. find the rules
2. filter out those rules which are not desired
3. fire the remaining rules
4. combine the results and return to the caller

The complex trigger method, `triggerSituational` does this sequence of steps twice, the first step to find the classification which is fed into the second step. The second step

triggers rules which have the classification equal to the value returned in the first step.

How each of these steps is performed can be modified through various methods on the TriggerPoint object. The implementation of each step is defined by a **strategy** object. For more information on strategies, see “Using strategy objects to control triggers” on page 27.

Determining where to place a trigger point

Using Use Case Analysis to place trigger points.

Trigger points can be found during analysis by inspecting the use cases or user interaction scenarios that are typically developed as statements of requirement as input to the analysis process. A fragment of a use case is shown below:

The vehicle is entered into the system or chosen. The customer service representative attempts to locate the named driver in the system. If the driver is not found, she/he is added to the system and then picked.

Otherwise the found driver is just picked. If the vehicle is an auto, anyone between the ages of 16 and 75 can be picked as a driver. If the vehicle is a truck, only drivers 16 to 70 years old can be picked. And if the vehicle is a motorcycle, drivers 14 to 65 can be picked.

After the driver has been picked, a rate quote can be performed...

To identify potential trigger locations in use case analyses such as this one, look for certain **keywords** such as:

- "if X is in a special category Y" (e.g., “if the vehicle is a truck” above)
- “except when”,
- “unless”, or
- “depends on”

Using an Object Interaction Diagram to place trigger points

OIDs that are based on use cases can yield a number of observable patterns that can be used to identify trigger points fairly easy. Below are some of the rules to look for and where the trigger point might be placed:

- Validation of edits on create methods.
- Validation of edits on set methods.
- Referential integrity of edits on methods that set references.
- Cardinality checks at a consistency point (a point in time where all of the data is expected to be self con-sistent).
- Required fields checks at a consistency point.
- Cross field edits at a consistency point.
- Constraints or derivations that have a high potential for reuse (especially if the algorithm is complex) at any appropriate point.
- Constraints or derivations that a business desires to be consistent across applications (at any appropriate point).
- Constraints or derivations where the business wants to decouple the maintenance cycle for a rule from the maintenance cycle for the code (at any appropriate point).

How to place a trigger point

The `TriggerPoint` class is the primary interface of the BRBeans Trigger Point Framework, and is used to transfer control to the Trigger Point Framework in order to find

and fire those rules in the application's trigger point.

Perform the following steps to place a trigger point in your code:

1. Create an instance of the `com.ibm.websphere.brb.TriggerPoint` class. All rule triggers must be performed against an instance of this class. You should also set any desired strategies on the `TriggerPoint` instance.
2. Gather together the parameters to be passed on the trigger.

   For the simple `trigger()` and `triggerClassifier()` methods this includes the following:

   - **An optional target object.** This can be used to specify an object that is to be the target of the rule's algorithm. Whether or not this makes sense depends completely on the design of the rule implementor being used.
   - **The firing parameters for this rule trigger.** This is an array of runtime parameters needed by the rule to satisfy its business purpose. Note that any firing parameters defined on the rule itself will override whatever is passed here.
   - **Information identifying the rule(s) to be triggered.** Normally this is either a single String containing the name of the rule to be triggered or an array of Strings each element of which is the name of a rule to be triggered. However if a custom finding strategy is being used, this could be whatever information it needs in order to find the correct rules.

   The `triggerSituational` method differs in that it takes two sets of firing parameters and two sets of rule identification information: one set for the classifier rules and one for the classified rules.

3. Invoke the `trigger()`, `triggerClassifier()`, or `triggerSituational()` method of the TriggerPoint instance. This will actually trigger the rule(s).
4. Process the results of the triggered rule(s)

Examples of how to code a trigger point call are shown in “Types of Trigger Points - overview” on page 24.

For a detailed description of the trigger point programming interfaces refer the **Trigger Point Class** in the

Types of Trigger Points - overview

Follow these links for examples of three types of trigger points used in the BRBeans framework.

Simple Trigger Point - an example

A simple trigger point is used to trigger a rule or rules specified by name. This type of trigger point is used by invoking the trigger method on an instance of the `TriggerPoint` class. All rules with the specified name will be triggered and the results combined using the combining strategy specified on the `TriggerPoint` object. This type of trigger point only finds rules that are marked as not being classifiers.

The following shows an example of using a simple trigger point to trigger a rule named `isSeniorCitizen` (in the `com/acme/ageRules` folder) that determines whether a person is classified as a senior citizen based on the passed in age.

```
 ...
 // create an instance of TriggerPoint for triggering the rule and
specify that the
 // ReturnFirstCombiningStrategy is to be used to return only the first
result if
 // multiple rules are found.
 TriggerPoint tp = new TriggerPoint();
 tp.setCombiningStrategy(CombiningStrategy.RETURN_FIRST,
TriggerPoint.ALL_RULES);
 // define paramdter list that's passed to the rule
 Object [ ] plist = new Object[1];
 // define age of person to be tested
 Integer age = new Integer(64);
 // define name of rule to be fired
 String ruleName = "com/acme/ageRules/isSeniorCitizen";
 // define result of rule firing
 Object result = null;
 // initialize parameter list
 plist[0] = age;
 try {
         // fire "com/acme/ageRules/isSeniorCitizen" rule passing
paramdter list containing age.
         // Note: in this case the target object is not used and could
be null.
         result = tp.trigger(this, plist, ruleName);
         // put result into usable format.  A single result is returned
since we specified to use
         // the ReturnFirstCombiningStrategy.  By default an array of
results would be returned.
         boolean seniorCitizen = ((Boolean)result).booleanValue();
         // make use of result
         if( seniorCitizen ) {
                 ...
         }
 }
 catch(BusinessRuleBeansException e ) {
         // handle exception
         ...
 }
```

Classifier Trigger Point - an example

A classifier trigger point is identical to a simple trigger point except that it only finds rules marked as being classifiers. These are rules whose purpose is to determine what sort of business situation is present and return a classification string indicating the result. Usually these rules are used as part of a situational trigger point, but they can be triggered on their own too. This type of trigger point is used by invoking the `triggerClassifier` method on an instance of the `TriggerPoint` class.

The following shows an example of using a classifier trigger point to trigger a rule named `determineCustomerLevel` (in folder `com/acme/customerClassfiers`). This rule classifies customers into levels (Gold, Silver, and Bronze) based on their spending history.

```
 ...
 // create an instance of TriggerPoint for triggering the rule and
specify that the
 // ReturnFirstCombiningStrategy is to be used to return only the first
result if
 // multiple rules are found.
 TriggerPoint tp = new TriggerPoint();
 tp.setCombiningStrategy(CombiningStrategy.RETURN_FIRST,
TriggerPoint.ALL_RULES);
 // define paramdter list that's passed to the rule
 Object [ ] plist = new Object[1];
 // information about the customer to be checked is stored in this
object
 Customer cust = ...;
 // define name of rule to be fired
 String ruleName =
"com/acme/customerClassifiers/determineCustomerLevel";
 // define result of rule firing
 Object result = null;
 // initialize parameter list
```

```
 plist[0] = cust;
 try {
         // fire "com/acme/customerClassifiers/determineCustomerLevel"
rule passing parameter
         // list containing the customer to be checked.
         // Note: in this case the target object is not used and could
be null.
         result = tp.triggerClassifier(this, plist, ruleName);
         // put result into usable format.  A single result is returned
since we specified to use
         // the ReturnFirstCombiningStrategy.  By default an array of
results would be returned.
         String customerLevel = (String) result;
         // make use of result
         if( customerLevel.equals("Gold") ) {
                 ...
         } else if ( customerLevel.equals("Silver") ) {
                 ...
         } else if ( customerLevel.equals("Bronze") ) {
                 ...
         } else {
                 ...
         }
 }
 catch(BusinessRuleBeansException e ) {
         // handle exception
         ...
 }
```

Situational Trigger Point - an example

A situational trigger point is used when the rule(s) to be triggered depend on the business situation. This example evaluates a customer's past purchasing history to place them into one of three levels (Gold, Silver, or Bronze) which in turn determines how much of a discount they receive.

To use a situational trigger point to handle this case, it is first necessary to define four rules:

- one **classifier** rule to determine which of the three levels the customer falls into, and
- three **classified** rules to determine the actual discount to offer.

All of the classified rules have the same name and each is marked as applying to one of the three customer levels by specifying the level in its classification attribute. For example, the rule to determine the discount for a Gold level customer will contain the string "Gold" in its classification attribute.

The situational trigger point then proceeds in two phases:

1. Find the specified classifier rule and trigger it to generate a classification string.
2. Find the rules with the same name of the classification string returned in the first step.

These rules are then triggered to produce the final result, in this case the discount to offer.

The following shows an example of using a situational trigger point to handle the case described above.

```
 ...
 // create an instance of TriggerPoint for triggering the rule and
specify that the
 // ReturnFirstCombiningStrategy is to be used to return only the first
result if
 // multiple rules are found.
 TriggerPoint tp = new TriggerPoint();
 tp.setCombiningStrategy(CombiningStrategy.RETURN_FIRST,
TriggerPoint.ALL_RULES);
 // define parameter list that's passed to the classifier rule
 Object [ ] classifierPlist = new Object[1];
 // define parameter list that's passed to the classified rule
 Object [ ] classifiedPlist = new Object[1];
 // information about the customer to be checked is stored in this
object
 Customer cust = ...;
 // define name of classifier rule to be fired
```

```
 String classifierRuleName =
"com/acme/customerClassifiers/determineCustomerLevel";
 // define name of classified rule to be fired
 String classifiedRuleName =
"com/acme/discountRules/determineDiscount";
 // define result of rule firing
 Object result = null;
 // initialize parameter lists
 classifierPlist[0] = cust;
 classifiedPlist[0] = cust;
 try {
        // fire the rules to get the discount to offer
        // Note: in this case the target object is not used and could
be null.
        result = tp.triggerSituational(this, classifiedPlist,
classifierPlist, classifiedRuleName, classifierRuleName);
        // put result into usable format.  A single result is returned
since we specified to use
        // the ReturnFirstCombiningStrategy.  By default an array of
results would be returned.
        Float discountToOffer = (Float) result;
        // make use of result
        ...
 }
 catch(BusinessRuleBeansException e ) {
        // handle exception
        ...
 }
```

Using strategy objects to control triggers

Strategy objects are used to alter `TriggerPoint` functions.

Recall that the two simple trigger methods, `trigger()` and `triggerClassifier()`, perform their function in the following sequence:

1. find the rules
2. filter out those rules which are not desired
3. fire the remaining rules
4. combine the results and return to the caller

And that the complex trigger method, `triggerSituational()` does this sequence of steps twice, the first step to find the classification to feed into the second step.

Default strategy objects are already defined for each of the four `TriggerPoint` steps, and they are used if none are specified explicitly. For each of these steps, there are at least two strategy objects used, one for triggering classifier rules, and one for triggering non-classifier rules. For the filtering step, there are actually three pairs of strategies which are used, based on the number of rules which the finding strategy returns (zero, one, or multiple).

While the sheer number of strategies which are available can be intimidating (twelve different strategy classes can be set), typically very few will need updating, and in reality most users will only modify the filtering strategies or the combining strategies.

A number of pre-defined strategy objects are provided that should be adequate for the majority of cases. The Java classes for these strategy object are defined in package `com.ibm.websphere.brb.strategy`. Static constants are also defined in the interfaces for the various strategies to allow easy access to instances of the strategy classes to set them on the `TriggerPoint`.

It is also possible to write your own strategy class if the supplied ones don't perform the function you need. See “Writing Your Own Strategies” on page 45for more details.

Finding strategy

The job of the `FindingStrategy` is to access the datastore and return those rules which meet the search criteria specified. There are two `FindingStrategy` classes provided by BRBeans:

- `DefaultClassifierFindingStrategy`, and
- `DefaultNonClassifierFindingStrategy`

Both of these strategies perform a case-sensitive search for Rules marked ready that match the given search criteria. Results are ordered by precedence from highest to lowest (the first rule in the array has the numerically smallest precedence, the next rule has the next smallest precedence, etc.). If no rules are found, then an empty array is returned. The former strategy only returns classifier rules (`classifier=true`) and the latter only returns non-classifier rules (`classifier=false`).

These default strategies are used automatically by the `TriggerPoint`. There is no need to call `setFindingStrategy` to use these strategies. Instances of these two default finding strategies are stored in static constants defined on the `FindingStrategy` interface.

Filtering strategy

The job of the `FilteringStrategy` is to take the list of rules which were found by the `FindingStrategy` and filter out those rules which should not be fired. There are three sets of filtering strategies used in `TriggerPoint:`

1. strategy for **zero** rules found
2. strategy for **one** rule found
3. strategy for **multiple** rules found

A different strategy can be used for each of these scenarios, along with different strategies for classifier and non-classifier rules. The zero rules strategy is invoked if no rules are found by the finding strategy, the one rule strategy is invoked if exactly one rule is found, and the multiple rules strategy is invoked if more than one rule is found.

BRBeans provides several filtering strategies that can be used:

- **Accept Any** - any and all rules found are utilized (this is the default).
- **Accept One** - exactly one rule is expected.
- **Accept First** - only the first rule found is utilized.
- **Accept Last** - only the last rule found is utilized.

Instances of these filtering strategies are stored in static constants defined in the `FilteringStrategy` interface. You can use these for setting the strategies on a `TriggerPoint`.

As an example, here is one common way to use filtering strategies. Say you want to ensure that exactly one rule is found on a `TriggerPoint` call. You would set all three strategies (zero rules, one rule, and multiple rules) for this `TriggerPoint` to `FilteringStrategy.ACCEPT_ONE`. This strategy throws an exception if the number of rules is not exactly one. The following sequence of method calls would accomplish this for `TriggerPoint tp`:

```
 tp.setNoRulesFilteringStrategy(FilteringStrategy.ACCEPT_ONE,
TriggerPoint.ALL_RULES);
 tp.setOneRuleFilteringStrategy(FilteringStrategy.ACCEPT_ONE,
TriggerPoint.ALL_RULES);
 tp.setMultipleRulesFilteringStrategy(FilteringStrategy.ACCEPT_ONE,
TriggerPoint.ALL_RULES);
```

Firing strategy

The firing strategy takes the rules which were found by the `FindingStrategy`, (possibly modified by the `FilteringStrategy`), fires them each in order, and returns an array containing the results of each rule.

A single default `FiringStrategy` is provided by BRBeans, as all types of rules are fired in the same way. This implementation takes each rule in order and performs the following steps:

1. Determines what firing parameters to pass to the rule. If there are no firing parameters specified for this rule, uses the firing parameters passed on the `TriggerPoint` call. Otherwise uses the firing parameters specified in the rule in place of the parameters passed on the `TriggerPoint` call.
2. Calls the fire method on the rule, passing the firing parameters from the first step.

Unexpected exceptions result in an `BusinessRuleBeansException` being thrown that contains the original exception.

Combining strategy

The job of the `CombiningStrategy` is to take the results of the rules which were fired by the `FiringStrategy` and to combine them to form a reasonable result to the `TriggerPoint` caller. BRBeans provides several combining strategies to be used in applications:

- **Return All** - Return results from all rules fired in an array (this is the default)
- **Return First** - Return only the result from the first rule fired
- **Return Last** - Return only the result from the last rule fired
- **Return AND** - Return the logical AND of the results from all the rules fired. This strategy requires that all the results returned by the fired rules are either `ConstraintReturn` objects or `java.lang.Boolean` objects. An exception is thrown if this is not the case.
- **Return OR** - Return the logical OR of the results from all the rules fired. This strategy requires that all the results returned by the fired rules are either `ConstraintReturn` objects or `java.lang.Boolean` objects. An exception is thrown if this is not the case.
- **Throw Violation** - Throws a `ConstraintViolationException` containing all failed `ConstraintReturn` objects if any `ConstraintReturns` contain false. Otherwise just return a true `ConstraintReturn`.

Instances of these combining strategies are stored in static constants defined in the `CombiningStrategy` interface. You can use these for setting the strategies on a `TriggerPoint`. For example, the following method call sets the combining strategy on `TriggerPoint` tp to be the Return First strategy:

```
 tp.setCombiningStrategy(CombiningStrategy.RETURN_FIRST,
TriggerPoint.ALL_RULES);
```

Rule Implementors interface - overview

A BRBeans Rule is a persistent object that exists on the BRBeans Rule server. It has several persistent attributes, such as startDate, endDate, initParams, etc. One of the persistent attributes is `javaRuleImplementorName`, which is the name of its Rule Implementor.

A BRBeans RuleImplementor is an alogrithm written in Java that implements the BRBeans `RuleImplementor` interface. BRBeans provides a set of common implementations that can be used as the logic for specific user defined BRBeans Rules.

User defined `RuleImplementors` are written in Java code that implements the BRBeans `RuleImplementor` Interface. This code should be packaged in a jar file which appears in the CLASSPATH of the BRBeans Rule Server (for "remote" firing), or co-located with and in the CLASSPATH of the application(s) using it (for "local" firing). Typically, the

`RuleImplementor` will be in the application ear file.

How Rule Implementors are invoked

When a Rule is fired for the first time, the following sequence of events takes place:

1. an instance of the `RuleImplementor` class is created using the default constructor,
2. the `init` method is called, passing the initialization parameters defined for the Rule, and
3. the `fire` method is called to elicit the RuleImplementor's behavior.

It is guaranteed that the `init` method is called at least once before the `fire` method is called for the first time, although it may be called more than once. Once the `RuleImplementor` is instantiated, the Rule caches it so that it doesn't have to be created and initialized again the next time that Rule is fired. On subsequent fires, only the `RuleImplementor`'s `fire` method is called, not its `init` method. Thus the `RuleImplementor` is generally initialized only once but can be fired many times.

In some scenarios, the method `fire` may be called from multiple processes on the same instance of a `RuleImplementor`. This could happen if more than one client triggers the same rule at the same time. The implication of this is that the method `fire` should not change instance attributes of the implementor unless synchronization of the data being changed is performed.

Writing your own rule implementors

To write your own rule implementor, create a new Java class that implements the `com.ibm.websphere.brb.RuleImplementor` interface. This class must implement the following methods:

- **A default constructor**

  The class must have a default, no argument constructor so that it can be instantiated when a rule using it is triggered.

- **init**

  This method comes from the `RuleImplementor` interface and is called when the rule implementor is first instantiated. Its purpose is to perform an initialization needed by the rule implementor instance before it is actually fired. The following parameters are passed to the `init` method:

  - **The initialization parameters defined for the rule being triggered.** These can be any parameters needed to properly initialize the rule implementor instance. Often the initialization parameters consist of constants required by the algorithm. For example, when using a rule implementor that checks whether a number is greater than a threshold value, the threshold value would normally be passed as an initialization parameter. This will be null if there are no initialization parameters for the rule.
  - **An array of names of dependent rules for the rule being triggered.** Normally the rule implementor should store these to be used when the `fire` method is called. These depenent rules are intended to be triggered as part of the algorithm performed by the rule implementor. Refer to “Dependent Rules” on page 21 for more information. This will be null if there are no dependent rules defined for the rule.
  - **The user-defined data for the rule being triggered.** This data is completely defined by the user of BRBeans. BRBeans does not interpret this data in any way. This will be null if there is no user-defined data defined for the rule.

- **A reference to the actual rule being triggered.** This can be used to extract attribute values from the rule, if needed.

- **fire**

  This comes from the `RuleImplementor` interface. This method is called to actually perform the algorithm of the rule implementor. Any desired algorithm can be performed here. Normally some value is returned by the `fire` method. This value is ultimately returned as the result of triggering the rule. The following parameters are passed to the fire method:

  - **The TriggerPoint object which is being used to trigger the rule.** This is needed if the rule has dependent rules that the fire method needs to trigger.
  - **The target object for this particular trigger call.** This can be any object that can be thought of as the target of the rule. This may be null.
  - **A reference to the actual rule being triggered.** This can be used to extract attribute values from the rule, if needed.
  - **The firing parameters for this particular trigger call.** Normally these are the firing parameters passed by the code invoking the trigger point. However, these can be overridden by specifying firing parameters on the rule itself. Wherever they ultimately come from, these are the parameters that the rule implementor needs at runtime to perform its function. Normally these will be runtime variables that are to be processed in some fashion by the rule implementor. For example, when using a rule implementor that checks whether a number is greater than a threshold value, the number to be checked would normally be passed as a firing parameter. This will be null if no firing parameters are passed by the caller and none are defined on the rule itself.

- **getDescription**

  This comes from the `RuleImplementor` interface. The purpose of this method is to return a text string that describes the function of the rule implementor. This could be used, for example, to display on a user interface to help a user select what implementor to use. This method is currently not used by the BRBeans framework.

Pre-defined rule implementors

BRBeans supplies a number of pre-defined rule implementor classes that can be used in user-defined rules. (see). The Java source code for these rule implementors is supplied as BRBeans sample code in package `com.ibm.websphere.brb.implementor`. These can be used as examples when writing your own rule implementors.

BRBeans framework runtime

The BRBeans runtime code that is used to find and trigger rules is made up of two parts:

1. The part that runs on the **client** ("client" here meaning wherever a the trigger point is located). This consists of code that is used to do the following:
   - finds the specified rules,
   - decides where they should be triggered,
   - calls the fire method on all the rules, and
   - combines the results from the rules.
2. The part that runs on the **server** consists of the EJBs used to represent rules and rule

folders. These EJBs do the following:

- provide for business rule persistence, and
- provide query functions that the client part of the runtime can use to find rules to be triggered

Externalized business rules

The objects used to implement a business rule contain methods and attributes used by the BRBeans runtime and/or its administrative component. An externalized business rule is implemented as a pair of objects:

- a **Rule**, and
- a **RuleImplementor**.

The **Rule** is an entity EJB that stores all the persistent data for the business rule. This is the object that the trigger point framework code actually deals with directly. When a trigger point is invoked, the internal framework code performs a query to find the Rule object(s) representing the business rules to be triggered. Once the Rules are found, the framework code determines where the Rule is to be invoked, either local to the trigger point or remotely on the application server. It then invokes the `fire` method on either the Rule EJB itself (for remote triggering) or on a local copy of the EJB (for local triggering) to perform the function of the business rule.

The class name of the business rule's **`RuleImplementor`** is stored persistently in the Rule. The `RuleImplementor` is a transient object (not managed by the application server) which the Rule instantiates and then uses to do the actual work. When the `fire()` method is called on the Rule object, the Rule object combines its persistent set of values with the parameters it received on invocation to create the parameter list for the `RuleImplementor`, then it invokes `fire()` on the `RuleImplementor` with this parameter list. The actual execution of the `RuleImplementor` algorithm can take place either remotely (within the application server where the BRBeans EJBs are installed) or locally (within the JVM where the trigger point was called).

Runtime behavior

BRBeans runtime behaviour can best be described by giving a simple example of a trigger point selecting, executing, and then responding to the results of a business rule.

The first step in triggering a rule is for the trigger point framework to invoke a query method on the rule server to find the rules to be triggered. The main item used for the query is the fully-qualified rule name. Other items used in the query include start and end date, whether or not this is a classifier, the classification of the rule, and whether or not the `Rule` is marked "ready". This query will return zero or more Rules. If there is at least one `Rule`, the trigger point will assemble the data that will be sent as parameters to each `Rule`. The trigger point will then loop through the list of `Rule` invoking the fire method on each and passing the parameters. The results will be combined depending on the combining strategy being used.

When the trigger point framework invokes fire on a `Rule`, it instantiates the `RuleImplementor` and uses it to do the actual work (to execute the rule algorithm or test). Once it has arrived at a result, the `RuleImplementor` returns that result. For constraint rules (ones that arrive at a boolean true/false answer) the returned value is, by convention, a ConstraintReturn. A ConstraintReturn is a data structure indicating whether or not the constraint was satisfied, and if not, what went wrong. For derivation rules (ones that calculate a single, generally non-boolean value) the return value may be of any type. In the simplest case, the return value from each `RuleImplementor` is returned back to the trigger point where it is analyzed to determine what action to take.

Here is an example of what happens when a rule is triggered:

A `Rule` exists named `maxTruckGrossWeight`. The puspose of this rule is to check that the weight entered by a user for a particular truck does not exceed the maximum allowed value. This `Rule` contains an initialization parameter list consisting of a single value of 42000 (meaning a maximum gross weight of 42,000 lbs.). This `Rule` is bound to a `RuleImplementor` class called `MaxRuleImpl`. `MaxRuleImpl`, when invoked, tests the parameter it is passed against the initialization list value and returns a `ConstraintReturn`. The `ConstraintReturn` will be set to true if the passed parameter is less than or equal to the initialization value. Otherwise, a `ConstraintReturn` is set to false and some information is added describing which values were compared and why the test failed.

Here is what actually happens when this rule is triggered:

1. During the execution of the application, it reaches a point where it needs to check that the truck weight entered is valid. The application code invokes a simple trigger point passing the name of the rule to be triggered and a parameter list containing the entered weight of the truck.
2. The trigger point framework performs a query on the rule server to find a non-classifier rule with the specified name. It receives back an sequence of `Rule` objects. In this case this sequence contains one `Rule`, `maxTruckGrossWeight`.
3. The framework determines whether this rule is to be triggered locally or remotely. If local, the framework gets a local copy of the `Rule` object and calls the fire method on the copy. If remote, the framework calls fire on the EJB reference. The parameter list containing te enter weight is passed on the fire method.
4. The `maxTruckGrossWeight` rule (either the copy of the EJB itself) creates an instance of the rule implementor class, `maxRuleImpl`, if it does not already have one. When a new rule implementor instance is created, the rule calls its init method passing any initialization parameters defined for the rule. In this case the initialization parameter list contains the single value 42000. If the rule already has a rule implementor instance, it will use that one and will not call the init method again.
5. The `maxTruckGrossWeight` rule calls the fire method on the rule implementor instance. The firing parameters passed on the trigger point are passed to the rule implementor, possibly modified by any firing parameters defined in the rule itself. In this case the firing parametering are passed directly from the trigger point.
6. The `maxRuleImpl` returns a `ConstraintReturn` object to the `Rule` indicating the result of its comparison. This `ConstraintReturn` is returned to the trigger point framework and ultimately to the application.
7. The application checks the value in the `ConstraintReturn` and takes appropriate action.

Runtime exception handling

BRBeans defines one general exception class for exceptions that could be exposed to the user. All other BRBeans exceptions inherit from this class. The name of this class is `com.ibm.websphere.brb.BusinessRuleBeansException`. A **`BusinessRuleBeansException`** will generally be thrown when an unexpected error occurs within BRBeans. A `BusinessRuleBeansException` may have information in it about the original exception that caused the error. Doing a `printStackTrace` on the `BusinessRuleBeansException` will print out this information as well as the stack trace for the BusinessRuleBeansException itself. There are also methods on `BusinessRuleBeansException` to access the original exception programmatically, if desired.

BRBeans also defines a **`ConstraintViolationException`** , which extends `BusinessRuleBeansException`. A `ConstraintViolationException` is thrown if the

`ThrowViolationCombiningStrategy` is specified on the `TriggerPoint` and the rule returns a false value (either a `ConstraintReturn` or a Boolean).

Finally, BRBeans defines two exceptions, **`NoRuleFoundException`** and **`MultipleRulesFoundException`** , that are thrown by some of the pre-defined filtering strategies if an unexpected number of rules is found on a trigger point call. These two exception both extend `UnexpectedRulesFoundException` which, in turn, extends `BusinessRuleBeansException`.

Rule Management APIs

BRBeans provides a set of APIs to perform rule management tasks programmatically. These tasks include creating, deleting, and updating rules and folders. These APIs are provided to simplify the interaction with the BRBeans EJBs. Users should use these APIs to perform rule management tasks instead of coding directly to the EJB interfaces.

The rule management APIs consist of the classes in the `com.ibm.websphere.brb.mgmt` package. The main classes that users will be interested in are the following:

IRule

This is the interface used to access the object representing a business rule in BRBeans. It provides methods to read and update attributes of the rule, to delete the rule, and to make a copy of the rule. The methods to create rules are on the IRuleFolder interface since you must always create a rule into a particular folder. In the rule management APIs, any time you get a rule you have the option to get a reference to the EJB itself or to get a local copy of the data contained in the EJB. Regardless of which option is chosen, the IRule interface can be used to access the returned object. If a local copy is requested, it is possible to cast the returned object to an IRuleCopy. IRuleCopy extends IRule and adds a couple additional methods to those defined by IRule. See below for more details.

IRuleCopy

This is the interface used to access a local copy of the EJB representing a business rule. An object implementing this interface is returned from rule management API methods if you ask for a local copy of the rule. The main reason for requesting a local copy is performance. Calling a method on a local copy will be much faster than calling the method on the actual EJB. If you need to access several different rule attributes, this may make a big difference. Similarly, when updating a rule, all updates can be sent to the EJB in one method call instead of many. The individual set methods are called on the copy and then the updatePersistentRule method is called to actually send the updates to the EJB.

IRuleFolder

This is the interface used to access the object representing a rule folder. It provides methods to create, delete, and find rules and subfolders. It also provides methods to move and rename the folder, and to get the parent folder. The IRuleFolder representing the root folder is generally what you start with when performing rule management tasks. Once you have the root folder you can navigate up and down the folder hierarchy and access rules contained with the folders.

RuleMgmtHelper

This is a helper class intended to contain methods that are of general use for performing rule management tasks. Currently the only methods available are used to get the IRuleFolder representing the root folder. The root folder is normally the starting point for

performing rule management tasks.

IParameter

This is the interface used to represent an initialization or firing parameter stored in a Rule EJB. Every parameter has a user description and a value. Methods are provided on this interface to access these. Three classes are provided that implement the IParameter interface:

- **ConstantParameter:** This is the most common type of parameter. It represents a single constant value that is to be passed as an initialization or firing parameter.
- **MethodCallParameter:** This class represents a parameter whose value is determined by calling a method the target object. The method to call must be a public method and must take zero parameters. This is only used for firing parameters.
- **TriggerPointParameter:** This class represents a parameter which is retrieved from one of the trigger point firing parameters. This is mainly used for reordering the firing parameters passed on the trigger point. This is only used for firing parameters.

For more details on the rule management interfaces, including a number of coding examples, refer to the

Part V: Tools

Using the Rule Management Application (RMA) - overview

The Rule Management Application (RMA) is a simple tool that assists the user in the high level administration of rules and rule folders. This includes the capability to create, modify, delete, import or export rules or rule folders. The RMA tool can be used initially by the programmer to define rules interactively, and then by the domain analyst for rule management tasks.

The graphic user interface main window for the RMA is the Rules Browser.

The column on the left side of the Rule Browser window shows a nested hierarchy of all existing rule folders. Click on one of these folders to display the rules it contains. The names of these folders appear in the right column.

Navigate as you would in a typical file-management browser.

- Click the "+" icon to expand by one level; click the "-" icon to collapse it.
- Click a filename to highlight it; right-click it to launch a list of actions, or select an option from the main menu.

RMA is designed to be a very general purpose tool for interactive management of rules. Many users of BRBeans will want to write their own user interface that is tailored more specifically for the domain in which they work. For instance, a domain-specific user interface may be able to provide more help to the user in the task of managing rules than a general purpose tool such as RMA. Users wishing to write their own user interface can refer to the .

Creating rules

To create a rule using the Rule Management Application, proceed as follows:

1. In the **Rule Browser** window, select the folder where you want the new rule to

be created.

2. From the main menu, click **File > New > Rule**.

   In the **New Rule** properties window, use the following tabs to define the rule:

   - **General:** Use this tab to enter general information about the rule.
   - **Implementation:** Use this tab to define the manner in which the rule is implemented.
   - **Description:** Use this tab to define the purpose and intent of the rule.
   - **Dependent Rules:** Use this tab to specify the rules that the newly created rule will depend upon.
   - **Other:** Use this tab to to establish precedence, and enter information that is relevant to you, but doesn't fit into any other category.

3. To complete the creation of the rule, click **OK**.
   If there are any mandatory fields still undefined, you will either have to go back and give them a value, or make the rule unavailable for use (see **Status** in the **General** tab for more information on this).

Creating rule folders

To create a rule folder using the Rule Management Application, proceed as follows:

1. In the **Rule Browser** window, select the folder where you want the new folder to be nested.
2. From the main menu, click **File > New > Folder**.
   A new folder appears in the folder hierarchy in edit mode. Enter a folder name and hit the enter key.

Copying or moving rules or rule folders

Copy or move rules or rule folders either by cutting and pasting, or dragging and dropping.

- **Cutting and pasting.**Use menu commands (**Edit > Copy, Edit > Cut** and **Edit > Paste**) or keyboard commands (CTRL-C, CTRL-V and CTRL-X).
- **Dragging and dropping** Highlight the rule or rule folder you want to copy. Then press and hold the right mouse button, drag the cursor to the target location, and release. Select **Copy** or **Move** from the list.

Note: A rule can also be copied so that the copy will replace the existing rule at a specified date. This is referred to as a “ **Quick Copy** ” on page 37.

Using Quick Copy

Use **Quick Copy** to make a copy of a rule that will replace the existing one on a specified date.

For example, suppose that we have an "isSeniorCitizen" rule. Currently a person is considered a senior citizen if they are 62 years of age or older. Starting on January 1, 2002, we are going to change this to 65. Use Quick Copy to specify the new date, and change the age from 62 to 65. The current rule will be set to expire on the same date that the new rule will take effect with the new senior citizen age defined as 65.

Use **Quick Copy** from the **Rule Browser** or **Search Results** windows.

1. Select the rules you want to **Quick Copy**.

2. From the main menu, click **Edit > Quick Copy**.
3. In the **Quick Copy** window, specify how the copy will differ from the original as follows:

   follows:
4. Click **OK** to finish.

Note: The **Quick Copy** function should only be used for simple changes, and is not intended to be used in all cases.

Finding a rule

You can search for a specific rule using the RMA **Find** function.

1. To search through all rules in all folders:
   a. From the main menu of the Rule Browser, click **Edit > Find**.
   b. Use the tabs in the **Find Rules** window to determine your search criteria.
2. To search a specific folder:
   a. Right-click on the folder and select **Find** from the list.
   b. Use the tabs in the **Find Rules** window to determine your search criteria.

The results of your search are displayed in a **Search Results** window.

Deleting rules

You can delete rules from the **Rule Browser** or **Search Results** windows.

1. Select the rules you want to delete.
2. From the main menu, click **File > Delete.**
3. Click **Delete**, and then confirm the delete request.

Note: you cannot delete `com/ibm/websphere/brb/BRB CacheRule` as this rule is needed by the Business Rule Beans runtime.

Deleting rule folders

You can delete rule folders from the **Rule Browser** window.

1. Select the folder you want to delete.
2. From the main menu, click **File > Delete.**
3. Click **Delete**, and then confirm the delete request.

Note: you cannot delete the root folder, or any of the folders in the path `com/ibm/websphere/brb`.

Changing the Properties of a rule

To change to properties of a rule, perform the following steps in either the **Rule Browser** or **Search Result** windows:

1. Highlight the rule you wish to edit.
2. From the main menu, click **File >Properties**.

   In the **Rule Properties** properties window, use the following tabs to edit the rule's definition:

- **General:** Use this tab to edit general information about the rule.
- **Implementation:** Use this tab to edit the manner in which the rule is implemented.
- **Description:** Use this tab to edit the purpose and intent of the rule.
- **Dependent Rules:** Use this tab to edit the list of rules that the newly created rule will depend upon.
- **Other:** Use this tab to to establish precedence, and enter information that is relevant to you, but doesn't fit into any other category.

3. To complete the editing of the rule, click OK.
   If there are any mandatory fields still undefined, you will either have to go back and give them a value, or make the rule unavailable for use (see **Status** in the **General** tab for more information on this).

Importing a rule

You can use the **Rule Browser** window to import rules from an XML format.

1. In the main menu click **File > Import.**
2. In the **Import Rules** window, specify the file you want to import.
3. Click **OK**.

   Rules and rule folders are created as specified within the XML.

Exporting a rule

You can export rules from the **Rule Browser** or **Search Results** windows.

1. In the main menu click **File > Export.**
2. In the **Export Rules** series of windows, proceed as follows:
   a. In the **Specify Rules to Export** window, select the rule(s) that you want to export, and click **Next**.
   b. In the **Change Effective Dates On Exported Rules** window, alter the start and end dates of the rule if desired, and click **Next**.
   c. In the **Select File For Rule Export** window, chose a name and location for the exported rule.
3. Click **Export** to finish.

Renaming rules

You can rename rules from the **Rule Browser** or **Search Results** windows.

1. Highlight the rule you want to rename.
2. From the main menu, click **File->Rename.**
3. Type a new name and press the **Enter** key.

   To cancel the name change while still in progress, press the **Esc** key.

Renaming rule folders

You can rename rule folders from the **Rule Browser** or **Search Results** windows.

1. Place the folder name in edit mode by doing either of the following:

   Place the folder name in edit mode by doing either of the following:

2. Type a new name and press the **Enter** key.

   To cancel the name change while still in progress, press the **Esc** key.

Note: You cannot change the name of the root folder.

Formatting columns

To choose which columns you want shown in your Rule Browser window, perform the following steps in either the **Rule Browser** or **Search Results** windows:

1. From the main menu, click **View > Specify Columns**.
2. In the Specify Column window, proceed as follows:

   In the Specify Column window, proceed as follows:

Changing the date/time format

You can change the date and time format from either the **Rule Browser** or **Search Results** windows:

1. In the main menu, click **View > Specify Date/Time Format.**
2. In the **Specify Date/Time Format** window, choose one of the three radio button options:

   options:

3. When the example in the lower left of the window meets your needs, click **OK** to finish.

Using the RuleImporter and the RuleExporter tools

Rules can be imported or exported using the `RuleImport` or `RuleExport` classes respectively. The import takes place into a database from one or more XML documents. By contrast, the export takes place from the database into an XML document. The rules that are exported are determined by an XML document which is provided to the tool.

The rule importer and rule exporter functions can be invoked using the “ **Rule Management Application** ” on page 36 . The user interface in RMA provides some assistance in specifying the parameters required by the importer and exporter. Alternatively, the rule importer and rule exporter can be invoked from the command line using the following scripts:

- **For Windows platforms**
  - `ruleimporter.bat`
  - `ruleexporter.bat ()`
- **For UNIX platforms**
  - `ruleimporter.sh`
  - `ruleexporter.sh (f)`

Using the RuleImporter tool

```
ruleimporter <properties-file> <import-files> [options]
```

<properties-file>

The fully qualified name of a file containing the JNDI names of the BRBeans EJBs

for the rule set that is to be accessed. Refer to “The BRBeans Properties file” on page 47for a definition of the contents of this file. This parameter is required.

<export-list-files>

One or more fully qualified names of the files containing XML rule definitions to be imported. These files must contain XML in the format defined in `WAS_HOME\AppServer\Enterprise\bin\brb.dtd`. This XML format is also defined here. This parameter is required.

Options

-[v]erbose

Show verbose output while importing. This will show the rule definition of every rule that is imported.

-[t]est

Only parse the input files, do not create rules in the application server. This will ensure that there are no errors in the syntax of the rule definitions provided in the XML document. Combined with the -verbose option it can also be used to see exactly what rules will be imported.

-[u]pdate

When a rule in an input file has the same primary key as an existing rule, update the existing rule with values from the input file. If this option is not specified, then any rule with the same primary key as an existing rule will cause an error and that rule will not be imported.

-[c]ommiteach

Perform a commit after each rule is created rather than creating all rules in a single transaction. If this option is not specified, then all rules are created in a single transaction. This means that if any rule causes an error, the entire transaction will be rolled back and none of the rules will be imported. If -commiteach is specified, then when a rule causes an error only that rule will not be imported. Other rules will still be imported.

Using the RuleExporter tool

```
ruleexporter <properties-file> <export-list-files>
[options]
```

<properties-file>

The fully qualified name of a file containing the JNDI names of the BRBeans EJBs for the rule set that is to be accessed. Refer to “The BRBeans Properties file” on page 47for a definition of the contents of this file. This parameter is required.

<export-list-files>

One or more fully qualified names of files containing a list of rules and/or folders to be exported. These files must contain XML in the format defined in `WAS_HOME\AppServer\Enterprise\bin\brb-export-list.dtd`. This XML format is also defined here. This parameter is required.

Options

-[v]erbose

Show verbose output while exporting.

-[o]utput <file-name>

The name of the file to which rules should be output. This is where the XML rule definitions are stored. This is a required parameter.

Part VI: Advanced Topics

Improving performance - overview

The externalization of business logic using BRBeans has many benefits, but doesn't come completely without a cost. Since every business rule is represented by an EJB, then, in the general case, every rule trigger is performed in two parts:

1. a query is performed to find the EJBs representing the rules to be triggered,
2. a remote method call is performed on the EJB to actually trigger the rule.

Since both of these steps require going to the server, this can get rather slow. (There is also a third remote method call that is made to determine whether the rule is to be fired locally or remotely.)

This section documents the following ways to improve performance:

Caching to improve performance

The BRBeans framework incorporates a cache on the client side, i.e. wherever the trigger method on the TriggerPoint object is called. This cache is scoped to the JVM in which the client is running so that any trigger calls performed in a particular JVM will use the same cache, and two triggers performed in different JVMs will use two different caches. The BRBeans cache caches the results of all queries performed to find a set of rules to be triggered. The next time a trigger is performed in that JVM with the same rules specified, the rules will be found in the cache and the query will not actually require going to the server.

Once the rules are found in the cache they are triggered, either locally or remotely, depending on how they were defined. If a rule found in the cache is specified to be triggered locally, then the entire trigger process for that rule is performed on the client. No calls to the server are required. Even if the rule is specified to be triggered remotely, finding the rule in the cache eliminates one call to the server since the query does not have to be performed on the server.

The BRBeans cache can improve performance greatly, however it has one disadvantage: changes made to rules are not recognized immediately.

When a change is made to a rule on the server there is no way to inform all the potential clients that something has changed and that they therefore may need to refresh their caches. This effectively means that the client cache must check periodically to see if

anything in the persistent rule data has changed. This is implemented by associating a polling frequency with the cache. This polling frequency specifies an interval of time that the cache will wait before checking to see if anything has changed. The next time a trigger is performed after a polling interval has passed, the cache will check to see if any changes at all have been made to the persistent rule data stored on the server. If no changes have been made, then the cache is not refreshed. If any changes at all have been made, the entire cache is cleared so that the changes will be picked up. Thus changes to the rules are only picked up by the cache after a polling interval has passed.

The default polling frequency is 10 minutes. The user can change this value by changing the single initialization parameter specified for the special rule named `com/ibm/websphere/brb/BRB CacheRule`. The value for this initialization parameter has the following format:

- `hh:mm:ss`.

  where hh stands for hours, mm stands for minutes, and ss stands for seconds.

  Thus the default of 10 minutes is specified by a value of `00:10:00`. To specify a polling frequency of, for example, 1 hour, 30 minutes, specify `01:30:00`

. Note that when this value is changed it will not take effect until the previous polling interval has passed. Thus, if the previous polling interval is set to 24 hours and the polling frequency is changed to 1 hour, the new frequency will not take effect until the previous 24 hour polling interval passes. The only other ways to get the new frequency to take effect are to restart the client (since this will cause the cache to be reinitialized from scratch) or to have the client code call the `refreshCache` method on the `TriggerPoint` object. If there is more than one client JVM performing triggers, this has to be done for each client since each JVM has its own cache. Note that there is only one BRB `CacheRule` and this rule applies to all clients. There is no way to set different polling frequencies for different clients.

Caching can be disabled for a particular `TriggerPoint` object using the `disableCaching` method. After `disableCaching` is called any triggers performed using that `TriggerPoint` object will not use the cache. Triggers performed using other `TriggerPoint` objects are not affected.

Using servlets to improve performance

Another way that performance can be improved is to have all "client" code that triggers rules run in a servlet. This assumes that the servlet is running on the same physical system as the EJB server where the BRBeans EJB are installed. This way when remote calls are made to the EJB rule server, they are going to another JVM on the same machine and not going across a network to a different physical system. Of course, this becomes less important if the BRBeans cache hit ratio is high enough and most triggers are local. If this is the case, then most triggers will be completely local to the client code triggering the rules and it doesn't matter which machine it is running on.

Using indexes to improve performance

Creating an index over the database table that is used to store rules is an important way to improve the performance of rule queries. It is recommended that an index be created over the rulename column of the table containing the rules. This greatly improves the performance of rule-triggered queries that are looking for a rule(s) with a specific name. The index saves teh query the effort of searching every row in the table. Please refer to the documentation for your database for instructions on how to create an index.

Changing the firing location to improve performance

The BRBeans framework allows you to specify where a particular rule should be fired. This determines where the rule implementor for the rule is actually instantiated and invoked. There are three possible values for the firing location:

1. **Local:** Fires the Java rule implementor local to the client that fired the rule. This means in the same JVM in which the trigger was performed.
2. **Remote:** Fires the Java rule implementor on the server where the rules exist.
3. **Anywhere:** First tries to fire the Java rule implementor locally. If the Java rule implementor cannot be found, then it fires the Java rule implementor remotely. This is the default value.

For simple rule implementors that do not perform any server-intensive work, specifying **local** usually results in the best performance. This is true both with and without caching.

Without caching, the work done to fire a rule remotely involves the following:

1. finding the rule
2. determining whether the rule is to be fired locally or remotely
3. calling fire on the remote rule

Each of these three operations requires a remote call to the server.

With caching the work done to fire a rule locally involves the following:

1. finding the rule
2. determining whether the rule is to be fired locally or remotely
3. calling fire on a local copy of the rule

This requires only two remote calls.

With caching, local firing results in even more dramatic improvements since, if the rule is found in the cache, the entire rule firing process takes place locally with no remote calls. In fact, to get the full benefit of the cache, rules should be fired locally, (although remotely fired rules still benefit from the cache due to the elimination of the query on the server).

There may be some cases where a rule implementor must perform some work that requires much interaction with the server. In these cases it may actually be beneficial to have rules using this rule implementor defined to be fired remotely. This should make the server interaction performed by the implementor more efficient.

Note that, in addition to performance, maintenance must also be considered in relation to specifying a firing location. The rule implementor classes for rules that are defined to be fired locally must be present on any client system that tries to fire those rules. Otherwise the implementor cannot be instantiated when the rule is fired. This can result in maintenance problems when the rule implementors are changed since they must be updated on many different systems.

Writing your own strategies

The process of triggering a rule or set of rules is controlled by a set of strategy objects. Four strategies are used each time a rule is triggered:

1. **Finding strategy:** The finding strategy accesses the persistent datastore to find the set of rules matching the search criteria passed on the trigger call. The search criteria are based on the rule ID information passed on the trigger call. The set of rules found is passed to the filtering strategy.
2. **Filtering strategy:** The filtering strategy can change the set of rules that were found by the finding strategy. The set of rules returned is the set that will actually be fired by the firing strategy.
3. **Firing strategy:** The firing strategy fires the rules found by the finding strategy, possibly modified by the filtering strategy. It gathers up the results of the individual rules and these results are passed to the combining strategy.

4. **Combining strategy:** The combining strategy takes the results from firing the rules and combines them in some fashion to produce the final result of the trigger.

`TriggerPoint` object has its own set of strategies that can be changed independent of any other `TriggerPoint` object. There is a set of default strategies that are used by the `TriggerPoint` if none are explicitly set.

For each of the four strategies you are allowed to set different strategies for **classifier** rules and for **non-classifier** rules. The strategies set for classifier rules are to be used when the BRBeans framework is triggering a classifier rule. The strategies for non-classifier rules are used in all other cases.

It is also possible to set three different sets of filtering strategies:

1. one to be used if no rules are found
2. one to be used if exactly one rule is found
3. one to be used if more than one rule is found

This capability can be used to set up filtering strategies that will throw exceptions if the expected number of rules is not found.

Strategy classes must implement one of the strategy interfaces provided by BRBeans in the `com.ibm.websphere.brb` package:

1. `FindingStrategy`
2. `FilteringStrategy`
3. `FiringStrategy`
4. `CombiningStrategy`

The user is also allowed to write his or her own strategy implementations to perform special functions not performed by the predefined implementations. This should be done with care since part of the functionality of the BRBeans framework is actually being replaced when you write a custom strategy. One simple example of writing a custom strategy is creating a new firing strategy that logs every rule that is fired.

The basic requirement for a strategy implementation is that it implements the appropriate strategy interface.

The **filtering** and **combining** strategies are particularly simple. For these, a class should be created that implements either `FilteringStrategy` or `CombiningStrategy` and implements either the `filterRules` method (for `FilteringStrategy`) or the `combineResults` method (for `CombiningStrategy`) to perform the required functions. At runtime, an instance of the new class should be created and passed to the `TriggerPoint` object using the appropriate set method so that the new strategy is used when rules are triggered using that `TriggerPoint`.

The **finding** and **firing** strategies are somewhat more complicated to customize since they provide more function than the simple filtering and combining strategies. Default finding and firing strategy implementations are provided that define a general outline of the steps necessary to perform the function. We recommend subclassing these when customizing your own strategies, and then overriding the desired methods on the default implementation to provide the new behavior.

Refer to either the or the source code for the default implementations in `com.ibm.websphere.brb.strategy`for more details.

As Of date

Normally, a rule can only be triggered if it is "in effect" (see “Rule States” on page 19) as of the current date and time. By using the As Of Date method, you can trick the rules into triggering themselves prematurely. This is especially useful when you want to test a rule, see what effect a future change in rules or regulations may have on the overall framework, or see what past or future rates and/or discounts might be.

To set an "As Of Date", call the `setAsOfDate` method on the `TriggerPoint` object, and pass the date that you want to be used. To use the current date again, call `unsetAsOfDate` or call `setAsOfDate` and pass null for the date.

The BRBeans Properties file

Applications that use the BRBeans EJBs (this includes those that trigger rules or use the rule management APIs) must specify the JNDI names for these EJBs so that the code can find them at runtime. If the application is running in a J2EE client container, in a servlet, or on the application server itself (e.g. as part of another EJB), then these names have probably already been specified by the person who configured the application.

At runtime, the BRBeans code looks for a special Java property that identifies the name of the properties file. The default name of the Java property is `brbPropertiesFile`, but it can be specified on the command line as `-DbrbPropertiesFile=<file-name>`.

When an application attempts to reference BRBeans EJBs, the code will first look for the brbPropertiesFile Java property. If this property is specified, then the names listed in that file are used to find the EJBs, overriding any EJB references that were specified in the container (if the application is running in a container). If the property is not specified, then BRBeans attempts to use the EJB references specified in the container.

The host name and port number used to access the name server can also be set in this file. If these are not specified, then the name server used by the container in which the application is running is used. If the application is not running in a container, then localhost is used for the host name, and 900 is used for the port number.

The properties file must be in the following format (entries can be specified in any order):

```
host=<host-name-for-name-server>
port=<port-number-for-name-server>
RuleJndi=<JNDI-name-for-Rule-EJB>
RuleFolderJndi=<JNDI-name-for-RuleFolder-EJB>
RuleHelperJndi=<JNDI-name-for-RuleHelper-EJB>
```

A default properties file is shipped as `WAS_HOME\AppServer\Enterprise\bin\brbeansDefaultProperties`. This default file contains default names that are used in the `BRBeans.jar` file that is shipped with BRBeans. This file can be used if that jar file is installed without changing the names. Note that the file name still must be specified even if you want to use the default file. There is no file that is used automatically if the `brbPropertiesFile` property is not set.

The tools shipped with BRBeans (the Rule Management Application, the rule importer, and the rule exporter) all run outside of any container. Hence the JNDI names need to be specified when these tools are run. The scripts for these tools all require that a proerties file name be passed as a command line parameter. This name is then specified as the value for the brbPropertiesFile property when the tool is run.

Including BRBeans in your application

When you are ready to ship your application, you should include a BRBeans jar file in your

ear file. There are several of these jar files in the <WAS-HOME>\Enterprise\BRBeans directory, one for each supported database. Each name reflects the database type that it uses (ex: BRBeans**DB2**.jar). These jar files contain 3 EJBs, with the following JNDI names:

- brbeans/application/Rule
- brbeans/application/RuleFolder
- brbeans/application/RuleHelper

In your ear file, you should change these names to make them unique for your application. For example, if your application is called MyApp, you could change the first one to brbeans/MyApp/Rule, or possibly com/MyCompany/MyApp/Rule.

In your ear file, you will also need to define EJB references to these 3 EJBs. These should be defined in any module where a trigger...() method exists in one or more of its classes. You can do this using the **Application Assembly Tool**. The Name field should contain the following, corresponding to the EJBs listed above:

- ejb/com/ibm/ws/brb/Rule
- ejb/com/ibm/ws/brb/RuleFolder
- ejb/com/ibm/ws/brb/RuleHelper

The JNDI name on the Bindings tab should be the same as the JNDI names that you gave earlier to the EJBs.

Part VII: Samples

Business Rule Beans samples - overview

If you selected to install samples during the installation of WebSphere Enterprise Edition, the Business Rule Beans samples can be found in `WAS_HOME\Appserver\Enterprise\samples\index.htm` directory.

The BRBeans Simple Sample allows you to do 2 things:

- create 2 rules using the rule management APIs and,
- fire the 2 rules using `TriggerPoint.trigger`...() methods

For directions on how to install and run the samples, launch `...samples\BRBeans\SimpleSample\BRBeansSimpleSample.html`. The source for this sample is in `BRBeansSimpleSampleSource.jar` in the same directory.

In this sample, you will:

- configure a database for use with BRBeans
- install an application containing BRBeans
- run a client application to create 2 rules
- use the Rule Management Application to view the 2 rules that were created
- run a client application to trigger the 2 rules that were created.

The BRBeans Movie Samples demonstrate the capabilities of Business Rule Beans (BRBeans) Framework. These have a common theme which is www.moviestore.com, an e-business that sells movies. A complete working system is not provided but rather only the components of a rule-enabled application necessary for demonstration of basic trigger point patterns and variations in trigger point behavior.

The samples themselves utilize rule-enabled CMP beans to persistently store data and Session bean to keep the shopping cart contents. The interface to the user is in the form of servlets that are used to send input and requests and display results to/from the servlets. Additional version that utilizes interface html input form->servlet->jsp is also provided.

The following samples are available:

- Customer Sample
- Movie Sample
- Shopping Cart Sample
- Online Movie Store Sample

For directions on how to install and run the samples, launch `...samples\BRBeans\MovieSample\BRBeansMovieSample.html`. The source for this sample is in `BRBeansMovieSampleSource.jar` in the same directory.