Java Rule Engine API™
JSR-94

Java Community Process

http://java.sun.com/jcp/

Specification and Maintenance Lead:
Alex Toussaint, BEA Systems, Inc.

Technical Comments:
jsr-94-comments@jcp.org

Java Rule Engine API Specification ("Specification")
Version: 1.0
Status: JCP Final Review
Release: September 15, 2003

BEA SYSTEMS, INC. ("BEA") IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY IT, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

_____ for JavaTM Specification ("Specification")

Version: _____

Status: FCS

Release: [insert date]

2315 North First Street, San Jose CA, 95131

NOTICE; LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. BEA hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under BEA's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation, which shall be understood to include developing applications intended to run on an implementation of the Specification provided that such applications do not themselves implement any portion(s) of the Specification.

2. License for the Distribution of Compliant Implementations. BEA also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 3 below, patent rights it may have covering the Specification to create and/or distribute an implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality, and (b) passes the Technology Compatibility Kit for such Specification ("Compliant Implementation").

3. Reciprocity Concerning Patent Licenses.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b With respect to any patent claims owned by BEA and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against BEA that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by BEA and covered by the license granted under subparagraph, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against BEA that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

4. Definitions. For the purposes of this Agreement: "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying documentation provided by BEA which corresponds to the particular version of the Specification being tested.

BEA shall have the right to terminate this Agreement immediately notice if you fail to comply with any material provision of or act outside the scope of the licenses granted above.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of BEA or BEA's licensors is granted hereunder. Java is a registered trademark of Sun Microsystems, Inc. in the United States and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". BEA MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. BEA MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL BEA OR ITS BEAS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF BEA AND/OR ITS BEAS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend BEA and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your application or applet written to and/or Your implementation of the Specification; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide BEA with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant BEA a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable copyright license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

Contents

Figures

Java Rule Engine API

1 Expert Group Members

Company	Representative
ATG Inc.	Allan Scott
BEA Systems Inc.	Alex Toussaint
Fair Isaac Inc.	Johan Majoor
Fujitsu Ltd.	Frank McCabe
IBM Inc.	Rainer Kerth
ILOG SA.	Changhai Ke, Daniel Selman
Oracle Inc.	Mark Hornick
Sandia National Labs	Ernest Friedman-Hill
Silverstream Inc.	Gregg McMullin
Unisys Inc.	Sridhar Iyengar

2 Target Audience

The specification is aimed at software engineers using the API to implement rules-based applications and rule engine vendors building rule-engine implementations compliant with the specification.

3 Scope

The scope of the specification is to define a lightweight-programming interface that constitutes a standard API for acquiring and using a rule engine.

The scope of the specification specifically excludes defining a standard rule description language to describe the rules within a rule execution set.

The specification targets the J2SE platform.

The specification is compatible with JDK1.3.x (with optional packages) as well as JDK 1.4.x (unchanged)

The following items are in the scope of the specification:

- The restrictions and limits imposed by a compliant implementation.
- The mechanisms to acquire interfaces to a compliant implementation.
- The interfaces through which rule execution sets are invoked by runtime clients of a complaint implementation.
- The interfaces through which rule execution sets are loaded from external resources and registered for use by runtime clients of a compliant implementation.

The following items are outside the scope of the specification:

- The binary representation of rules and rule execution sets.
- The syntax and file-formats of rules and rule execution sets.

- The semantics of interpreting rules and rule execution sets.
- The mechanism by which rules and rule execution sets are transformed for use by a rule engine.
- All minimal system requirements required to support a compliant implementation.

4 Compliance

Compliance is of interest to the following audiences:

- Those designing, implementing, or maintaining JSR-94 implementations.
- Governmental or commercial entities wishing to procure JSR-94 implementations.
- Testing organizations wishing to provide a JSR-94 compliance test suite.
- Programmers wishing to port code from one complaint implementation to another.
- Educators wishing to teach JSR-94 compliant rule engines.
- Authors wanting to write about JSR-94 compliant rule engines.

Clients of a compliant JSR-94 implementation should understand that they will only get semantic interoperability between rule engines that implement semantically equivalent rule execution set evaluation cycles. Compile time compatibility and binary compatibility do not necessarily imply runtime compatibility of compliant implementations.

The text in this specification that specifies requirements is considered normative. All other text in this specification is informative, that is, for information purposes only. Normative text is further broken into required and conditional categories. Conditionally normative text specifies requirements for a feature such that if that feature is provided, its syntax and semantics must be exactly as specified.

If any requirement of the specification is violated, the behavior is undefined. Undefined behavior is otherwise indicated in the specification by the words undefined behavior or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe behavior that is undefined.

5 References

RuleML: http://www.dfki.uni-kl.de/ruleml/

Java Specification Requests: http://www.jcp.org/

- JSR-41: A Simple Assertion Facility
- JSR-47: Logging API Specification
- JSR-73: Data Mining API

"**RETE: A fast algorithm for the many pattern/many object pattern match problem**," Artificial Intelligence, Volume 19, Number 1, 1982, C.L. Forgy

Provides useful background information about the RETE algorithm, which is at the heart of several rule engine implementations.

Business Rules for Electronic Commerce: Project at IBM T.J. Watson Research

Provides useful background information.

Jess, The Java Expert System Shell

JESS is a rule engine and scripting environment written entirely in Sun's Java language by Ernest Friedman-Hill at Sandia National Laboratories in Livermore, CA. JESS can be licensed free of charge for academic use.

RFC 2119, Key Words for use in RFCs to Indicate Requirement Levels

http://www.ietf.org/rfc/rfc2119.txt

Java Logging API

http://java.sun.com/j2se/1.4/docs/guide/util/logging/overview.html

Java Security Architecture

http://java.sun.com/security/

JDK 1.4 Security

http://java.sun.com/j2se/1.4/docs/guide/security/index.html

6 Definitions

6.1 Rule Engine

The key underlying technology is the rule engine. A rule engine may be viewed as a sophisticated if/then statement interpreter. The if/then statements that are interpreted are called rules. The *if* portions of rules contain conditions such as `shoppingCart.totalAmount > $100`. The *then* portions of rules contain actions such as `recommendDiscount(5%)`. The inputs to a rule engine are a rule execution set and some data objects. The outputs from a rule engine are determined by the inputs and may include the original input data objects with possible modifications, new data objects, and side effects such as `sendMail('Thank you for shopping')`.

There are many differences between rule engines, and the term is used extremely loosely across the software industry. Typically, common features:

- Promote declarative programming by externalizing business or application logic.
- Include a documented file-format or tools to author rules and rule execution sets external to the application.
- Act upon input objects to produce output objects. Input objects are often referred to as *facts* and are a representation of the state of the application domain. Output objects are often referred to as *conclusions* or *inferences* and are grounded by the application into the application domain.
- The rule engine may execute actions directly, which affect the application domain, input objects, the execution cycle, rules, or the rule engine.
- The rule engine may merely create output objects, delegating the interpretation and execution of the output objects to the caller.

One of the most common classes of rule engines is the forward-chaining rule engine. Forward-chaining rule engines implement an execution cycle that allows the action of one rule to cause the condition of other rules to become met. In this way, a cascade of rules may become activated and each rule action executed. Forward-chaining rule engines are suitable for problems that require drawing higher-level conclusions from simple input facts. Forward-chaining rule engines are often implemented using a variant of the RETE-algorithm.

While the specification recognizes the importance of forward-chaining rule engines, it does not mandate an execution cycle or the semantics of executing a rule execution set. The specification defines a lightweight API that could be implemented by a wide variety of rule engines. It is expected, and desired, that non forward-chaining rule engines will also implement the APIs of this specification.

6.2 Rule

A rule is typically composed of two parts: a condition and an action. When the condition is met, the action is executed. This specification does not address the structure of rules, as there are considerable differences between vendors, and differences often relate to the requirements of different types of rule engines and execution algorithms. For the purposes of this specification, a rule merely exposes basic metadata, such as a name and a description.

6.3 Rule Execution Set

A rule execution set is a collection of rules. The specification does not define the structure of a rule execution set other than to say that a rule execution set is composed of a collection of rules. Additionally, a rule execution set also exposes basic metadata such as name and description.

6.4 Rule Session

A rule session is a runtime connection between a client and a rule engine. A rule session is associated with a single rule execution set. A rule session may consume rule engine resources and must be explicitly released when the client no longer requires the rule session.

6.5 Stateful Rule Session

A stateful rule session allows a client to have a prolonged interaction with a rule execution set. Input objects can be progressively added to the session and output objects can be queried repeatedly.

6.6 Stateless Rule Session

A stateless rule session provides a high-performance and simple API that executes a rule execution set with a List of input objects. Stateless rule session methods are idempotent.

7 Document Conventions

7.1 Key Words

The key words "`MUST`", "`MUST NOT`", "`REQUIRED`", "`SHALL`", "`SHALL NOT`", "`SHOULD`", "`SHOULD NOT`", "`RECOMMENDED`", "`MAY`", and "`OPTIONAL`" in this document are to be interpreted as described in [RFC 2119].

7.2 Typography

Code Font:

```
RuleRuntime ruleRuntime = RuleServiceProvider.getRuleRuntime();
```

Code Notation Font:

```
// this is a code notation for the Foo Class.
```

Inline class, interface, method, or package references:

The `Foo` class, defined in the `org.bar` package defines the `isComplete` method and implements the `Bar` interface.

8 Acronyms and Abbreviations

Acronym	Abbreviation
J2SE	Java 2 Standard Edition
JCA	Java Connector Architecture

9 Introduction

The specification defines a Java API for rule engines. The API prescribes a set of fundamental rule engine operations. The set of operations is based on the assumption that most clients need to be able to execute a basic multiple-step rule engine cycle that consists of parsing rules, adding objects to an engine, firing rules, and getting resultant objects from the engine.

This specification targets the J2SE platform.

A primary input to a rule engine is a collection of rules called a rule execution set. The rules in an execution set are expressed in a rule language. This specification does not prescribe a rule language but is focused on facilitating runtime interoperability between rule engines.

The specification strives to be inclusive across rule engines and does not mandate the semantics of the rule execution cycle or a rule language. The specification errs on the side of simplicity and generality over mandating specific implementation, deployment, or management methodologies. The specification supports rule engines that are running wholly within the caller's JVM as well as rule engines that are proxy rule engine requests to remote JVMs.

The authors acknowledge that the generality of the specification comes at the price of semantic interoperability of implementations and expects that future revisions of the specification will impose additional semantic requirements on different classes of rule engines. This approach mirrors that of the JCA specification, where compile time compatibility does not necessarily infer similar runtime behavior between vendor Resource Adapter implementations.

The authors welcome suggestions from the JCP community and the Java community at large on these issues.

9.1 Rationale

This specification addresses the community need to reduce the cost associated with incorporating business logic within applications and the community need to reduce the cost associated with implementing platform-level business logic tools and services.

Dissimilar vendor-specific API specifications exist. However, the differences between these specifications are significant enough to cause costly difficulties for application builders, platform vendors, and software architects.

9.2 Goals

The goals of the specification are to:

- Facilitate adding rule engine technology to Java applications.
- Increase communication and standardization between rule engine vendors.

- Encourage the creation of a market for third-party application and tool vendors through a standard rule engine API.
- Facilitate embedding rule engine technology in other JSRs to support declarative programming models.
- Promote independence of client code from J2SE environment.
- Make Java applications more portable from one rule engine vendor to another.
- Provide implementation patterns for rules-based applications for the J2SE platform.
- Support rule engine vendors by offering a harmonized API that meets the needs of their existing customers and is easily implemented.

10 Architecture

The interfaces and classes defined by the specification are in the `javax.rules` and `javax.rules.admin` packages. The `javax.rules` package contains classes and interfaces that are aimed at *runtime clients* of the rule engine. The runtime client API exposes methods to acquire a rule session for a *registered* rule execution set and interact with the rule session. The administrator API exposes methods to load an execution set from these external resources: URI, `InputStream`, XML `Element`, binary abstract syntax tree, or `Reader`. The administrator API also provides methods to register and unregister rule execution sets. Only registered rule execution sets are accessible through the runtime client API.

A packaging separation between the runtime client API and the administrator API was made to reinforce the distinction between executing a rule execution set that has been previously loaded and registered into the runtime environment by an administrator, and the dynamic loading and execution of external resources. The later actions can only be performed using the classes and interfaces in the `javax.rules.admin` package.

The distinction between the runtime and admin packages allows a more fine grained control of the user population; for example, some users may be allowed to execute rules but not to administer them.

10.1 Runtime API

The runtime API for the specification is defined in the `javax.rules` package. The high-level capabilities of the runtime API are:

- Acquire an instance of a rule engine vendors `RuleServiceProvider` interface through the `RuleServiceProviderManager` class.
- Acquire an instance of the `RuleRuntime` interface through the `RuleServiceProvider` class.
- Create a `RuleSession` through the `RuleRuntime`.
- Get a `java.util.List` of registered URIs.
- Interact with an acquired `RuleSession`.
- Retrieve metadata for a `RuleSession` through the `RuleExecutionSetMetadata` interface.
- Provide an `ObjectFilter` interface to filter the results of executing a `RuleExecutionSet`.
- Use `Handle` instances to access objects added to a `StatefulRuleSession`.

RuleServiceProviderManager

The `RuleServiceProviderManager` class allows J2SE runtime clients to retrieve a `RuleServiceProvider` implementation for a given rule engine vendor. The `RuleServiceProviderManager` class provides methods to allow a rule engine vendor's `RuleServiceProvider` implementation to be registered with a URL in a manner similar to the JDBC classes `Driver` and `DriverManager`.

`RuleServiceProvider` implementers should make efforts to ensure uniqueness of their registration URL. The recommended convention is to use a name within the Internet domain namespace (or Java package namespace) of the `RuleServiceProvider` implementer.

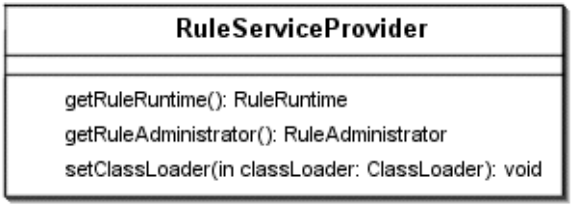
For example:

```
Class.forName( "org.jcp.jsr94.ri.RuleServiceProvider" );
```

```
RuleServiceProvider serviceProvider =
RuleServiceProviderManager.getRuleServiceProvider (
"org.jcp.jsr94.ri.RuleServiceProvider" );
```

RuleServiceProvider

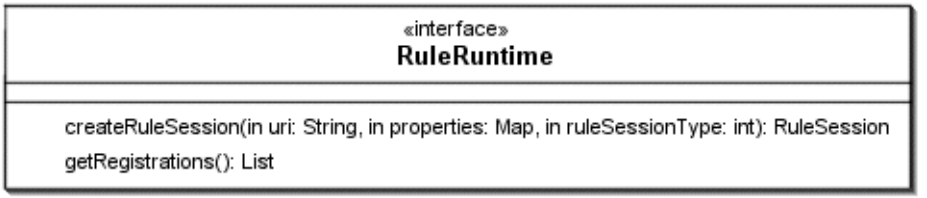
Figure 1 RuleServiceProvider Class Diagram



The `RuleServiceProvider` class implements a single point of access to the `RuleRuntime` and `RuleAdministrator` interfaces when running in the J2SE environment. It must insulate client code from the mechanism used to create implementations of the interfaces.

RuleRuntime

Figure 2 RuleRuntime Class Diagram



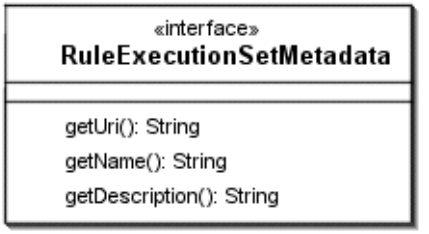
The `RuleRuntime` interface must expose methods to create `RuleSession` implementations given a previously registered `RuleExecutionSet` URI. The `RuleRuntime` implementation must also expose a method to retrieve a `List` of all registered `RuleExecutionSet` URIs.

Note that the methods on the `RuleRuntime` interface have been defined to throw `java.rmi.RemoteException` to allow implementers to provide a RMI stub-based implementation.

Please refer to the API documentation in Appendix A for more detail.

RuleExecutionSetMetadata

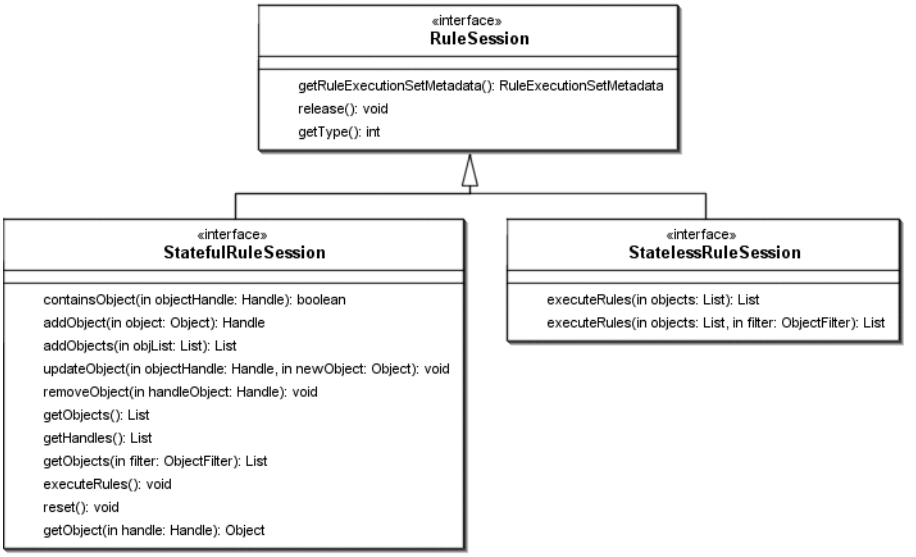
Figure 3 RuleExecutionSetMetadata Class Diagram



The `RuleExecutionSetMetadata` interface exposes metadata about a `RuleExecutionSet` to runtime clients of a `RuleSession`. The `RuleExecutionSet` is not exposed directly to runtime clients as it may contain data that is only appropriate for rule administrators or which could change without notice.

RuleSession

Figure 4 Stateful and Stateless RuleSession Class Diagram



The `RuleSession` interface defines the common behavior for the `StatefulRuleSession` and `StatelessRuleSession` interfaces. It provides a client programmer who has acquired a `RuleSession` with the means to:

- Retrieve the `RuleExecutionSetMetadata` for the `RuleSession`.
- Get the type of the `RuleSession` (must be one of `RuleRuntime.STATEFUL_SESSION_TYPE` or `RuleRuntime.STATELESS_SESSION_TYPE`).
- Release the resource associated with the `RuleSession`, rendering the `RuleSession` invalid. Subsequent attempts to access the `RuleSession` instance must throw an `InvalidRuleSessionException`.

StatelessRuleSession

The `StatelessRuleSession` interface provides client programmers with a convenient mechanism to submit a `List` of input `Objects` to a rule engine, have them evaluated against a `RuleExecutionSet`, and have the output `Objects` returned. In

addition, the client can supply an `ObjectFilter` implementation to select those `Objects` that should be returned from the rule engine. A well-written `ObjectFilter` could prevent output `Objects` from the rule engine being unnecessarily serialized between the caller and the rule engine.

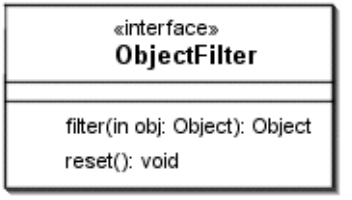
If no `ObjectFilter` is supplied, the default `ObjectFilter` attached to the `RuleExecutionSet` must be used to perform output `Object` filtering. If no default `RuleExecutionSet` `ObjectFilter` has been specified, all output `Objects` must be returned.

StatefulRuleSession

The `StatefulRuleSession` interface provides client programmers with the ability to conduct potentially long running conversations with the rule engine. Input `Objects` can be progressively added to the `StatefulRuleSession` through the `addObject` method and output `Objects` can be progressively retrieved through the `getObject` method. `Objects` that have been added to the `StatefulRuleSession` must be removed and updated using the `removeObject` and `updateObject` methods. A client programmer must test for the existence of an added `Object` using the `containsObject` method. The `removeObject`, `updateObject`, and `containsObject` methods must all use rule engine vendor created `Handle` instances to refer to and identify `Object` instances.

ObjectFilter Interface

Figure 5 ObjectFilter Class Diagram



The client programmer writes instances of classes implementing the `ObjectFilter` interface. An `ObjectFilter` instance can be passed to the `StatefulRuleSession.getObjects` and `StatelessRuleSession.executeRules` methods. The rule engine vendor must use the supplied `ObjectFilter` implementation to filter the `Objects` returned in the output `Lists` from both methods.

For example, the simple `ObjectFilter` shown in the code below filters objects based on `Class`.

Note that this `ObjectFilter` may not be suitable for use in an environment where multiple `ClassLoader`s are present.

```
public class ClassFilter implements ObjectFilter
{
   private Class filterClass;


   public ClassFilter( Class clazz )
   {
       filterClass = clazz;
   }


   /**
    * The main filtering method on the interface.
    * @param obj the object to be filtered.
    * @return the result of the filtering or <tt>null</tt>.
    */
   public Object filter( Object obj )
   {
          if ( filterClass.isAssignableFrom( obj.getClass() ) )
          {
              return obj;
          }

          return null;
       }

   /**
    * Stateful filters should implement this interface to
    * allow them to be reset to an intial state.
    */
   public void reset()
   {
   }
}
```

Handle Interface

Figure 6 Handle Class Diagram

To ensure that `Object` instances can be unambiguously identified in the event of multiple `ClassLoaders` being used or the `StatefulRuleSession` being serialized, responsibility for tracking `Object` references is delegated to the rule engine. This allows multiple instances of `Objects` that are equivalent using `Object.equals` to exist within the `StatefulRuleSession`—a common requirement of rule engines.

`Handle` instances are used by the `StatefulRuleSession` to uniquely identify instances of `Objects`. The `containsObject`, `getObject`, `removeObject`, and `updateObject` methods operate on an `Object` instance that has been previously added to the `StatefulRuleSession` using the `addObject` or `addObjects` methods. The `addObject` methods must return a `Handle` instance for an `Object` added to a `StatefulRuleSession`. The returned `Handle` instance must be subsequently usable to refer to the added `Object` using the `containsObject`, `getObject`, `removeObject`, and `updateObject` methods.

Note that a call to `executeRules` may invalidate Handles held by the client, if the `Objects` bound to the `Handles` are removed from the `RuleSession` state. A client should use the `containsObject` method to test for the existence of an `Object` bound to a `Handle` or catch `InvalidHandleExceptions` appropriately.

`Handle` instances must still be valid after the `Handle` has been serialized or the `StatefulRuleSession` has been serialized.

The implementation strategy backing the `Handle` instance returned by a rule engine vendor is not defined in the specification and must be opaque to the client code using the `Handle`.

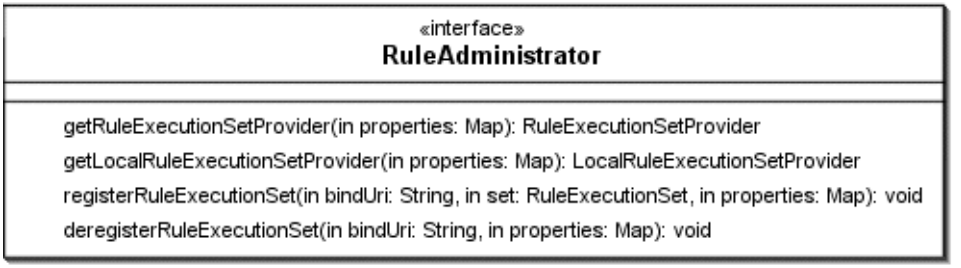
10.2 Administrator API

The administrator API for the specification is defined in the `javax.rules.admin` package. The high-level capabilities of the administrator API are:

- Acquire an instance of the `RuleAdministrator` interface through the `RuleServiceProvider` class.
- Create a `RuleExecutionSet` from external Serializable or non-Serializable resources, as listed below:
  - `org.w3c.dom.Element` – for reading from an XML sub-document.
  - `java.io.InputStream` – for reading from binary streams.
  - `java.lang.Object` – for reading from vendor specific abstract-syntax-trees.
  - `java.io.Reader` – for reading from character streams.
  - `java.lang.String` – for reading from a URI.
- Register a `RuleExecutionSet` object against a URI for use from the `RuleRuntime`. Registrations should be persistent and the rule engine vendor should clearly document the scope of a registration.
- Deregister a `RuleExecutionSet` object from a URI so it is no longer accessible from the `RuleRuntime`.
- Query the structural metadata of a `RuleExecutionSet` by retrieving a list of `Rule` objects from the `RuleExecutionSet`.
- Set and get application or vendor specific properties on `RuleExecutionSets` and `Rules`.

RuleAdministrator

Figure 7 RuleAdministrator Class Diagram



The `RuleAdministrator` interface defines the Administration API listed above.

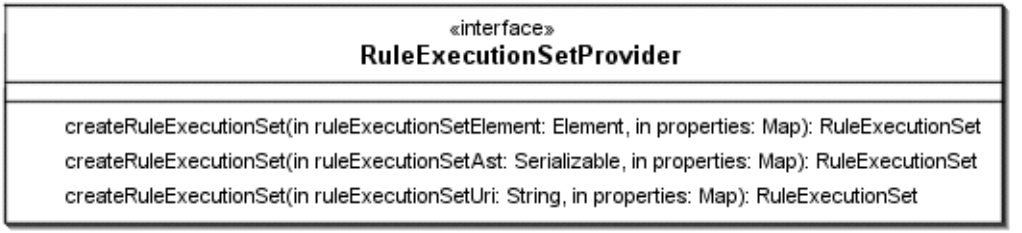
Note that the methods on the `RuleAdministrator` interface have been defined to throw `java.rmi.RemoteExcepion` to allow implementers to provide a RMI stub-based implementation.

The `RuleAdministrator` allows `RuleExecutionSet` instances to be registered against a URI for use from the runtime API, as well as methods to retrieve a `RuleExecutionSetProvider` and a `LocalRuleExecutionSetProvider` implementation.

Note that rule engine vendors may choose not to implement a `LocalRuleExecutionSetProvider`, in which case rule engine vendors should return null when invoking the `getRuleExecutionSetProvider` method.

RuleExecutionSetProvider

Figure 8 RuleExecutionSetProvider Class Diagram



The `RuleExecutionSetProvider` interface defines methods to create a `RuleExecutionSet` from a number of `Serializable` sources. The contents of these sources may be serialized or marshaled across JVMs to a remote rule engine implementation at a rule engine vendor's discretion. This is in contrast to the `LocalRuleExecutionSetProvider`, which creates `RuleExecutionSet` instances from resources that cannot be referenced by a remote rule engine.

Please refer to the Javadoc API documentation for more detail.

LocalRuleExecutionSetProvider

Figure 9 LocalRuleExecutionSetProvider Class Diagram

The `LocalRuleExecutionSetProvider` interface defines methods to create a `RuleExecutionSet` from non-`Serializable` resources, such as binary `InputStreams` or character-based `Readers`. The `LocalRuleExecutionSetProvider` may only be relevant to rule engines that are running in the JVM of the caller, and hence providing an implementation of this interface is optional for rule engine vendors. Rule engine vendors must return 'null' from the `RuleAdministrator.getLocalRuleExecutionSetProvider` method if not supporting this functionality.

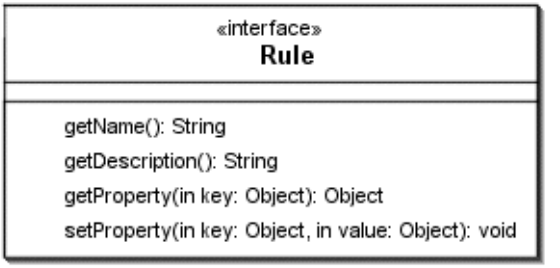
Please refer to the API documentation in Appendix A for more detail.

RuleExecutionSet Registration URI

The binding mechanism used to associate a binary `RuleExecutionSet` instance with a `String` URI is not prescribed by the specification.

Rule

Figure 10 Rule Class Diagram

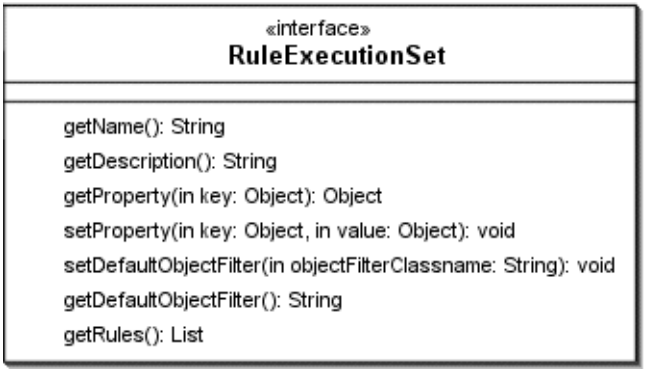


The `Rule` interface merely exposes name and description metadata for a `Rule`.

A `Rule` interface must also contain `getProperty` and `setProperty` methods to allow vendor specific (opaque to the specification) properties to be associated with `Rules`.

RuleExecutionSet

Figure 11 RuleExecutionSet Class Diagram



The `RuleExecutionSet` interface exposes name and description metadata for a `RuleExecutionSet`. The `RuleExecutionSet` interface also contains the `getRules` method, which allow the client programmer to retrieve the `Rule` objects contained in the `RuleExecutionSet`.

A default `ObjectFilter` is also associated with a `RuleExecutionSet`. If supplied by the client programmer through the `setDefaultObjectFilter` method, this `ObjectFilter` must be used when the `StatelessRuleSession.executeRules` or `StatefulRuleSession.getObjects` methods are called and no overriding `ObjectFilter` is supplied.

The `RuleExecutionSet` interface must also contain `getProperty` and `setProperty` methods to allow vendor specific (opaque to the specification) properties to be associated with `RuleExecutionSets`.

11 Use Cases

11.1 Usage from J2SE

The following code illustrates how a `RuleExecutionSet` can be created from an external resource using the `RuleAdministrator`, a `RuleSession` created for the `RuleExecutionSet`, and the `RuleSession` used to calculate output `Objects` from input `Objects`.

Runtime API

```
// load the RuleServiceProvider for the vendor
Class.forName( "org.jcp.jsr94.ri.RuleServiceProvider" );

RuleServiceProvider serviceProvider =
 RuleServiceProviderManager.getRuleServiceProvider( RULE_SERVICE_PROVIDER );

// create a stateless RuleSession
RuleRuntime ruleRuntime = serviceProvider.getRuleRuntime();
StatelessRuleSession srs = (StatelessRuleSession)
 ruleRuntime.createRuleSession( bindUri, null,
RuleRuntime.STATELESS_SESSION_TYPE );

// execute all the rules
List inputList = new LinkedList();
inputList.add( new String( "Foo" ) );
inputList.add( new String( "Bar" ) );
inputList.add( new Integer( 5 ) );
inputList.add( new Float( 6 ) );
List resultList = srs.executeRules( inputList );
System.out.println( "executeRules: " + resultList );

// release the session
srs.release();
```

11.2 Scenario: Rule Administration

The following code illustrates how a `RuleExecutionSet` can be created from an external resource and then registered so that it is accessible from the `RuleRuntime`.

```
String RULE_SERVICE_PROVIDER = "org.jcp.jsr94.jess";

// Load the rule service provider of the reference
// implementation.
// Loading this class will automatically register this
// provider with the provider manager.

Class.forName( "org.jcp.jsr94.jess.RuleServiceProviderImpl" );

// Get the rule service provider from the provider manager.

RuleServiceProvider serviceProvider =

  RuleServiceProviderManager.getRuleServiceProvider(
RULE_SERVICE_PROVIDER );

// get the RuleAdministrator

RuleAdministrator ruleAdministrator =
  serviceProvider.getRuleAdministrator();

// get an input stream to a ruleset

InputStream inStream = getResourceAsStream( "input_rules.xml" );

// parse the ruleset

RuleExecutionSet res1 =
  ruleAdministrator.getLocalRuleExecutionSetProvider( null ).
    createRuleExecutionSet( inStream, null );

inStream.close();

// register the RuleExecutionSet

String uri = res1.getName();
ruleAdministrator.registerRuleExecutionSet(uri, res1, null );
```

11.3 Scenario: Stateless Rule Session

The following code illustrates acquiring a `StatelessRuleSession` instance for a previously registered `RuleExecutionSet` and executing it with a `List` of input `Objects`.

```
String RULE_SERVICE_PROVIDER = "org.jcp.jsr94.jess";

// Load the rule service provider of the reference
// implementation.
// Loading this class will automatically register this
// provider with the provider manager.

Class.forName( "org.jcp.jsr94.jess.RuleServiceProviderImpl" );

// Get the rule service provider from the provider manager.

RuleServiceProvider serviceProvider =
RuleServiceProviderManager.getRuleServiceProvider(
RULE_SERVICE_PROVIDER );

// Get a RuleRuntime and invoke the rule engine.

RuleRuntime ruleRuntime = serviceProvider.getRuleRuntime();

// create a StatelessRuleSession

StatelessRuleSession statelessRuleSession =
       (StatelessRuleSession) ruleRuntime.createRuleSession(uri,
       new HashMap(), RuleRuntime.STATELESS_SESSION_TYPE);

// call executeRules with some input objects

Customer inputCustomer = new Customer("test");
inputCustomer.setCreditLimit(5000);

// Create a input list.

List input = new ArrayList();
input.add(inputCustomer);

// Execute the rules without a filter.

List results = statelessRuleSession.executeRules(input);

// Release the session.

statelessRuleSession.release();
```

11.4 Scenario: Stateful Rule Session

The following code illustrates acquiring a `StatefulRuleSession` instance for a previously registered `RuleExecutionSet`, periodically adding input `Objects`, accessing `Objects` using `Handles`, and periodically extracting output `Objects`.

```
String RULE_SERVICE_PROVIDER = "org.jcp.jsr94.jess";

// Load the rule service provider of the reference
// implementation.
// Loading this class will automatically register this
// provider with the provider manager.

Class.forName( "org.jcp.jsr94.jess.RuleServiceProviderImpl" );

// Get the rule service provider from the provider manager.

RuleServiceProvider serviceProvider =
  RuleServiceProviderManager.getRuleServiceProvider(
RULE_SERVICE_PROVIDER );

RuleRuntime ruleRuntime = serviceProvider.getRuleRuntime();

// create a StatefulRuleSession

StatefulRuleSession statefulRuleSession =
       (StatefulRuleSession) ruleRuntime.createRuleSession( uri,
       new HashMap(),
       RuleRuntime.STATEFUL_SESSION_TYPE );

// Add an Invoice.

Invoice inputInvoice = new Invoice("Invoice");
inputInvoice.setAmount(1750);

// add an Object to the statefulRuleSession

statefulRuleSession.addObject( inputInvoice );

//execute the rules

statefulRuleSession.executeRules();

// extract the Objects from the statefulRuleSession

results = statefulRuleSession.getObjects();

// Add another Invoice.
```

```
Invoice inputInvoice2 = new Invoice("Invoice 2");
inputInvoice2.setAmount(3000);

//execute the rules

statefulRuleSession.executeRules();

// extract the Objects from the statefulRuleSession

results = statefulRuleSession.getObjects();

// release the statefulRuleSession

statefulRuleSession.release();
```

12 Roles and Responsibilities

The following section specifies responsibilities of the roles involved in the configuration and use of a compliant implementation.

12.1 Rule Engine Vendor

The rule engine vendor is responsible for providing a compliant implementation of the specification.

The rule engine vendor should provide an implementation that functions in the J2SE environment.

The rule engine vendor should document the semantics of executing the rule execution set.

The rule engine vendor must document all vendor specific properties, their effects, and the default behavior (if properties are not specified).

The rule engine vendor must document where vendor-specific rule execution set documents are to be located so they can be accessed from the `RuleAdmistrator` API.

The rule engine vendor may provide management tools to register and deregister `RuleExecutionSets` using the `RuleAdministrator` API.

12.2 Rule Execution Set Administrator

The `RuleExecutionSet` administrator is responsible for managing the external, vendor-specific rule execution sets. Rule execution set management entails using vendor specific management tools.

The `RuleExecutionSet` administrator must register all `RuleExecutionSet` instances that are to be accessible to runtime clients through the `RuleRuntime` interface. If provided by the rule engine vendor or application server vendor, this may require using management tools. If no management tools are available, the `RuleExecutionSet` administrator must write compliant code to the `RuleAdministrator` interface to make `RuleExecutionSets` accessible.

12.3 Rule Runtime Client

The client of the `RuleRuntime` is responsible for runtime interaction with a `RuleSession` to execute application logic. The `RuleRuntime` client should remain cognizant of their dependence on rule engine vendor-specific feature extensions and should avoid using feature extensions if binary compatibility between compliant rule engines is desired.

13 Deployment Scenarios

13.1 Scenario: J2SE

Deployment into the J2SE environment should be simple to perform for client programmers.

Typical steps may involve the following:

- Install and download the rule engine vendor's product.
- Perform rule engine vendor-specific configuration.

- The rule engine vendor should supply the name of the `RuleServiceProvider` class to be instantiated using a `Class.forName( … )` call as well as the `RuleServiceProvider` URI used by the rule engine vendor. The URI can then be used with the `RuleServiceProviderManager` class to instantiate the correct `RuleServiceProvider`.

The rule engine vendor should strive to keep client programmer code independent of their rule engine implementation, unless the client programmer requires access to additional features not covered by the specification.

The rule engine vendor should strive to keep client programmer code independent of the J2SE environment.

14 Error Logging and Tracing

Rule engine vendors and client programmers should use a JSR-47, Logging API Specification compliant logging implementation when available (JDK 1.4).

15 Security

The specification separates the runtime and administration functionality of registering rule execution sets and executing them into individual packages. This allows more flexibility in implementing security policy or accessing control.

The specification takes the view that security is the responsibility of the environment that hosts the compliant implementation and the client programmer.

Within J2SE a number of security features and specifications exist that can be employed by implementers to limit runtime access to `Objects`, methods on `Classes`, as well as performing authentication and authorization checking on the runtime user of the rule engine.

Amongst the standard security components of the Java 2 Platform are the following:

- Java Authentication and Authorization Service (JAAS)

- Java Cryptography Extension (JCE)
- Java Secure Socket Extension (JSSE)

JAAS, JCE, and JSSE are all standard components of the JDK 1.4 platform.

JDK 1.3 and above provides client programmers with the ability to define a declarative security policy using a security policy file. The security policy defines the constraints to be imposed on the Java sandbox used to execute all client code. By defining an appropriate security policy, client programmers can limit the methods and classes accessible from client code.

15.1 Scenario: J2SE

Implementers or client programmers may implement custom security solution using any standard or proprietary security technologies. Implementers should strive to make security features optional to their implementations and unobtrusive to the client programmer to ensure client programmer code is portable across specification implementations.

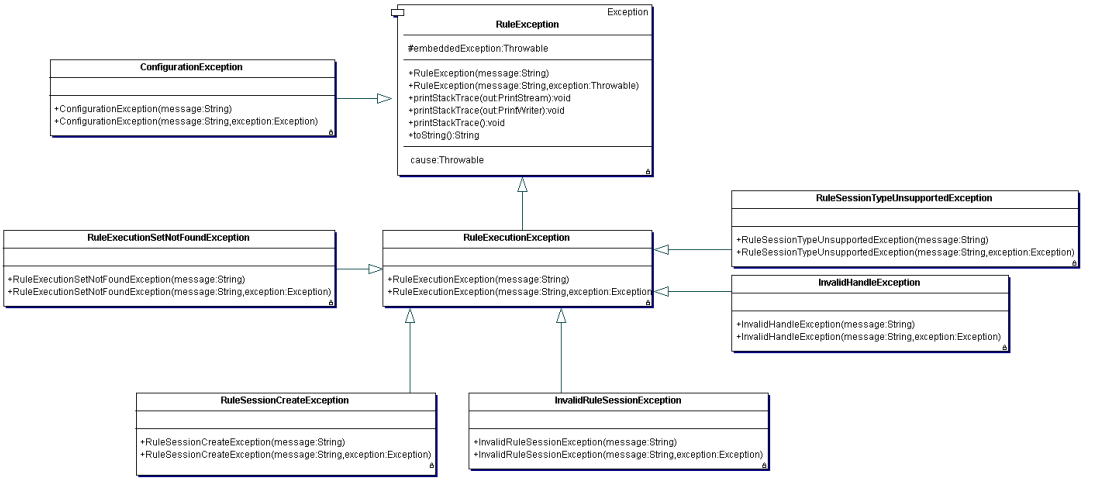
16 Exceptions

The specification defines the class `javax.rules.RuleException` as the root of the exception hierarchy. All exceptions are checked exceptions and must be explicitly caught or thrown by implementer and client code.

16.1 Rule Execution Exceptions

The specification defines the class `javax.rules.RuleExecutionException` as the root of the execution exception hierarchy.

Figure 12 Runtime Client Exceptions Class Diagram



Exception	Purpose
`InvalidHandleException`	Thrown when the client programmer passes an invalid `Handle` to an implementation. The `Handle` may reference an `Object` that is no longer within the rule engine, or the implementation class for the `Handle` may be invalid for the rule engine.
`InvalidRuleSessionException`	Thrown when a client programmer attempts to use a `RuleSession` when it is in an illegal state, or if an internal rule engine error occurs.
`RuleExecutionSetNotFoundException`	Thrown when a RuleExecutionSet cannot be resolved with the given URI.
`RuleSessionCreateException`	Thrown if the rule engine is unable to create a `RuleSession`. This may be due to resource constraints, the caller's credentials, or due to an internal error.
`RuleSessionTypeUnsupportedException`	Thrown if the client programmer requests a `RuleSession` of a type that is unsupported. A rule engine may not support stateful or stateless rule sessions as a matter of implementation or as an attribute of the requested rule execution set.

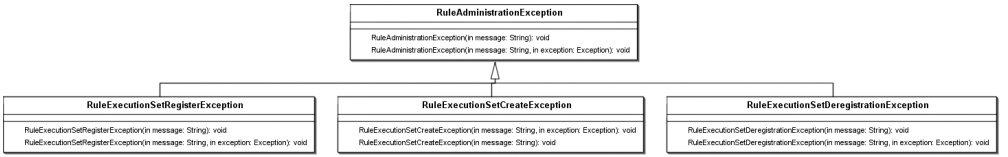
16.2 Configuration Exception

Exception	Purpose
`ConfigurationException`	Thrown when the `RuleServiceProvider` has not been correctly configured.

16.3 Administration Exceptions

The specification defines the class `javax.rules.admin.RuleAdministrationException` as the root of the administration exception hierarchy.

Figure 13 Administration Exceptions Class Diagram



Exception	Purpose
`RuleExecutionSetCreateException`	Thrown when a `RuleExecutionSet` cannot be created from an external resource.
`RuleExecutionSetRegisterException`	Thrown when a `RuleExecutionSet` instance cannot be registered against a given URI.
`RuleExecutionSetDeregisterationException`	Thrown when a `RuleExecutionSet` instance cannot be unregistered from a given URI.

17 Required APIs

This specification relies on a the following APIs:

- Java 2 SDK, Standard Edition, version 1.3 or above.
- Java API for XML Parsing 1.1 (included in JDK 1.4)

18 Change History

1.1 *12/14/2001*

- Changes to the `RuleServiceProvider` and the introduction of the `RuleServiceProviderManager` for the J2SE role (specifically).
- Moved `Rule` and `RuleExecutionSet` into the admin package and added the `RuleExecutionSetMetadata` interface for the runtime package.
- Updated class diagrams.

1.2 *7/10/2002*

- Updated the license agreement.

1.3 *7/29/2002*

- Updated the license agreement.

1.4 *9/20/2002*

- API Changes

1.5 *7/16/2003*

- License agreement completed and agreed upon by expert group members.

1.6 *9/15/2003*

- Updated references and incorporated edits.

19 Acknowledgments

Colleen McClintock, ILOG SA.

Jason Howes, BEA Systems Inc.

Robert Bergman, BEA Systems Inc.

Feng Ye, ProAct Technologies Corp.

Bob McWhirter, The Werken Company

Daniel Selman, ILOG SA.