Service Component Architecture Java Component Implementation Specification Version 1.1~~+simon~~

Working Draft 10~~8~~

30~~27~~th April 2009

Specification URIs:
This Version:
http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec-wd08.html
http://docs.oasis-open.org//sca-j/sca-javaci-1.1-spec-wd08.doc
http://docs.oasis-open.org//sca-j/sca-javaci-1.1-spec-wd08.pdf
Previous Version:

Latest Version:
http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec.html
http://docs.oasis-open.org//sca-j/sca-javaci-1.1-spec.doc
http://docs.oasis-open.org//sca-j/sca-javaci-1.1-spec.pdf
Latest Approved Version:

Technical Committee:
OASIS Service Component Architecture / J (SCA-J) TC
Chair(s):
David Booz, IBM
Mark Combellack, Avaya
Editor(s):
David Booz, IBM
Mike Edwards, IBM
Anish Karmarkar, Oracle
Related work:
This specification replaces or supersedes:

- Service Component Architecture Java Component Implementation Specification Version 1.00, 15 February 2007

This specification is related to:

- Service Component Architecture Assembly Model Specification Version 1.1
- Service Component Architecture Policy Framework Specification Version 1.1
- Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Declared XML Namespace(s):

http://docs.oasis-open.org/ns/opencsa/sca/200903

Abstract:

This specification extends the SCA Assembly Model by defining how a Java class provides an implementation of an SCA component, including its various attributes such as services, references, and properties and how that class is used in SCA as a component implementation type. It requires all the annotations and APIs as defined by ~~-~~the SCA Java Common Annotations and APIs specification.

This specification also details the use of metadata and the Java API defined in the context of a Java class used as a component implementation type.

Status:

This document was last revised or approved by the OASIS Service Component Architecture / J (SCA-J) TC on the above date. The level of approval is also listed above. Check the "Latest Version" or "Latest Approved Version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/sca-j/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/sca-j/ipr.php.

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/sca-j/.

Notices

Table of Contents

1 Introduction

This specification extends the SCA Assembly Model [ASSEMBLY] by defining how a Java class provides an implementation of an SCA component (including its various attributes such as services, references, and properties) and how that class is used in SCA as a component implementation type.

This specification requires all the annotations and APIs as defined by ~~-~~the SCA Java Common Annotations and APIs specification [JAVACAA]. All annotations and APIs referenced in this document are defined in the former unless otherwise specified. Moreover, the semantics defined in the Common Annotations and APIs specification are normative.

In addition, it details the use of metadata and the Java API defined in the SCA Java Common Annotations and APIs Specification [JAVACAA] ~~in [JAVACAA]~~ in the context of a Java class used as a component implementation type

1.1 Terminology

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 **[RFC2119]**.

1.2 Normative References

[RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[ASSEMBLY] SCA Assembly Model Specification Version 1.1, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf~~http://www.oasis-open.org/committees/download.php/31722/sca-assembly-1.1-spec-cd03.pdf~~

[POLICY] SCA Policy Framework Specification Version 1.1, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd02.pdf~~http://www.oasis-open.org/committees/download.php/31608/sca-policy-1.1-spec-cd02.pdf~~

[JAVACAA] SCA Java Common Annotations and APIs Specification Version 1.1, http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01.pdf[DAB1]

[WSDL] WSDL Specification, WSDL 1.1: http://www.w3.org/TR/wsdl

[OSGi Core] OSGI Service Platform Core Specification, Version 4.0.1 http://www.osgi.org/download/r4v41/r4.core.pdf

[JAVABEANS] JavaBeans 1.01 Specification, http://java.sun.com/javase/technologies/desktop/javabeans/api/

1.3 Non-Normative References

TBD TBD

2 Service

A component implementation based on a Java class can provide one or more services.

The services provided by a Java-based implementation MUST have an interface defined in one of the following ways:

- A Java interface
- A Java class
- A Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType.~~The services provided by a Java-based implementation MUST have an interface defined in one of the following ways:~~
  - ~~A Java interface~~
  - ~~A Java class~~
  - ~~A Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType.~~~~The services provided by a Java-based implementation MUST have an interface defined in one of the following ways:~~
    - ~~A Java interface~~
    - ~~A Java class~~
    - ~~A Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType.~~

[JCI20001]

Java implementation classes MUST implement all the operations defined by the service interface. [JCI20002] If the service interface is defined by a Java interface, the Java-based component can either implement that Java interface, or implement all the operations of the interface.

Java interfaces generated from WSDL portTypes are remotable, see the WSDL to Java and Java to WSDL ~~WSDL 2 Java and Java 2 WSDL s~~section of the SCA Java Common Annotations and APIs Specification [JAVACAA] for details.

A Java implementation type can specify the services it provides explicitly through the use of the @Service annotation. In certain cases as defined below, the use of the @Service annotation is not necessary and the services a Java implementation type offers can be inferred from the implementation class itself.

2.1 Use of @Service

Service interfaces can be specified as a Java interface. A Java class, which is a component implementation, can offer a service by implementing a Java interface specifying the service contract. As a Java class can implement multiple interfaces, some of which might not define SCA services, the @Service annotation can be used to indicate the services provided by the implementation and their corresponding Java interface definitions.

The following is an example of a Java service interface and a Java implementation~~,~~ which provides a service using that interface:

Interface:

```
package services.hello;

public interface HelloService {

    String hello(String message);
}
```

Implementation class:

```
@Service(HelloService.class)
public class HelloServiceImpl implements HelloService {

   public String hello(String message) {
         ...
   }
}
```

The XML representation of the component type for this implementation is shown below for illustrative purposes. There is no need to author the component type as it is introspected from the Java class.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903">

   <service name="HelloService">
         <interface.java interface="services.hello.HelloService"/>
   </service>

</componentType>
```

Another possibility is to use the Java implementation class itself to define a service offered by a component and the interface of the service. In this case, the @Service annotation can be used to explicitly declare the implementation class defines the service offered by the implementation. In this case, a component will only offer services declared by @Service. The following illustrates this:

```
package services.hello;

@Service(HelloServiceImpl.class)
public class HelloServiceImpl implements AnotherInterface {

   public String hello(String message) {
         ...
   }
   …
}
```

In the above example, HelloServiceImpl offers one service as defined by the public methods of the implementation class. The interface AnotherInterface in this case does not specify a service offered by the component. The following is an XML representation of the introspected component type:

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903">

<service name="HelloServiceImpl">
   <interface.java interface="services.hello.HelloServiceImpl"/>
</service>

</componentType>
```

The @Service annotation can be used to specify multiple services offered by an implementation as in the following example:

```
@Service(interfaces={HelloService.class, AnotherInterface.class})
public class HelloServiceImpl implements HelloService, AnotherInterface
{

      public String hello(String message) {
         ...
      }
   …
}
```

The following snippet shows the introspected component type for this implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903">

   <service name="HelloService">
         <interface.java interface="services.hello.HelloService"/>
   </service>
   <service name="AnotherService">
         <interface.java interface="services.hello.AnotherService"/>
   </service>

</componentType>
```

2.2 Local and Remotable Sservices

A Java service contract defined by an interface or implementation class uses the @Remotable annotation to declare that the service follows the semantics of remotable services as defined by the ~~SCA Assembly Specification~~SCA Assembly Model Specification [ASSEMBLY]. The following example demonstrates the use of the @Remotable annotation:

```
package services.hello;

@Remotable
public interface HelloService {

   String hello(String message);
}
```

Unless annotated with a @Remotable annotation, a service defined by a Java interface or a Java implementation class is inferred to be a local service as defined by the SCA Assembly Model Specification [ASSEMBLY].

An implementation class can provide hints to the SCA runtime about whether it can achieve pass-by-value ~~-~~semantics without making a copy by using the @AllowsPassByReference annotation.

2.3 Introspecting Sservices Ooffered by a Java Iimplementation

The services offered by a Java implementation class are determined through introspection, as defined in the section "Component Type of a Java Implementation".

If the interfaces of the SCA services are not specified with the @Service annotation on the implementation class, it is assumed that all implemented interfaces that have been annotated as @Remotable are the service interfaces provided by the component. If an implementation class has only implemented interfaces that are not annotated with a @Remotable annotation, the class is considered to implement a single ***local*** service whose type is defined by the class (note that local services can be typed using either Java interfaces or classes).

2.4 Non-Blocking Service Operations

Service operations defined by a Java interface or by a Java implementation class can use the @OneWay annotation to declare that the SCA runtime needs to honor non-blocking semantics as defined by the SCA Assembly Model Specification [ASSEMBLY] when a client invokes the service operation.

2.5 Callback Services

A callback interface can be declared by using the @Callback annotation on the service interface or Java implementation class as described in the Java Common Annotations and APIs Specification [JAVACAA]. Alternatively, the @callbackInterface attribute of the <interface.java/> element can be used to declare a callback interface.

3 References

A Java implementation class can obtain ***service references*** either through injection or through the ComponentContext API as defined in the SCA Java Common Annotations and APIs Specification [JAVACAA]. When possible, the preferred mechanism for accessing references is through injection.

3.1 Reference Injection

A Java implementation type can explicitly specify its references through the use of the @Reference annotation as in the following example:

```
public class ClientComponentImpl implements Client {
      private HelloService service;

      @Reference
      public void setHelloService(HelloService service) {
            this.service = service;
      }
}
```

If @Reference marks a setter method, the SCA runtime provides the appropriate implementation of the service reference contract as specified by the parameter type of the method. This is done by invoking the setter method of an implementation instance of the Java class. When injection occurs is defined by the ***scope*** of the implementation. However, injection always occurs before the first service method is called.

If @Reference marks a field, the SCA runtime provides the appropriate implementation of the service reference contract as specified by the field type. This is done by setting the field on an implementation instance of the Java class. When injection occurs is defined by the scope of the implementation. However, injection always occurs before the first service method is called.

If @Reference marks a parameter on a constructor, the SCA runtime provides the appropriate implementation of the service reference contract as specified by the constructor parameter during creation~~instantiation~~ of an implementation instance of the Java class.

Except for constructor parameters, r~~R~~eferences marked with the @Reference annotation can be declared with required=false, as defined by the Java Common Annotations and APIs Specification [JAVACAA] - i.e., the reference multiplicity is 0..1 or 0..n, where the implementation is designed to cope with the reference not being wired to a target service.

In the case where a Java class contains no @Reference or @Property annotations, references are determined by introspecting the implementation class as described in the section "ComponentType of an Implementation with no @Reference or @Property annotations ".

3.2 Dynamic Reference Access

As an alternative to reference injection, service references can be accessed dynamically through the API methods ComponentContext.getService() and ComponentContext.getServiceReference() methods as described in the Java Common Annotations and APIs Specification [JAVACAA].

4 Properties

4.1 Property Injection

Properties can be obtained either through injection or through the ComponentContext API as defined in the SCA Java Common Annotations and APIs Specification [JAVACAA]. When possible, the preferred mechanism for accessing properties is through injection.

A Java implementation type can explicitly specify its properties through the use of the @Property annotation as in the following example:

```
      public class ClientComponentImpl implements Client {
            private int maxRetries;

            @Property
            public void setMaxRetries(int maxRetries) {
                  this.maxRetries = maxRetries;
         }
   }
```

If the @Property annotation marks a setter method, the SCA runtime provides the appropriate property value by invoking the setter method of an implementation instance of the Java class. When injection occurs is defined by the scope of the implementation. However, injection always occurs before the first service method is called.

If the @Property annotation marks a field, the SCA runtime provides the appropriate property value by setting the value of the field of an implementation instance of the Java class. When injection occurs is defined by the scope of the implementation. However, injection always occurs before the first service method is called.

If the @Property annotation marks a parameter on a constructor, the SCA runtime provides the appropriate property value during creation~~instantiation~~ of an implementation instance of the Java class.

Except for constructor parameters, p~~P~~roperties marked with the @Property annotation can be declared with required=false as defined by the Java Common Annotations and APIs Specification [JAVACAA], i.e.~~.~~ the property mustSupply attribute is false and where the implementation is designed to cope with the component configuration not supplying a value for the property.

In the case where a Java class contains no @Reference or @Property annotations, properties are determined by introspecting the implementation class as described in the section "ComponentType of an Implementation with no @Reference or @Property annotations ".

4.2 Dynamic Property Access

As an alternative to property injection, properties can also be accessed dynamically through the ComponentContext.getProperty() method as described in the Java Common Annotations and APIs Specification [JAVACAA].

5 Implementation Instance Creation

A Java implementation class MUST provide a public or protected constructor that can be used by the SCA runtime to create the implementation instance. [JCI50001] The constructor can contain parameters; in the presence of such parameters, the SCA container passes the applicable property or reference values when invoking the constructor. Any property or reference values not supplied in this manner are set into the field or are passed to the setter method associated with the property or reference before any service method is invoked.

The constructor to use for the creation of an implementation instance MUST be selected by the SCA runtime using the sequence:

1. A declared constructor annotated with a @Constructor annotation.
2. A declared constructor, all of whose parameters are annotated with either @Property or @Reference.
3. A no-argument constructor.~~The constructor to use for the creation of an implementation instance MUST be selected by the SCA runtime using the sequence:~~

    ~~1. A declared constructor annotated with a @Constructor annotation.~~

    ~~2. A declared constructor, all of whose parameters are annotated with either @Property or @Reference.~~

    ~~3. A no-argument constructor.~~~~The constructor to use for the creation of an implementation instance MUST be selected by the SCA runtime using the sequence:~~

    ~~1. A declared constructor annotated with a @Constructor annotation.~~

    ~~2. A declared constructor, all of whose parameters are annotated with either @Property or @Reference.~~

    ~~3. A no-argument constructor.~~

[JCI50004]

The @Constructor annotation MUST only be specified on one constructor; the SCA container MUST raise an error if multiple constructors are annotated with @Constructor. [JCI50002]

The SCA runtime MUST raise an error if there are multiple constructors that are not annotated with @Constructor and have a non-empty parameter list with all parameters annotated with either @Property or @Reference. [JCI50005]

The property or reference associated with each parameter of a constructor is identified through the presence of a @Property or @Reference annotation on the parameter declaration

Cyclic references between components MUST be handled by the SCA runtime in one of two ways:

- If any reference in the cycle is optional, then the container can inject a null value during construction, followed by injection of a reference to the target before invoking any service.
- The container can inject a proxy to the target service; invocation of methods on the proxy can result in a ServiceUnavailableException~~Cyclic references between components MUST be handled by the SCA runtime in one of two ways:~~

    ~~• If any reference in the cycle is optional, then the container can inject a null value during construction, followed by injection of a reference to the target before invoking any service.~~

~~• The container can inject a proxy to the target service; invocation of methods on the proxy can result in a ServiceUnavailableException~~~~Cyclic references between components MUST be handled by the SCA runtime in one of two ways:~~

    ~~• If any reference in the cycle is optional, then the container can inject a null value during construction, followed by injection of a reference to the target before invoking any service.~~

    ~~• The container can inject a proxy to the target service; invocation of methods on the proxy can result in a ServiceUnavailableException.~~

[JCI50003]

The following are examples of legal Java component constructor declarations:

```
/** Constructor declared using @Constructor annotation */
public class Impl1 {
	private String someProperty;
	@Constructor
	public Impl1( @Property("someProperty") String propval ) {...}
}

/** Declared constructor unambiguously identifying all Property */
/** and Reference values */
public class Impl2 {
	private String someProperty;
	private SomeService someReference;
	public Impl2( @Property("someProperty") String a,
			@Reference("someReference") SomeService b )
	{...}
}


/** Declared constructor unambiguously identifying all Property */
/** and Reference values
plus an additional Property injected */
/** via a setter method */
public class Impl3 {
	private String someProperty;
	private String anotherProperty;
	private SomeService someReference;
	public Impl3( @Property("someProperty") String a,
			@Reference("someReference") SomeService b)
	{...}
	@Property
	public void setAnotherProperty( String anotherProperty ) {...}
}

/** No-arg constructor */
public class Impl4 {
	@Property
	public String someProperty;
	@Reference
	public SomeService someReference;
```

```
        public Impl4() {...}
    }

    /** Unannotated implementation with no-arg constructor */
    public class Impl5 {
        public String someProperty;
        public SomeService someReference;
        public Impl5() {...}
    }
```

6 Implementation Scopes and Lifecycle Callbacks

The Java implementation type supports all of the scopes defined in the Java Common Annotations and APIs Specification: STATELESS and COMPOSITE. The SCA runtime MUST support the STATELESS and COMPOSITE implementation scopes. [JCI60001]

Implementations specify their scope through the use of the @Scope annotation as in:

```
    @Scope("COMPOSITE")
    public class ClientComponentImpl implements Client {
    // …
}
```

When the @Scope annotation is not specified on an implementation class, its scope is defaulted to STATELESS.

A Java component implementation specifies init and destroy methods~~callbacks~~ by using the @Init and @Destroy annotations respectively, as described in the Java Common Annotations and APIs specification [JAVACAA].

For example:

```
    public class ClientComponentImpl implements Client {

        @Init
        public void init() {
            //…
    }

    @Destroy
        public void destroy() {
            //…
    }
}
```

7 Accessing a Callback Service

Java implementation classes that implement a service which has an associated callback interface can use the @Callback annotation to have a reference to the callback service associated with the current invocation injected on a field or injected via a setter method.

As an alternative to callback injection, references to the callback service can be accessed dynamically through the API methods RequestContext.getCallback() and RequestContext.getCallbackReference() as described in the Java Common Annotations and APIs Specification [JAVACAA].

8 Component Type of a Java Implementation

An SCA runtime MUST introspect the componentType of a Java implementation class following the rules defined in the section "Component Type of a Java Implementation". [JCI80001]

The component type of a Java Implementation is introspected from the implementation class as follows:

A <service/> element exists for each interface or implementation class identified by a @Service annotation:

- name attribute is the simple name of the interface or implementation class (i.e., without the package name)
- requires attribute is omitted unless the service implementation class is annotated with general or specific intent annotations - in this case, the requires attribute is present with a value equivalent to the intents declared by the service implementation class.
- policySets attribute is omitted unless the service implementation class is annotated with @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by the @PolicySets annotation.
- <interface.java> child element is present with the interface attribute set to the fully qualified name of the interface or implementation class identified by the @Service annotation. See the Java Common Annotations and APIs specification [JAVACAA] for a definition of how policy annotations on Java interfaces, Java classes, and methods of Java interfaces are handled.
- binding child element is omitted
- callback child element is omitted

A <reference/> element exists for each @Reference annotation:

- name attribute has the value of the name parameter of the @Reference annotation, if present, otherwise it is the name of the field or the JavaBeans property name [JAVABEANS] corresponding to the setter method name, depending on what element of the class is annotated by the @Reference (note: for a constructor parameter, the @Reference annotation needs to have a name parameter)
- autowire attribute is omitted
- wiredByImpl attribute is omitted
- target attribute is omitted
- a) where the type of the field, setter or constructor parameter is an interface, the multiplicity attribute is (1..1) unless the @Reference annotation contains required=false, in which case it is (0..1)
  b) where the type of the field, setter or parameter is an array or is a java.util.Collection, the multiplicity attribute is (1..n) unless the @Reference annotation contains required=false, in which case it is (0..n)
- requires attribute is omitted unless the field, setter method or parameter is also annotated with general or specific intent annotations - in this case, the requires attribute is present with a value equivalent to the intents declared by the Java reference.
- policySets attribute is omitted unless the field, setter method or parameter is also annotated with @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by the @PolicySets annotation.
- <interface.java> child element with the interface attribute set to the fully qualified name of the interface class which types the field or setter method. See the Java Common Annotations and

APIs specification [JAVACAA] for a definition of how policy annotations on Java interfaces and methods of Java interfaces are handled.

- binding child element is omitted
- callback child element is omitted

A <property/> element exists for each @Property annotation:

- name attribute has the value of the name parameter of the @Property annotation, if present, otherwise it is the name of the field or the JavaBeans property name [JAVABEANS] corresponding to the setter method name, depending on what element of the class is annotated by the @Property (note: for a constructor parameter, the @Property annotation needs to have a name parameter)
- value attribute is omitted
- type attribute which is set to the XML type implied by the JAXB mapping of the Java type of the field or the Java type defined by the parameter of the setter method. Where the type of the field or of the setter method is an array, the element type of the array is used. Where the type of the field or of the setter method is a~~n~~ java.util.Collection, the parameterized type of the Collection or its member type is used. If the JAXB mapping is to a global element rather than a type (JAXB @XMLRootElement annotation), the type attribute is omitted.
- element attribute is omitted unless the JAXB mapping of the Java type of the field or the Java type defined by the parameter of the setter method is to a global element (JAXB @XMLRootElement annotation). In this case, the element attribute has the value of the name of the XSD global element implied by the JAXB mapping.
- many attribute is set to “false” unless the type of the field or of the setter method is an array or a java.util.Collection, in which case it is set to "true".
- mustSupply attribute is set to "true" unless the @Property annotation has required=false, in which case it is set to "false"

An <implementation.java/> element exists if the service implementation class is annotated with general or specific intent annotations or with @PolicySets:

- requires attribute is omitted unless the service implementation class is annotated with general or specific intent annotations - in this case, the requires attribute is present with a value equivalent to the intents declared by the service implementation class.
- policySets attribute is omitted unless the service implementation class is annotated with @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by the @PolicySets annotation.

8.1 Component Type of an Implementation with no @Service A~~a~~nnotations

The section defines the rules for determining the services of a Java component implementation that does not explicitly declare them using the @Service annotation. Note that these rules apply only to implementation classes that contain ***no*** @Service annotations.

If there are no SCA services specified with the @Service annotation in an implementation class, the class offers:

- either: one Service for each of the interfaces implemented by the class where the interface is annotated with @Remotable.
- or: if the class implements zero interfaces where the interface is annotated with @Remotable, then by default the implementation offers a single local service whose type is the implementation class itself

A <service/> element exists for each service identified in this way:

- name attribute is the simple name of the interface or the simple name of the class
- requires attribute is omitted unless the service implementation class is annotated with general or specific intent annotations - in this case, the requires attribute is present with a value equivalent to the intents declared by the service implementation class.
- policySets attribute is omitted unless the service implementation class is annotated with @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by the @PolicySets annotation.
- <interface.java> child element is present with the interface attribute set to the fully qualified name of the interface class or to the fully qualified name of the class itself. See the Java Common Annotations and APIs specification [JAVACAA] for a definition of how policy annotations on Java interfaces, Java classes, and methods of Java interfaces are handled.
- binding child element is omitted
- callback child element is omitted

8.2 ComponentType of an Implementation with no @Reference or @Property Aannotations

The section defines the rules for determining the properties and the references of a Java component implementation that does not explicitly declare them using the @Reference or the @Property annotations. Note that these rules apply only to implementation classes that contain ***no*** @Reference annotations ***and no*** @Property annotations.

In the absence of any @Property or @Reference annotations, the properties and references of an implementation class are defined as follows:

The following setter methods and fields are taken into consideration:

1. Public setter methods that are not part of the implementation of an SCA service (either explicitly marked with @Service or implicitly defined as described above)
2. Public or protected fields unless there is a public setter method for the same name

An unannotated field or setter method is a ***reference*** if:

- its type is an interface annotated with @Remotable
- its type is an array where the element type of the array is an interface annotated with @Remotable
- its type is a java.util.Collection where the parameterized type of the Collection or its member type is an interface annotated with @Remotable

The reference in the component type has:

- name attribute with the value of the name of the field or the JavaBeans property name [JAVABEANS] corresponding to the setter method name
- multiplicity attribute is (1..1) for the case where the type is an interface multiplicity attribute is (1..n) for the cases where the type is an array or is a java.util.Collection
- <interface.java> child element with the interface attribute set to the fully qualified name of the interface class which types the field or setter method. See the Java Common Annotations and APIs specification [JAVACAA] for a definition of how policy annotations on Java interfaces and methods of Java interfaces are handled.
- requires attribute is omitted unless the field or setter method is also annotated with general or specific intent annotations - in this case, the requires attribute is present with a value equivalent to the intents declared by the Java reference.

- policySets attribute is omitted unless the field or setter method is also annotated with @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by the @PolicySets annotation.
- all other attributes and child elements of the reference are omitted

An unannotated field or setter method is a ***property*** if it is not a reference following the rules above.

For each property of this type, the component type has a property element with:

- name attribute with the value of the name of the field or the JavaBeans property name [JAVABEANS] corresponding to the setter method name
- type attribute and element attribute set as described for a property declared via a @Property annotation
- value attribute omitted
- many attribute set to "false" unless the type of the field or of the setter method is an array or a java.util.Collection, in which case it is set to "true".
- mustSupply attribute set to true

8.3 Component Type Introspection Examples

Example 8.1 shows how intent annotations can be applied to service and reference interfaces and methods as well as to a service implementation class.

```
// Service interface
package test;
import org.oasisopen.sca.annotation.Authentication;
import org.oasisopen.sca.annotation.Confidentiality;

@Authentication
public interface MyService {
    @Confidentiality
    void mymethod();
}

// Reference interface
package test;
import org.oasisopen.sca.annotation.Integrity;

public interface MyRefInt {
    @Integrity
    void mymethod1();
}

// Service implementation class
package test;
import static org.oasisopen.sca.Constants.SCA_PREFIX;
import org.oasisopen.sca.annotation.Confidentiality;
import org.oasisopen.sca.annotation.Reference;
import org.oasisopen.sca.annotation.Service;
@Service(MyService.class)
@Requires(SCA_PREFIX+"managedTransaction")
public class MyServiceImpl {
    @Confidentiality
    @Reference
    protected MyRefInt myRef;

    public void mymethod() {...}
```

```
}
```

Example 8.1. Intent annotations on Java interfaces, methods, and implementations.

Example 8.2 shows the introspected component type that is produced by applying the component type introspection rules to the interfaces and implementation from example 8.1.

```
<componentType xmlns:sca=
          "http://docs.oasis-open.org/ns/opencsa/sca/200903">
    <implementation.java class="test.MyServiceImpl"
            requires="sca:managedTransaction"/>
    <service name="MyService" requires="sca:managedTransaction">
        <interface.java interface="test.MyService"/>
    </service>
    <reference name="myRef" requires="sca:confidentiality">
        <interface.java interface="test.MyRefInt"/>
    </reference>
</componentType>
```

Example 8.2. Introspected component type with intents.

8.4 Java Implementation with Cconflicting Ssetter Mmethods

If a Java implementation class, with or without @Property and @Reference annotations, has more than one setter method with the same JavaBeans property name [JAVABEANS] corresponding to the setter method name, then if more than one method is inferred to set the same SCA property or to set the same SCA reference, the SCA runtime MUST raise an error and MUST NOT instantiate the implementation class. [JCI80002]

The following are examples of illegal Java implementation due to the presence of more than one setter method resulting in either an SCA property or an SCA reference with the same name:

```
    /** Illegal since two setter methods with same JavaBeans property name
    are annotated with @Property annotation. */
    public class IllegalImpl1 {
        // Setter method with upper case initial letter 'S'
        @Property
        public void setSomeProperty(String someProperty) {...}

        // Setter method with lower case initial letter 's'
        @Property
        public void setsomeProperty(String someProperty) {...}
    }

    /** Illegal since setter methods with same JavaBeans property name are
    annotated with @Reference annotation. */
    public class IllegalImpl2 {
        // Setter method with upper case initial letter 'S'
        @Reference
        public void setSomeReference(SomeService service) {...}

        // Setter method with lower case initial letter 's'
        @Reference
        public void setsomeReference(SomeService service) {...}
    }

    /** Illegal since two setter methods with same JavaBeans property name
    are resulting in an SCA property.  Implementation has no @Property or
    @Reference annotations. */
    public class IllegalImpl3 {
```

```
        // Setter method with upper case initial letter 'S'
        public void setSomeOtherProperty(String someProperty) {...}

        // Setter method with lower case initial letter 's'
        public void setsomeOtherProperty(String someProperty) {...}
    }

    /** Illegal since two setter methods with same JavaBeans property name
    are resulting in an SCA reference.  Implementation has no @Property or
    @Reference annotations. */
    public class IllegalImpl4 {
        // Setter method with upper case initial letter 'S'
        public void setSomeOtherReference(SomeService service) {...}

        // Setter method with lower case initial letter 's'
        public void setsomeOtherReference(SomeService service) {...}
    }
```

The following is an example of a legal Java implementation in spite of the implementation class having two setter methods with same JavaBeans property name [JAVABEANS] corresponding to the setter method name:

```
    /** Two setter methods with same JavaBeans property name, but one is
    annotated with @Property and the other is annotated with @Reference
    annotation. */
    public class WeirdButLegalImpl {
        // Setter method with upper case initial letter 'F'
        @Property
        public void setFoo(String foo) {...}

        // Setter method with lower case initial letter 'f'
        @Reference
        public void setfoo(SomeService service) {...}
    }
```

9 Specifying the Java Implementation Type in an Assembly

The following pseudo-schema defines the implementation element schema used for the Java implementation type:.

```
<implementation.java class="xs:NCName"
                        requires="list of xs:QName"?
                    policySets="list of xs:QName"?/>
```

The implementation.java element has the following attributes:

- ***class : NCName (1..1)*** – the fully qualified name of the Java class of the implementation
- ***requires : QName (0..n)*** – a list of policy intents. See the Policy Framework specification [POLICY] for a description of this attribute.
- ***policySets : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [POLICY] for a description of this attribute.

The <implementation.java> element MUST conform to the schema defined in sca-implementation-java.xsd. [JCI90001]

The fully qualified name of the Java class referenced by the @class attribute of <implementation.java/> MUST resolve to a Java class, using the artifact resolution rules defined in Section 10.2, that can be used as a Java component implementation. [JCI90002]

The Java class referenced by the @class attribute of <implementation.java/> MUST conform to Java SE version 5.0. [JCI90003]

10 Java Packaging and Deployment Model

The SCA Assembly Model Specification [ASSEMBLY] describes the basic packaging model for SCA contributions in the chapter on Packaging and Deployment. This specification defines extensions to the basic model for SCA contributions that contain Java component implementations.

The model for the import and export of Java classes follows the model for import-package and export-package defined by the OSGi Service Platform Core Specification [OSGi Core]. Similar to an OSGI bundle, an SCA contribution that contains Java classes represents a class loader boundary at runtime. That is, classes are loaded by a contribution specific class loader such that all contributions with visibility to those classes are using the same Class Objects in the JVM.

10.1 Contribution Metadata Extensions

SCA contributions can be self contained such that all the code and metadata needed to execute the components defined by the contribution is contained within the contribution. However, in larger projects, there is often a need to share artifacts across contributions. This is accomplished through the use of the import and export extension points as defined in the sca-contribution.xml document. An SCA contribution that needs to use a Java class from another contribution can declare the dependency via an <import.java/> extension element, contained within a <contribution/> element, as defined below:

```
<import.java package="xs:string" location="xs:anyURI"?/>
```

The import.java element has the following attributes:

- ***package : string (1..1) –*** The name of one or more Java package(s) to use from another contribution. Where there is more than one package, the package names are separated by a comma ",".

  The package can have a ***version number range*** appended to it, separated from the package name by a semicolon ";" followed by the text "version=" and the version number range, for example:
  package="com.acme.package1;version=1.4.1"
  package="com.acme.package2;version=[1.2,1.3]"

  Version number range follows the format defined in the OSGi Core specification [OSGi Core]:

  [1.2,1.3] - enclosing square brackets - inclusive range meaning any version in the range from the lowest to the highest, including the lowest and the highest
  (1.3.1,2.4.1) - enclosing round brackets - exclusive range meaning any version in the range from the lowest to the highest but not including the lowest or the highest.
  1.4.1 - no enclosing brackets - implies any version at or later than the specified version number is acceptable - equivalent to [1.4.1, infinity)

  If no version is specified for an imported package, then it is assumed to have a version range of [0.0.0, infinity) - ie any version is acceptable.

- ***location : anyURI (0..1) –*** The URI of the SCA contribution which is used to resolve the java packages for this import.

Each Java package that is imported into the contribution MUST be included in one and only one import.java element. [JCI100001] Multiple packages can be imported, either through specifying multiple packages in the @package attribute or through the presence of multiple import.java elements.

The SCA runtime MUST ensure that the package used to satisfy an import matches the package name, the version number or version number range and (if present) the location specified on the import.java element [JCI100002]

An SCA contribution that wants to allow a Java package to be used by another contribution can declare the exposure via an <export.java/> extension element as defined below:

```
<export.java package="xs:string"/>
```

The export.java element has the following attributes:

- ***package : string (0..1) –*** The name of one or more Java package(s) to expose for sharing by another contribution. Where there is more than one package, the package names are separated by a comma ",".
  The package can have a ***version number*** appended to it, separated from the package name by a semicolon ";" followed by the text "version=" and the version number:
  package="com.acme.package1;version=1.4.1"

  The package can have a ***uses directive*** appended to it, separated from the package name by a semicolon ";" followed by the text "uses=" which is then followed by a list of package names contained within single quotes "'" (needed as the list contains commas).

  The uses directive indicates that the SCA runtime MUST ensure that any SCA contribution that imports this package from this exporting contribution also imports the same version as is used by this exporting contribution of any of the packages contained in the uses directive. [JCI100003]
  Typically, the packages in the uses directive are packages used in the interface to the package being exported (eg as parameters or as classes/interfaces that are extended by the exported package). Example:

  package="com.acme.package1;uses='com.acme.package2,com.acme.package3'"

If no version information is specified for an exported package, the version defaults to 0.0.0.

If no uses directive is specified for an exported package, there is no requirement placed on a contribution which imports the package to use any particular version of any other packages.

Each Java package that is exported from the contribution MUST be included in one and only one export.java element. [JCI100004] Multiple packages can be exported, either through specifying multiple packages in the @package attribute or through the presence of multiple export.java elements.

For example, a contribution that wants to:

- use classes from the *some.package* package from another contribution (any version)
- use classes of the *some.other.package* package from another contribution, at exactly version 2.0.0
- expose the *my.package* package from its own contribution, with version set to 1.0.0

would specify an sca-contribution.xml file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200903>
 …
   <import.java package="some.package"/>
   <import.java package="some.other.package;version=[2.0.0]"/>
   <export.java package="my.package;version=1.0.0"/>
</contribution>
```

A Java package that is specified on an export element MUST be contained within the contribution containing the export element. [JCI100007]

10.2 Java Artifact Resolution

The SCA runtime MUST ensure that within a contribution, Java classes are resolved according to the following steps in the order specified:

1. If the contribution contains a Java Language specific resolution mechanism such as a classpath declaration in the archive's manifest, then that mechanism is used first to resolve classes. If the class is not found, then continue searching at step 2.

2. If the package of the Java class is specified in an import declaration then:

a) if @location is specified, the location searched for the class is the contribution declared by the @location attribute.

b) if @location is not specified, the locations which are searched for the class are the contribution(s) in the Domain which have export declarations for that package. If there is more than one contribution exporting the package, then the contribution chosen is SCA Runtime dependent, but is always the same contribution for all imports of the package.

If the java package is not found, continue to step 3.

3. The contribution itself is searched using the archive resolution rules defined by the Java Language.~~The SCA runtime MUST ensure that within a contribution, Java classes are resolved according to the following steps in the order specified:~~

~~1. If the contribution contains a Java Language specific resolution mechanism such as a classpath declaration in the archive's manifest, then that mechanism is used first to resolve classes. If the class is not found, then continue searching at step 2.~~

~~2. If the package of the Java class is specified in an import declaration then:~~

~~a) if @location is specified, the location searched for the class is the contribution declared by the @location attribute.~~

~~b) if @location is not specified, the locations which are searched for the class are the contribution(s) in the Domain which have export declarations for that package. If there is more than one contribution exporting the package, then the contribution chosen is SCA Runtime dependent, but is always the same contribution for all imports of the package.~~

~~If the java package is not found, continue to step 3.~~

~~3. The contribution itself is searched using the archive resolution rules defined by the Java Language.~~The SCA runtime MUST ensure that within a contribution, Java classes are resolved according to the following steps in the order specified:

~~1. If the contribution contains a Java Language specific resolution mechanism such as a classpath declaration in the archive's manifest, then that mechanism is used first to resolve classes. If the class is not found, then continue searching at step 2.~~

~~2. If the package of the Java class is specified in an import declaration then:~~

~~a) if @location is specified, the location searched for the class is the contribution declared by the @location attribute.~~

~~b) if @location is not specified, the locations which are searched for the class are the contribution(s) in the Domain which have export declarations for that package. If there is more than one contribution exporting the package, then the contribution chosen is SCA Runtime dependent, but is always the same contribution for all imports of the package.~~

~~If the java package is not found, continue to step 3.~~

~~3. The contribution itself is searched using the archive resolution rules defined by the Java Language.~~

[JCI100008]

10.3 Class L~~l~~oader Model

The SCA runtime MUST ensure that the Java classes used by a contribution are all loaded by a class loader that is unique for each contribution in the Domain. [JCI100010] The SCA runtime MUST ensure

that Java classes that are imported into a contribution are loaded by the exporting contribution's class loader [JCI100011], as described in the section "Contribution Metadata Extensions"

For example, suppose contribution A using class loader ACL, imports package some.package from contribution B that is using class loader BCL then the expression:~~;~~

```
ACL.loadClass(importedClassName) == BCL.loadClass(importedClassName)
```

evaluates to true.

The SCA runtime MUST set the thread context class loader of a component implementation class to the class loader of its containing contribution. [JCI100009]

11 Conformance

The XML schema pointed to by the RDDL document at the namespace URI, defined by this specification, are considered to be authoritative and take precedence over the XML schema defined in the appendix of this document.

There are three categories of artifacts that this specification defines conformance for: SCA Java Component Implementation Composite Document, SCA Java Component Implementation Contribution Document and SCA Runtime.

11.1 SCA Java Component Implementation Composite Document

An SCA Java Component Implementation Composite Document is an SCA Composite Document, as defined by the SCA Assembly Model S~~s~~pecification Section 13.1 [ASSEMBLY], that uses the <implementation.java> element. Such an SCA Java Component Implementation Composite Document MUST be a conformant SCA Composite Document, as defined by [ASSEMBLY], and MUST comply with the requirements specified in Section 9 of this specification.

11.2 SCA Java Component Implementation Contribution Document

An SCA Java Component Implementation Contribution Document is an SCA Contribution Document, as defined by the SCA Assembly Model specification Section 13.1 [ASSEMBLY], that uses the contribution metadata extensions defined in Section 10. Such an SCA Java Component Implementation Contribution document MUST be a conformant SCA Contribution Document, as defined by [ASSEMBLY], and MUST comply with the requirements specified in Section 10 of this specification.

11.3 SCA Runtime

An implementation that claims to conform to this specification MUST meet the following conditions:

1. The implementation MUST meet all the conformance requirements defined by the SCA Assembly Model Specification [ASSEMBLY].
2. The implementation MUST reject an SCA Java Composite Document that does not conform to the sca-implementation-java.xsd schema.
3. The implementation MUST reject an SCA Java Contribution Document that does not conform to the sca-contribution-java.xsd schema.
4. The implementation MUST meet all the conformance requirements, specified in 'Section 11 Conformance', from the SCA Java Common Annotations and APIs Specification [JAVACAA].
5. This specification permits an implementation class to use any and all the APIs and annotations defined in the Java Common Annotations and APIs Specification [JAVACAA], therefore the implementation MUST comply with all the statements in Appendix B: Conformance Items of [JAVACAA], notably all mandatory statements have to be implemented.
6. The implementation MUST comply with all statements related to an SCA Runtime, specified in 'Appendix B~~X~~: Conformance Items' of this specification, notably all mandatory statements have to be implemented.

A. XML Schemas

A.1 sca-contribution-java.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2009. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200903"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200903"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>

   <!-- Import.java -->
   <element name="import.java" type="sca:JavaImportType"/>
   <complexType name="JavaImportType">
      <complexContent>
         <extension base="sca:Import">
            <attribute name="package" type="NCName" use="required"/>
            <attribute name="location" type="anyURI" use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Export.java -->
   <element name="export.java" type="sca:JavaExportType"/>
   <complexType name="JavaExportType">
      <complexContent>
         <extension base="sca:Export">
            <attribute name="package" type="NCName" use="required"/>
         </extension>
      </complexContent>
   </complexType>

</schema>
```

A.2 sca-implementation-java.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
     OASIS trademark, IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200903"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200903"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-cd03.xsd"/>

   <!-- Java Implementation -->
   <element name="implementation.java" type="sca:JavaImplementation"
            substitutionGroup="sca:implementation"/>
   <complexType name="JavaImplementation">
      <complexContent>
         <extension base="sca:Implementation">
```

```
            <sequence>
               <any namespace="##other" processContents="lax" minOccurs="0"
                    maxOccurs="unbounded"/>
            </sequence>
            <attribute name="class" type="NCName" use="required"/>
            <anyAttribute namespace="##other" processContents="lax"/>
         </extension>
      </complexContent>
   </complexType>

</schema>
```

B. Conformance Items

This section contains a list of conformance items for the SCA Java Component Implementation specification.

Conformance ID	Description
[JCI20001]	The services provided by a Java-based implementation MUST have an interface defined in one of the following ways: • A Java interface • A Java class • A Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType.
[JCI20002]	Java implementation classes MUST implement all the operations defined by the service interface.
[JCI50001]	A Java implementation class MUST provide a public or protected constructor that can be used by the SCA runtime to create the implementation instance.
[JCI50002]	The @Constructor annotation MUST only be specified on one constructor; the SCA container MUST raise an error if multiple constructors are annotated with @Constructor.
[JCI50003]	Cyclic references between components MUST be handled by the SCA runtime in one of two ways: • If any reference in the cycle is optional, then the container can inject a null value during construction, followed by injection of a reference to the target before invoking any service. • The container can inject a proxy to the target service; invocation of methods on the proxy can result in a ServiceUnavailableException
[JCI50004]	The constructor to use for the creation of an implementation instance MUST be selected by the SCA runtime using the sequence: 1. A declared constructor annotated with a @Constructor annotation. 2. A declared constructor, all of whose parameters are annotated with either @Property or @Reference. 3. A no-argument constructor.
[JCI50005]	The SCA runtime MUST raise an error if there are multiple constructors that are not annotated with @Constructor and have a non-empty parameter list with all parameters annotated with either @Property or @Reference.
[JCI60001]	The SCA runtime MUST support the STATELESS and COMPOSITE implementation scopes.
[JCI80001]	An SCA runtime MUST introspect the componentType of a Java implementation class following the rules defined in the section "Component Type of a Java Implementation".
[JCI80002]	If a Java implementation class, with or without @Property and @Reference annotations, has more than one setter method with the same JavaBeans property name [JAVABEANS] corresponding to the setter method name, then if more than

	one method is inferred to set the same SCA property or to set the same SCA reference, the SCA runtime MUST raise an error and MUST NOT instantiate the implementation class.
[JCI90001]	The <implementation.java> element MUST conform to the schema defined in sca-implementation-java.xsd.
[JCI90002]	The fully qualified name of the Java class referenced by the @class attribute of <implementation.java/> MUST resolve to a Java class, using the artifact resolution rules defined in Section 10.2, that can be used as a Java component implementation.
[JCI90003]	The Java class referenced by the @class attribute of <implementation.java/> MUST conform to Java SE version 5.0.
[JCI100001]	Each Java package that is imported into the contribution MUST be included in one and only one import.java element.
[JCI100002]	The SCA runtime MUST ensure that the package used to satisfy an import matches the package name, the version number or version number range and (if present) the location specified on the import.java element.
[JCI100003]	The uses directive indicates that the SCA runtime MUST ensure that any SCA contribution that imports this package from this exporting contribution also imports the same version as is used by this exporting contribution of any of the packages contained in the uses directive.
[JCI100004]	Each Java package that is exported from the contribution MUST be included in one and only one export.java element.
[JCI100007]	A Java package that is specified on an export element MUST be contained within the contribution containing the export element.
[JCI100008]	The SCA runtime MUST ensure that within a contribution, Java classes are resolved according to the following steps in the order specified: 1. If the contribution contains a Java Language specific resolution mechanism such as a classpath declaration in the archive's manifest, then that mechanism is used first to resolve classes. If the class is not found, then continue searching at step 2. 2. If the package of the Java class is specified in an import declaration then: a) if @location is specified, the location searched for the class is the contribution declared by the @location attribute. b) if @location is not specified, the locations which are searched for the class are the contribution(s) in the Domain which have export declarations for that package. If there is more than one contribution exporting the package, then the contribution chosen is SCA Runtime dependent, but is always the same contribution for all imports of the package. If the java package is not found, continue to step 3. 3. The contribution itself is searched using the archive resolution rules defined by the Java Language.
[JCI100009]	The SCA runtime MUST set the thread context class loader of a component implementation class to the class loader of its containing contribution.
[JCI100010]	The SCA runtime MUST ensure that the Java classes used by a contribution are all loaded by a class loader that is unique for each contribution in the Domain.
[JCI100011]	The SCA runtime MUST ensure that Java classes that are imported into a

	contribution are loaded by the exporting contribution's class loader

975

C. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

Participant Name	Affiliation
Bryan Aupperle	IBM
Ron Barack	SAP AG*
Michael Beisiegel	IBM
Henning Blohm	SAP AG*
David Booz	IBM
Martin Chapman	Oracle Corporation
Graham Charters	IBM
Shih-Chang Chen	Oracle Corporation
Chris Cheng	Primeton Technologies, Inc.
Vamsavardhana Reddy Chillakuru	IBM
Roberto Chinnici	Sun Microsystems
Pyounguk Cho	Oracle Corporation
Eric Clairambault	IBM
Mark Combellack	Avaya, Inc.
Jean-Sebastien Delfino	IBM
Mike Edwards	IBM
Raymond Feng	IBM
Bo Ji	Primeton Technologies, Inc.
Uday Joshi	Oracle Corporation
Anish Karmarkar	Oracle Corporation
Michael Keith	Oracle Corporation
Rainer Kerth	SAP AG*
Meeraj Kunnumpurath	Individual
Simon Laws	IBM
Yang Lei	IBM
Mark Little	Red Hat
Ashok Malhotra	Oracle Corporation
Jim Marino	Individual
Jeff Mischkinsky	Oracle Corporation
Sriram Narasimhan	TIBCO Software Inc.
Simon Nash	Individual
Sanjay Patil	SAP AG*
Plamen Pavlov	SAP AG*
Peter Peshev	SAP AG*
Ramkumar Ramalingam	IBM
Luciano Resende	IBM
Michael Rowley	Active Endpoints, Inc.
Vladimir Savchenko	SAP AG*
Pradeep Simha	TIBCO Software Inc.
Raghav Srinivasan	Oracle Corporation

Scott Vorthmann TIBCO Software Inc.
Feng Wang Primeton Technologies, Inc.
Robin Yang Primeton Technologies, Inc.

980

D. Non-Normative Text

E. Revision History

[optional; should not be included in OASIS Standards]

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
wd02	2008-12-16	David Booz	* Applied resolution for issue 55, 32 * Editorial cleanup to make a working draft - [1] style changed to [ASSEMBLY] - updated namespace references
wd03	2009-02-26	David Booz	• Accepted all changes from wd02 • Applied 60, 87, 117, 126, 123
wd04	2009-03-20	Mike Edwards	Accepted all changes from wd03 Issue 105 - RFC 2119 Language added - covers most of the specification. Accepted all changes after RFC 2119 language added. Editorial fix to ensure the term "class loader" is used consistently
wd05	2009-03-24	David Booz	Applied resolution for issues: 119, 137
wd06	2009-03-27	David Booz	Accepted all previous changes and applied issues 145,146,147,151
wd07	2009-04-06	David Booz	Editorial cleanup, namespace changes, changed XML encoding to UTF-8 in examples, applied 144
wd08	2009-04-27	David Booz	Applied issue 98, 152
wd09	2009-04-29	David Booz	Editorial fixes throughout (capitalization, quotes, fonts, spec references, etc.)
wd10	2009-04-30	David Booz	Editorial fixes, indention, etc.

[DAB1]Update to CD03/PRD link at the appropriate time.