Service Component Architecture POJO Component Implementation Specification Version 1.1

Committee Specification Draft 03 / Public Review Draft 03

8 November 2010

Specification URIs:
This Version:

http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csprd03.html
http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csprd03.doc
http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csprd03.pdf (Authoritative)

Previous Version:

http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-cd02.html
http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-cd02.doc
http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-cd02.pdf (Authoritative)

Latest Version:

http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec.html
http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec.doc
http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec.pdf (Authoritative)

Technical Committee:

OASIS Service Component Architecture / J (SCA-J) TC

Chair(s):

David Booz, IBM
Anish Karmarkar, Oracle Corporation

Editor(s):

David Booz, IBM
Mike Edwards, IBM
Anish Karmarkar, Oracle Corporation

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
- SCA Policy Framework Version 1.1
- Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1

Declared XML Namespace(s):

http://docs.oasis-open.org/ns/opencsa/sca/200912

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-J 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.

Citation Format:

When referencing this specification the following citation format should be used:

SCA-JAVACI-v1.1 OASIS Committee Specification Draft 03, *Service Component Architecture POJO Component Implementation Specification Version 1.1*, November 2010. http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csd03.pdf

Notices

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The names "OASIS", “SCA” and “Service Component Architecture” are trademarks of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see http://www.oasis-open.org/who/trademark.php for above guidance.

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-J 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 SCA-J Common Annotations and APIs specification are normative.

In addition, it details the use of metadata and the Java API defined in the SCA-J Common Annotations and APIs Specification [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] OASIS Committee Draft 06, *SCA Assembly Model Specification Version 1.1*, January 2010.

http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd06.pdf

[POLICY] OASIS Committee Draft 04, *SCA Policy Framework Specification Version 1.1*, September 2010.

http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd04.pdf

[JAVACAA] OASIS Committee Draft 04, *Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1*, February 2010.

http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd04.pdf

[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/

[JAX-WS] JAX-WS 2.1 Specification (JSR-224), http://www.jcp.org/en/jsr/detail?id=224

[WSBINDING] OASIS Committee Draft 04, *SCA Web Service Binding Specification Version 1.1*, May 2010.

http://docs.oasis-open.org/opencsa/sca-bindings/sca-wsbinding-1.1-spec-cd04.pdf

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.

[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 section of the SCA-J 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.

Snippet 2-1 and **Error! Reference source not found.** are 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);
}
```

Snippet 2-1: Example Java Service Interface

Implementation class:

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

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

Snippet 2-2: Example Java Component Implementation

The XML representation of the component type for this implementation is shown in Snippet 2-3 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/200912">

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

</componentType>
```

Snippet 2-3: Effective Component Type for Implementation in Snippet 2-2

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. Snippet 2-4 illustrates this:

```
package services.hello;

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

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

Snippet 2-4: Example of Java Class Defining a Service

In Snippet 2-4, 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. Snippet 2-5 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/200912">

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

</componentType>
```

Snippet 2-5: Effective Component Type for Implementation in Snippet 2-4

The @Service annotation can be used to specify multiple services offered by an implementation as in Snippet 2-6:

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

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

```
   …
}
```

Snippet 2-6: Example of @Service Specifying Multiple Services

Snippet 2-7 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/200912">

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

</componentType>
```

Snippet 2-7: Effective Component Type for Implementation in Snippet 2-6

2.2 Local and Remotable Services

A Java interface or implementation class that defines an SCA service can use the @Remotable annotation to declare that the service follows the semantics of remotable services as defined by the SCA Assembly Model Specification [ASSEMBLY]. Snippet 2-8 and Snippet 2-9 demonstrate the use of the @Remotable annotation on a Java interface:

Interface:

```
package services.hello;

@Remotable
public interface HelloService {

   String hello(String message);
}
```

Snippet 2-8: Example Remotable Interface

Implementation class:

```
package services.hello;

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

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

Snippet 2-9: Implementation for Remotable Interface

Snippet 2-10 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/200912">
   <service name="HelloService">
      <interface.java interface="services.hello.HelloService"/>
   </service>
</componentType>
```

Snippet 2-10: Effective Component Type for Implementation in Snippet 2-9

The interface specified in the @interface attribute of the <interface.java/> element is implicitly remotable because the Java interface contains @Remotable.

If a service is defined by a Java implementation class instead of a Java interface, the @Remotable annotation can be used on the implementation class to indicate that the service is remotable. Snippet 2-11 demonstrates this:

```
package services.hello;

@Remotable
@Service(HelloServiceImpl.class)
public class HelloServiceImpl {

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

Snippet 2-11: Remotable Inteface Defined by a Class

Snippet 2-12 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/200912">
   <service name="HelloServiceImpl">
      <interface.java interface="services.hello.HelloServiceImpl"/>
   </service>
</componentType>
```

Snippet 2-12: Effective Component Type for Implementation in Snippet 2-11

The interface specified in the @interface attribute of the <interface.java/> element is implicitly remotable because the Java implementation class contains @Remotable.

It is also possible to use a Java interface with no @Remotable annotation to define an SCA service with remotable semantics. In this case, the @Remotable annotation is placed on the service implementation class, as shown in Snippet 2-13 and Snippet 2-14:

Interface:

```
package services.hello;

public interface HelloService {

   String hello(String message);
}
```

Snippet 2-13: Interface without @Remotable

Implementation class:

```
package services.hello;

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

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

Snippet 2-14: Interface Made Remotable with @Remotable on Implementation Class

In this case the introspected component type for the implementation uses the @remotable attribute of the <interface.java/> element, as shown in Snippet 2-15:

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
   <service name="HelloService">
<interface.java interface="services.hello.HelloService"
        remotable="true"/>
   </service>
</componentType>
```

Snippet 2-15: Effective Component Type for Implementation in Snippet 2-14

An SCA service defined by a @Service annotation specifying a Java interface, with no @Remotable annotation on either the interface or the service implementation class, is inferred to be a local service as defined by the SCA Assembly Model Specification [ASSEMBLY]. Similarly, an SCA service defined by a @Service annotation specifying a Java implementation class with no @Remotable annotation is inferred to be a local service.

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 Services Offered by a Java Implementation

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 and the implementation class does not contain any @Reference or @Property annotations, 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 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 SCA-J 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-J 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 Snippet 3-1:

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

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

Snippet 3-1: Specifying a Reference

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 of an implementation instance of the Java class.

Except for constructor parameters, references marked with the @Reference annotation can be declared with required=false, as defined by the SCA-J 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.

The @Remotable annotation can be used either on the service reference contract or on the reference itself to specify that the service reference contract follows the semantics of remotable services as defined by the SCA Assembly Model Specification [ASSEMBLY]; otherwise, the service reference contract has local semantics.

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 SCA-J 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-J 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 Snippet 4-1:

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

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

Snippet 4-1: Specifying a Property

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 of an implementation instance of the Java class.

Except for constructor parameters, properties marked with the @Property annotation can be declared with required=false as defined by the SCA-J 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 ".

For an unannotated field or setter method that is introspected as a property and where the Java type of the field or setter method is a JAXB [JAXB] annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition into an instance of the property's Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled. [JCI40001]

For an unannotated field or setter method that is introspected as a property and where the Java type of the field or setter method in not a JAXB [JAXB] annotated class, the SCA runtime can use any XML to Java mapping when converting property values into instances of the Java type.

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 SCA-J 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.

[JCI50004]

The @Constructor annotation MUST NOT appear on more than one constructor. [JCI50002]

In the absence of an @Constructor annotation, there MUST NOT be more than one constructor that has 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.

The construction and initialization of component implementation instances is described as part of the SCA component implementation lifecycle in the SCA-J Common Annotations and APIs specification [JAVACAA].

Snippet 5-1 shows 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() {...}
}
```

Snippet 5-1: Examples of Valid Constructors

6 Implementation Scopes and Lifecycle Callbacks

The Java implementation type supports all of the scopes defined in the SCA-J 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 shown in Snippet 6-1:

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

Snippet 6-1: Specifying the Scope of an Implementation

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 by using the @Init and @Destroy annotations respectively, as described in the SCA-J Common Annotations and APIs specification [JAVACAA].

For example:

```
public class ClientComponentImpl implements Client {

@Init
public void init() {
//…
   }

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

Snippet 6-2: Example Init and Destroy Methods

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 SCA-J 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 using the rules:

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 SCA-J 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.
- remotable attribute of <interface.java> child element is omitted unless the service is defined by a Java interface with no @Remotable annotation and the service implementation class is annotated with @Remotable, in which case the <interface.java> element has remotable="true".
- 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
- the multiplicity attribute is set according to the rules in section “@Reference” of the SCA Common Annotations and APIs Specification [JAVACAA]
- 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 or constructor parameter. See the SCA-J Common Annotations and APIs specification [JAVACAA] for a definition of how policy annotations on Java interfaces and methods of Java interfaces are handled.

- remotable attribute of <interface.java> child element is omitted unless the interface class has no @Remotable annotation and there is a @Remotable annotation on the field, setter method or constructor parameter, in which case the <interface.java> element has remotable="true".
- 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 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. Note that JAXB mapping is the default mapping, but that other mappings are possible, where supported by the SCA runtime (for example, SDO). How such alternative mappings are indicated is not described in this specification.
- 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 according to the rules in section “@Property” of the SCA Common Annotations and APIs Specification [JAVACAA].
- 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, @Reference or @Property Annotations

The section defines the rules for determining the services of a Java component implementation that contains no @Service annotations, no @Reference annotations, and no @Property annotations. If the implementation class contains any @Service, @Reference or @Property annotations, the rules in this section do not apply.

The SCA services offered by the implementation class are defined using the rules:

- 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 SCA-J 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.
- remotable attribute of <interface.java> child element is omitted
- binding child element is omitted
- callback child element is omitted

The SCA properties and references of the implementation class are defined using the rules:

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 SCA-J Common Annotations and APIs specification [JAVACAA] for a definition of how policy annotations on Java interfaces and methods of Java interfaces are handled.
- remotable attribute of <interface.java> child element is omitted
- 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 using the immediately preceeding rules.

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 are set as described for a property declared via a @Property annotation, following the JAXB mapping of the Java type of the field or setter method by default. Note that other mappings are possible, where supported by the SCA runtime (for example, SDO). How such alternative mappings are indicated is not described in this specification.
- 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

8.2 Impact of JAX-WS Annotations on ComponentType

As described in the Java Common Annotations and APIs specification [JAVACAA], there are a number of JAX-WS [JAX-WS] annotations that can affect the introspection and interpretation of Java classes and Java interfaces. This section describes the effect of the JAX-WS annotations on the introspected componentType of a Java implementation class.

8.2.1 @WebService

An interface or implementation class annotated with @WebService is treated as if it had an @Service annotation:

- The value of the name property of the @WebService annotation is used as the name of the <service/> element
- If the endpointInterface property of the @WebService annotation has a non-default value, then the interface attribute of the <interface.java/> child element of the <service/> element is set to the interface identified by the endpointInterface property.
- The <interface.java/> child element of the <service/> has the remotable attribute set to "true".
- If the wsdlLocation property of the @WebService annotation has a non-default value, then the <service/> element has an <interface.wsdl/> child element instead of an <interface.java/> child element. The value of the @interface attribute of the <interface.wsdl/> element is constructed by pointing to the portType, in the WSDL definition pointed to by @wsdlLocation, which resulted from the JAX-WS mapping for the annotated class or interface.
- If both the endpointInterface and wsdlLocation properties of the @WebService annotation have default values and there is no @Service annotation, then the interface attribute of the <interface.java/> child element of the <service/> element is set to the fully qualified name of the interface or implementation class.

As noted in the the SCA-J Common Annotations and APIs Specification [JAVACAA], a service name explicitly provided in a @Service annotation overrides any name defined by a @WebService annotation.

8.2.2 @WebMethod

- The value of the name property of the @WebMethod annotation is used when testing interface compatibility.
- If the value of the exclude property of the @WebMethod annotation is "true", then the method is excluded from the SCA interface.

8.2.3 @WebParam

- The value of the mode property of the @WebParam is considered when testing interface compatibility.
- If the value of the header property of the @WebParam is "true", then the “SOAP” intent is added to the requires attribute of the <service/> element.

8.2.4 @WebResult

- If the value of the header property of the @WebResult is "true", then the “SOAP” intent is added to the requires attribute of the <service/> element.

8.2.5 @SOAPBinding

- If an interface or class is annotated with @SOAPBinding, then the “SOAP” intent is added to the requires attribute of the <service/> element. The same is true if any method of the interface or class is annotated with @SOAPBinding

8.2.6 @WebServiceProvider

An implementation class annotated with @WebServiceProvider is treated as if it had an @Service annotation:

- Where the Java implementation class implements a Java interface that is annotated with @Remotable:
  - The @name attribute of the <service/> element in the component type is the simple name of the Java interface class where the Java implementation class implements the Java interface marked with @Remotable.
  - The <service/> element has a <interface.java/> subelement with an @interface set to the fully qualified name of the Java interface class.
- Where the Java implementation class does not implement a Java interface that is annotated with @Remotable:
  - The @name attribute of the <service/> element in the component type is the simple name of the Java implementation class.
  - The <service/> element has a <interface.java/> subelement with an @interface set to the fully qualified name of the Java implementation class and the @remotable attribute is set to "true".
- If the wsdlLocation property of the @WebServiceProvider annotation has a non-default value, then the <service/> element has an <interface.wsdl/> child element instead of an <interface.java/> child element. The value of the @interface attribute of the <interface.wsdl/> element is constructed by pointing to the portType, in the WSDL definition pointed to by @wsdlLocation, which resulted from the JAX-WS mapping for the annotated class or interface.

8.2.7 Web Service Binding

By default, the JAX-WS specification requires that JAX-WS service implementation classes have endpoints that are made available using the SOAP 1.1 HTTP WSDL binding which is denoted by the URL http://schemas.xmlsoap.org/wsdl/soap/http [JAX-WS].

Therefore, the presence of ***any*** JAX-WS annotations in an SCA implementation or in an interface class requires that any SCA services exposed by an implementation class are made available using the SOAP 1.1 HTTP WSDL binding by default. As a result, the respective <service/> elements in the component type of the implementation class each have a <binding.ws/> subelement [WSBINDING] with the SOAP.v1_1 intent added to the requires attribute of the <binding.ws/> subelement.

Note that JAX-WS annotations do not cause <reference/> elements in the component type of an implementation class to have a <binding.ws/> subelement.

8.2.7.1 @BindingType

If the default WSDL binding is not acceptable for a <service/>, the JAX-WS @BindingType annotation can be used to specify a different WSDL binding URL. If the JAX-WS @BindingType annotation is used, then the set of intents added to the requires attribute of the <binding.ws/> subelement is based on the value of the @BindingType annotation. Table 8-1 shows the mapping of the common binding types to intents. For any other URI not listed in the table, the mapped intents are undefined.

WSDL Binding Type	Intent(s)
http://schemas.xmlsoap.org/wsdl/soap/http	SOAP.v1_1
http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true	SOAP.v1_1
http://www.w3.org/2003/05/soap/bindings/HTTP/	SOAP.v1_2
http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true	SOAP.v1_2
http://www.w3.org/2010/soapjms/	SOAP, JMS

Table 8-1: Intents for WSDL Bindings

8.3 Component Type Introspection Examples

Snippet 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() {...}
}
```

Snippet 8-1: Intent Annotations on Java Interfaces, Methods, and Implementations.

Snippet 8-2 shows the introspected component type that is produced by applying the component type introspection rules to the interfaces and implementation from Snippet 8-1.

```
<componentType xmlns:sca=
          "http://docs.oasis-open.org/ns/opencsa/sca/200912">
```

```
    <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>
```

Snippet 8-2: Introspected Component Type with Intents

8.4 Java Implementation with Conflicting Setter Methods

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]

Snippet 8-3shows 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) {...}
}
```

Snippet 8-3: Example Conflicting Setter Methods

Snippet 8-4 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) {...}
}
```

Snippet 8-4: Example of Valid Combination of Settter Methods

9 Specifying the Java Implementation Type in an Assembly

Snippet 9-1 shows the pseudo-schema that 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"?/>
```

Snippet 9-1: Pseudo-Schema for implementation.java

The implementation.java element has the 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 shown in Snippet 10-1:

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

Snippet 10-1: Pseudo-Schema for import.java

The import.java element has the 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 shown in Snippet 10-2:

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

Snippet 10-2:Pseudo-Schema for export.java

The export.java element has the attributes:

- ***package : string (1..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 shown in Snippet 10-3 :

```
<?xml version="1.0" encoding="UTF-8"?>
<contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200912>
 …
   <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>
```

Snippet 10-3: Example Imports and Exports

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.

[JCI100008]

10.3 Class Loader 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)
```

Snippet 10-4: Example Class Loader Use

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 Specification 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 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-J 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 SCA-J 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: 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) 2005,2010. 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/200912"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   elementFormDefault="qualified">

   <include schemaLocation="sca-contribution-1.1-cd06.xsd"/>

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

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

</schema>
```

A.2 sca-implementation-java.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2010. 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/200912"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-cd06.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"/>
         </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.
[JCI40001]	For an unannotated field or setter method that is introspected as a property and where the Java type of the field or setter method is a JAXB [JAXB] annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition into an instance of the property's Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled.
[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 NOT appear on more than one constructor.
[JCI50004]	The constructor to use for the creation of an implementation instance MUST be selected by the SCA runtime using the sequence: 7. A declared constructor annotated with a @Constructor annotation. 8. A declared constructor, all of whose parameters are annotated with either @Property or @Reference. 9. A no-argument constructor.
[JCI50005]	In the absence of an @Constructor annotation, there MUST NOT be more than one constructor that has 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

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*
Mirza Begg	Individual
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
Derek Dougans	Individual
Mike Edwards	IBM
Ant Elder	IBM
Raymond Feng	IBM
Bo Ji	Primeton Technologies, Inc.
Uday Joshi	Oracle Corporation
Anish Karmarkar	Oracle Corporation
Khanderao Kand	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.
Paul Yang	Changfeng Open Standards Platform Software

D. 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.
cd01	2009-05-04	David Booz	Final editorial fixes for CD and PRD
cd01-rev1	2009-08-12	David Booz	Editorial fixes, applied issues: 143,153,176
cd01-rev2	2009-09-14	David Booz	Applied issues: 157,162
cd01-rev3	2010-01-18	David Booz	Upgraded namespace to latest 200912 Applied issues: 168, 171, 181, 184, 186, 192,193
cd01-rev4	2010-01-20	Bryan Aupperle	Editorial updates to match OASIS document standards

CD02	2010-02-02	David Booz	Editorial updates to produce Committee Draft All changes accepted
CD02-rev1	2010-07-13	David Booz	Applied Issue 197
CSD02-rev2	2010-11-04	David Booz	Applied Issue 203, 204, 212, 213 and prep for CSD03