Style Definition: Ref: Complex Script Font: Not Bold

# Service Component Architecture Java Component Implementation Specification Version 1.1

## Working Draft 03

## 26th February 2009

Deleted: 2

Deleted: 16

Deleted: December

Deleted: 8

**Specification URIs:**

**This Version:**

http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec-wd03.html

http://docs.oasis-open.org//sca-j/sca-javaci-1.1-spec-wd03.doc

http://docs.oasis-open.org//sca-j/sca-javaci-1.1-spec-wd03.pdf

Deleted: 2

Deleted: 2

Deleted: 2

**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 supercedes:

- Service Component Architecture Java Component ImplementationSpecification 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 Sepcification Version 1.1

**Declared XML Namespace(s):**

http://docs.oasis-open.org/ns/opencsa/sca/200712

**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



Deleted: 7

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", [insert specific trademarked names and abbreviations here] 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 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 [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 Specification, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf

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

**[WSDL]** WSDL Specification, WSDL 1.1: http://www.w3.org/TR/wsdl, WSDL 2.0: http://www.w3.org/TR/wsdl20/

**[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/

**Formatted:** Complex Script Font: Bold

**Formatted:** Complex Script Font: Bold

**Formatted:** French France

**Formatted:** Hyperlink, French France

**Formatted:** French France

## 1.3 Non-Normative References

**TBD** TBD

# 2 Service

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

The services provided by a Java-based implementation may 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.

Java implementation classes must implement all the operations defined by the service interface. 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.

A service whose interface is defined by a Java class (as opposed to a Java interface) is not remotable. Java interfaces generated from WSDL portTypes are remotable, see the WSDL 2 Java and Java 2 WSDL section of the SCA Java Common Annotations and API Specification for details.

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

## 2.1 Use of @Service

Service interfaces may be specified as a Java interface. A Java class, which is a component implementation, may offer a service by implementing a Java interface specifying the service contract. As a Java class may implement multiple interfaces, some of which may 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:

```
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 can be reflected from the Java class.

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">

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

</componentType>
```

The Java implementation class itself, as opposed to an interface, may also define a service offered by a component. In this case, @Service may 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:

```
@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 on 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="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">

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

</componentType>
```

@Service may be used to specify multiple services offered by an implementation as in:

```
@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="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">

      <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 services

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

```
package services.hello;

@Remotable
public interface HelloService {

      String hello(String message);
}
```

Unless @Remotable is declared, a service defined by a Java interface or implementation class is inferred to be a local service as defined by the SCA Assembly Model Specification.

If an implementation class has implemented interfaces that are not decorated with an @Remotable annotation, the class is considered to implement a single ***local*** service whose type is defined by the class (note that local services may be typed using either Java interfaces or classes).

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

## 2.3 Introspecting services offered by a Java implementation

In the cases described below, the services offered by a Java implementation class may be determined through introspection, eliding the need to specify them using @Service. The following algorithm is used to determine how services are introspected from an implementation class:

*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 none of the implemented interfaces is remotable, then by default the implementation offers a single service whose type is the implementation class.*

## 2.4 Non-Blocking Service Operations

Service operations defined by a Java interface or implementation class may use @OneWay to declare that the SCA runtime must honor non-blocking semantics as defined by the SCA Assembly Specification when a client invokes the service operation.

## 2.5 Callback Services

A callback interface is declared by using the @Callback annotation on the service interface implemented by a Java class.

**Deleted:** <#>Non-Conversational and Conversational Services¶
The Java implementation type supports all of the conversational service annotations as defined by the SCA Java Common Annotations and API Specification: @Conversational, @EndsConversation, and @ConversationAttributes.¶
The following semantics hold for service contracts defined by Java interface or implementation class. A service contract defined by a Java interface or implementation class is inferred to be non-conversational as defined by the SCA Assembly Specification unless it is decorated with @Conversational. In the latter case, @Conversational is used to declare that a component implementation offering the service implements conversational semantics as defined by the SCA Assembly Specification.¶

**Formatted:** Bullets and Numbering

# 3 References

References may be obtained through injection or through the ComponentContext API as defined in the SCA Java Common Annotations and API Specification. When possible, the preferred mechanism for accessing references is through injection.

## 3.1 Reference Injection

A Java implementation type may explicitly specify its references through the use of @Reference 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 public or protected setter method, the SCA runtime is required to provide the appropriate implementation of the service reference contract as specified by the parameter type of the method. This must done by invoking the setter method an implementation instance. When injection occurs is defined by the scope of the implementation. However, it will always occur before the first service method is called.

If @Reference marks a public or protected field, the SCA runtime is required to provide the appropriate implementation of the service reference contract as specified by the field type. This must done by setting the field on an implementation instance. When injection occurs is defined by the scope of the implementation.

If @Reference marks a parameter on a constructor, the SCA runtime is required to provide the appropriate implementation of the service reference contract as specified by the constructor parameter during instantiation of an implementation instance.

References may also be determined by introspecting the implementation class according to the rules defined in Section **Error! Reference source not found.**.

References may be declared optional as defined by the Java Common Annotations and API Specification.

## 3.2 Dynamic Reference Access

References may be accessed dynamically through ComponentContext.getService() and ComponentContext.getServiceReference(..) methods as described in the Java Common Annotations and API Specification.

# 4 Properties

## 4.1 Property Injection

Properties may be obtained through injection or through the ComponentContext API as defined in the SCA Java Common Annotations and API Specification. When possible, the preferred mechanism for accessing propertoes is through injection.

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

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

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

If @Property marks a public or protected setter method, the SCA runtime is required to provide the appropriate property value. This must done by invoking the setter method an implementation instance. When injection occurs is defined by the scope of the implementation.

If @Property marks a public or protected field, the SCA runtime is required to provide the appropriate property value. When injection occurs is defined by the scope of the implementation.

If @Property marks a parameter on a constructor, the SCA runtime is required to provide the appropriate property value during instantiation of an implementation instance.

Properties may also be determined by introspecting the implementation class according to the rules defined in Section **Error! Reference source not found.**.

Properties may be declared optional as defined by the Java Common Annotations and API Specification.

## 4.2 Dynamic Property Access

Properties may be accesses dynamically through ComponentContext. getProperty () method as described in the Java Common Annotations and API Specification.

# 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. The constructor may contain parameters; in the presence of such parameters, the SCA container will pass the applicable property or reference values when invoking the constructor. Any property or reference values not supplied in this manner will be set into the field or passed to the setter method associated with the property or reference before any service method is invoked.

The constructor to use is selected by the container as follows:

1. A declared constructor annotated with a @Constructor annotation.
2. A declared constructor that unambiguously identifies all property and reference values.
3. A no-argument constructor.

The @Constructor annotation must only be specified on one constructor; the SCA container must report an error if multiple constructors are annotated with @Constructor.

The property or reference associated with each parameter of a constructor is identified:

- by name in the @Constructor annotation (if present)
- through the presence of a @Property or @Reference annotation on the parameter declaration
- by uniquely matching the parameter type to the type of a property or reference

Cyclic references between components may be handled by the container in one of two ways:

- If any reference in the cycle is optional, then the container may inject a null value during construction, followed by injection of a reference to the target before invoking any service.
- The container may inject a proxy to the target service; invocation of methods on the proxy may result in a ServiceUnavailableException

The following are examples of legal Java component constructor declarations:

```
/** Simple class taking a single property value */
public class Impl1 {
    String someProperty;
    public Impl1(String propval) {...}
}

/** Simple class taking a property and reference in the constructor;
 * The values are not injected into the fields.
 *//
public class Impl2 {
    public String someProperty;
    public SomeService someReference;
```

Comment: Issue 126

Deleted: Instantiation

Deleted: instantiate

Deleted: s

```
287             public Impl2(String a, SomeService b) {...}
288         }
289
290         /** Class declaring a named property and reference through the
291         constructor */
292         public class Impl3 {
293             @Constructor({"someProperty", "someReference"})
294             public Impl3(String a, SomeService b) {...}
295         }
296
297         /** Class declaring a named property and reference through parameters
298         */
299         public class Impl3b {
300             public Impl3b(
301                 @Property("someProperty") String a,
302                 @Reference("someReference) SomeService b
303                 ) {...}
304         }
305
306         /** Additional property set through a method */
307         public class Impl4 {
308             public String someProperty;
309             public SomeService someReference;
310             public Impl2(String a, SomeService b) {...}
311             @Property public void setAnotherProperty(int x) {...}
312         }
```

# 6 Implementation Scopes and Lifecycle Callbacks

The Java implementation type supports all of the scopes defined in the Java Common Annotations and API Specification: STATELESS and COMPOSITE. 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 callbacks by using @Init and @Destroy respectively. For example:

```
public class ClientComponentImpl implements Client {

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

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

Deleted: ,

Deleted: REQUEST, CONVERSATION,

Deleted: <#>Conversational Implementation¶
Java implementation classes that are CONVERSATION scoped may use @ConversationID to have the current conversation ID injected on a public or protected field or setter method. Alternatively, the Conversation API as defined in the Java Common Annotations and API Specification may be used to obtain the current conversation ID.¶
For the provider of a conversational service, there is the need to maintain state data between successive method invocations within a single conversation. For an Java implementation type, there are two possible strategies which may be used to handle this state data:¶
<#>The implementation can be built as a stateless piece of code (essentially, the code expects a new instance of the code to be used for each method invocation). The code must then be responsible for accessing the conversationID of the conversation, which is maintained by the SCA runtime code. The implementation is then responsible for persisting any necessary state data during the processing of a method and for accessing the persisted state data when required, all using the conversationID as a key.¶
<#>The implementation can be built as a stateful piece of code, which means that it stores any state data within the instance fields of the Java class. The implementation must then be declared as being of conversation scope using the @Scope annotation. This indicates to the SCA runtime that the implementation is stateful and that the runtime must perform correlation between client method invocations and a particular instance of the service implementation and that the runtime is also responsible for persisting and restoring the implementation instance if the runtime needs to clear the instance out of memory for any reason. (Note that ... [1]

# 7 Accessing a Callback Service

Java implementation classes that require a callback service may use @Callback to have a reference to the callback service associated with the current invocation injected on a public or protected field or setter method.

# 8 Component Type of a Java Implementation

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

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

- name attribute is the simple name of the interface (ie without the package name)
- requires attribute is omitted unless the @Service is also annotated with an @Requires - in this case, the requires attribute is present with a value equivalent to the intents declared by the @Requires annotation.
- policySets sttribute is omitted unless the @Service is also annotated with an @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by the @PolicySets annotation.
- interface child element is present with the interface attribute set to the fully qualified name of the interface class identified by the @Service annotation
- 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 is required 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 @Requires - in this case, the requires attribute is present with a value equivalent to the intents declared by the@Requires annotation.
- 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 child element with the interface attribute set to the fully qualified name of the interface class which types the field or setter method
- 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]

**Deleted:** name implied by

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 is required to have a name parameter)

Deleted: name implied by

- 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 an 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 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" unless the @Property annotation has required=false, in which case it is set to "false"

## 8.1 Component Type of an Implementation with no @Service annotations

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
- policySets sttribute is omitted
- interface 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
- binding child element is omitted
- callback child element is omitted

## 8.2 ComponentType of an Implementation with no @Reference or @Property annotations

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 child element the interface attribute set to the fully qualified name of the interface class which types the field or setter method
- 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

**Deleted:** name implied by the name of

**Deleted:** name implied by the name of

## 8.3 Java Implementation with conflicting setter methods

**Comment:** Issue 117

**Formatted:** Heading 2,H2

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 which results in either an SCA property or an SCA reference, the SCA runtime MUST raise an error and MUST NOT instantiate the implementation class.

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 {
    @Property
    public void setSomeProperty(String someProperty) {...}

    @Property
    public void setsomeProperty(String someProperty) {...}
}

/** Illegal since setter methods with same JavaBeans property name are
annotated with @Reference annotation. */
public class IllegalImpl2 {
    @Reference
    public void setSomeReference(SomeService service) {...}

    @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 {
    public void setSomeOtherProperty(String someProperty) {...}

    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 {
    public void setSomeOtherReference(SomeService service) {...}

    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 {
    @Property
    public void setFoo(String foo) {...}

    @Reference
    public void setfoo(SomeService service) {...}
}
```

Formatted: Normal

# 9 Specifying the Java Implementation Type in an Assembly

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

```
<implementation.java class="NCName" />
```

The implementation.java element has the following attributes:

- ***class (required)*** – the fully qualified name of the Java class of the implementation

**Comment:** Issue 87

**Deleted:** **<#>Specifying the Component Type¶**
For a Java implementation class, the component type is typically derived directly from introspection of the Java class .¶
A component type can optionally be specified in a side file. The component type side file is found with the same classloader that loaded the Java class. The side file must be located in a directory that corresponds to the namespace of the implementation and have the same name as the Java class, but with a .componentType extension instead of the .class extension.¶
The rules on how a component type side file adds to the component type information reflected from the component implementation are described as part of the SCA assembly model specification [1]. If the component type information is in conflict with the implementation, it is an error.¶
If the component type side file specifies a service interface using a WSDL interface, then the Java class should implement the interface that would be generated by the JAX-WS mapping of the WSDL to a Java interface. See the section 'WSDL 2 Java and Java 2 WSDL' in [JAVACAA].¶

# 10 Java Packaging and Deployment Model

The SCA Assembly 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 classloader boundary at runtime. That is, classes are loaded by a contribution specific classloader 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 requires the use of 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 is included in one and only one import.java element. Multiple packages can be imported, either through specifying multiple packages in the @package attribute or through the presence of multiple import.java elements.

The package used to satisfy an import MUST match the package name, the version number or version number range and (if present) the location specified on the import.java element.

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:

Comment: Issue 60

Formatted: Heading 1,Heading 1 Char,Heading 1 Char1 Char,Heading 1 Char Char Char

Formatted: Heading 2,H2

Formatted: Bullets and Numbering

```
<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 "'" (required as the list contains commas).

  The uses directive indicates that any SCA contribution that imports this package from this exporting contribution MUST also import the same version as is used by this exporting contribution of any of the packages contained in the uses directive. 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'"

Formatted: Bullets and Numbering

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 is included in one and only one export.java element. 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

Formatted: Bullets and Numbering

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

```
<?xml version="1.0" encoding="ASCII"?>
<contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712>
  …
    <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>
```

The same Java package MUST NOT be specified on more than one import.java element.

The same Java package MUST NOT be specified on more than one export.java element.

A Java package that is specified on an export element MUST be contained within that contribution.

## 10.2 Java Artifact Resolution

Formatted: Heading 2,H2

Within a contribution, Java classes MUST be 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.

Formatted: Bullets and Numbering

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

## 10.3 Classloader Model

Formatted: Heading 2,H2

The Java classes used by a contribution are all loaded by a class loader that is unique for each contribution in the Domain. Java classes that are imported into a contribution (as per section **Error! Reference source not found.** above) are loaded by the exporting contribution's class loader.

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

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

evaluates to true.

The thread context classloader of a component implementation class is set to the classloader of its containing contribution.

# 11 Conformance

The XML schema available at the namespace URI, defined by this specification, is considered to be authoritative and takes precedence over the XML Schema defined in the appendix of this document.

An SCA runtime MUST reject a composite file that does not conform to the sca-contribution-java.xsd schema.

Formatted: Heading 1,Heading 1 Char,Heading 1 Char1 Char,Heading 1 Char Char Char

Formatted: Font: Verdana, 9 pt, Complex Script Font: 9 pt

Formatted: Font: Verdana, 9 pt, Complex Script Font: 9 pt

Formatted: Font: Verdana, 9 pt, Complex Script Font: 9 pt

# 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/200712"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   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>
```

Formatted: Bullets and Numbering

Formatted: AppendixHeading2

Formatted: German Germany

Formatted: English U.S.

Formatted: Normal

Formatted: Font: (Default) Courier New, Complex Script Font: Courier New

Formatted: Bullets and Numbering

# B. Acknowledgements

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

**Participants:**

[Participant Name, Affiliation | Individual Member]

[Participant Name, Affiliation | Individual Member]

Formatted: Bullets and Numbering

# C. Non-Normative Text

# 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<br>* Editorial cleanup to make a working draft<br>- [1] style changed to [ASSEMBLY]<br>- updated namespace references |
| wd03 | 2009-02-26 | David Booz | • Accepted all changes from wd02<br>• Applied 60, 87, 117, 126<br>• Removed conversations |

Formatted: Bullets and Numbering

## *Conversational Implementation*

Java implementation classes that are CONVERSATION scoped may use @ConversationID to have the current conversation ID injected on a public or protected field or setter method. Alternatively, the Conversation API as defined in the Java Common Annotations and API Specification may be used to obtain the current conversation ID.

For the provider of a conversational service, there is the need to maintain state data between successive method invocations within a single conversation. For an Java implementation type, there are two possible strategies which may be used to handle this state data:

The implementation can be built as a stateless piece of code (essentially, the code expects a new instance of the code to be used for each method invocation). The code must then be responsible for accessing the conversationID of the conversation, which is maintained by the SCA runtime code. The implementation is then responsible for persisting any necessary state data during the processing of a method and for accessing the persisted state data when required, all using the conversationID as a key.

The implementation can be built as a stateful piece of code, which means that it stores any state data within the instance fields of the Java class. The implementation must then be declared as being of conversation scope using the @Scope annotation. This indicates to the SCA runtime that the implementation is stateful and that the runtime must perform correlation between client method invocations and a particular instance of the service implementation and that the runtime is also responsible for persisting and restoring the implementation instance if the runtime needs to clear the instance out of memory for any reason. (Note that conversations are potentially very long lived and that SCA runtimes may involve the use of clustered systems where a given instance object may be moved between nodes in the cluster over time, for load balancing purposes)