



Service Component Architecture Java Component Implementation Specification Version 1.1

Working Draft 03

26th February 2009

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>

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 Implementation Specification Version 1.00, 15 February 2007

This specification is related to:

- Service Component Architecture Assembly Model Specification Version 1.1
- Service Component Architecture Policy Framework Specification Version 1.1

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,

Deleted: 2

Deleted: 16

Deleted: December

Deleted: 8

Deleted: 2

Deleted: 2

Deleted: 2

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

Copyright © OASIS® 2005, 2009. All Rights Reserved.

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	5
1.1	Terminology	5
1.2	Normative References.....	5
1.3	Non-Normative References.....	5
2	Service.....	6
2.1	Use of @Service	6
2.2	Local and Remotable services	8
2.3	Introspecting services offered by a Java implementation	8
2.4	Non-Blocking Service Operations.....	9
2.5	Non-Conversational and Conversational Services	9
2.6	Callback Services.....	9
3	References	10
3.1	Reference Injection	10
3.2	Dynamic Reference Access	10
4	Properties	11
4.1	Property Injection	11
4.2	Dynamic Property Access	11
5	Implementation Instance Instantiation.....	12
6	Implementation Scopes and Lifecycle Callbacks	14
6.1	Conversational Implementation	14
7	Accessing a Callback Service	15
8	Component Type of a Java Implementation.....	16
8.1	Component Type of an Implementation with no @Service annotations.....	17
8.2	ComponentType of an Implementation with no @Reference or @Property annotations	17
9	Specifying the Java Implementation Type in an Assembly.....	21
10	Specifying the Component Type	21
A.	Acknowledgements.....	26
B.	Non-Normative Text.....	28
C.	Revision History.....	29

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

35 2 Service

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

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

- 39 • A Java interface
- 40 • A Java class
- 41 • A Java interface generated from a Web Services Description Language [WSDL] (WSDL)
42 portType.

43 Java implementation classes must implement all the operations defined by the service interface. If
44 the service interface is defined by a Java interface, the Java-based component can either
45 implement that Java interface, or implement all the operations of the interface.

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

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

52 2.1 Use of `@Service`

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

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

60 Interface:

```
61     public interface HelloService {  
62  
63         String hello(String message);  
64     }  
65
```

66 Implementation class:

```
67     @Service(HelloService.class)  
68     public class HelloServiceImpl implements HelloService {  
69  
70         public String hello(String message) {  
71             ...  
72         }  
73     }  
74
```

75 The XML representation of the component type for this implementation is shown below for
76 illustrative purposes. There is no need to author the component type as it can be reflected from
77 the Java class.

```

78
79 <?xml version="1.0" encoding="ASCII"?>
80 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
81
82     <service name="HelloService">
83         <interface.java interface="services.hello.HelloService"/>
84     </service>
85
86 </componentType>
87

```

88 The Java implementation class itself, as opposed to an interface, may also define a service offered
89 by a component. In this case, @Service may be used to explicitly declare the implementation class
90 defines the service offered by the implementation. In this case, a component will only offer
91 services declared by @Service. The following illustrates this:

```

92
93     @Service(HelloServiceImpl.class)
94     public class HelloServiceImpl implements AnotherInterface {
95
96         public String hello(String message) {
97             ...
98         }
99     ...
100 }
101

```

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

```

106 <?xml version="1.0" encoding="ASCII"?>
107 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
108
109     <service name="HelloServiceImpl">
110         <interface.java
111             interface="services.hello.HelloServiceImpl"/>
112     </service>
113
114 </componentType>
115

```

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

```

117
118     @Service(interfaces={HelloService.class, AnotherInterface.class})
119     public class HelloServiceImpl implements HelloService, AnotherInterface
120     {
121
122         public String hello(String message) {

```

```
123         ...
124     }
125     ...
126 }
```

127
128 The following snippet shows the introspected component type for this implementation.

```
129 <?xml version="1.0" encoding="ASCII"?>
130 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
131
132     <service name="HelloService">
133         <interface.java interface="services.hello.HelloService"/>
134     </service>
135     <service name="AnotherService">
136         <interface.java interface="services.hello.AnotherService"/>
137     </service>
138
139 </componentType>
```

140 2.2 Local and Remotable services

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

```
144     package services.hello;
145
146     @Remotable
147     public interface HelloService {
148
149         String hello(String message);
150     }
151
```

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

154
155 If an implementation class has implemented interfaces that are not decorated with an
156 `@Remotable` annotation, the class is considered to implement a single **local** service whose type is
157 defined by the class (note that local services may be typed using either Java interfaces or
158 classes).

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

161 2.3 Introspecting services offered by a Java implementation

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

165 *If the interfaces of the SCA services are not specified with the `@Service` annotation on the*
166 *implementation class, it is assumed that all implemented interfaces that have been annotated as*

167 *@Remotable* are the service interfaces provided by the component. If none of the implemented
168 interfaces is remotable, then by default the implementation offers a single service whose type is
169 the implementation class.

170 2.4 Non-Blocking Service Operations

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

174 2.5 Callback Services

175 A callback interface is declared by using the `@Callback` annotation on the service interface
176 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

177 3 References

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

181 3.1 Reference Injection

182 A Java implementation type may explicitly specify its references through the use of @Reference as
183 in the following example:

```
184  
185  
186     public class ClientComponentImpl implements Client {  
187         private HelloService service;  
188  
189         @Reference  
190         public void setHelloService(HelloService service) {  
191             this.service = service;  
192         }  
193     }  
194
```

195 If @Reference marks a public or protected setter method, the SCA runtime is required to provide
196 the appropriate implementation of the service reference contract as specified by the parameter
197 type of the method. This must done by invoking the setter method an implementation instance.
198 When injection occurs is defined by the scope of the implementation. However, it will always
199 occur before the first service method is called.

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

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

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

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

211 3.2 Dynamic Reference Access

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

215 4 Properties

216 4.1 Property Injection

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

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

```
222  
223  
224     public class ClientComponentImpl implements Client {  
225         private int maxRetries;  
226  
227         @Property  
228         public void setRetries(int maxRetries) {  
229             this.maxRetries = maxRetries;  
230         }  
231     }  
232
```

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

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

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

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

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

244 4.2 Dynamic Property Access

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

Comment: Issue 126

Deleted: Instantiation

247

5 Implementation Instance Creation

248

249

250

251

252

253

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.

Deleted: instantiate

Deleted: s

254

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

255

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

256

2. A declared constructor that unambiguously identifies all property and reference values.

257

3. A no-argument constructor.

258

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

259

260

261

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

262

- by name in the `@Constructor` annotation (if present)

263

- through the presence of a `@Property` or `@Reference` annotation on the parameter declaration

264

265

- by uniquely matching the parameter type to the type of a property or reference

266

267

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

268

269

- 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.

270

271

- The container may inject a proxy to the target service; invocation of methods on the proxy may result in a `ServiceUnavailableException`

272

273

The following are examples of legal Java component constructor declarations:

274

275

```
/** Simple class taking a single property value */
```

276

```
public class Impl1 {
```

277

```
    String someProperty;
```

278

```
    public Impl1(String propval) {...}
```

279

```
}
```

280

281

```
/** Simple class taking a property and reference in the constructor;
```

282

```
 * The values are not injected into the fields.
```

283

```
 */
```

284

```
public class Impl2 {
```

285

```
    public String someProperty;
```

286

```
    public SomeService someReference;
```

```

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

313
314 The Java implementation type supports all of the scopes defined in the Java Common Annotations
315 and API Specification: STATELESS, and COMPOSITE. Implementations specify their scope through
316 the use of the @Scope annotation as in:

```
317  
318     @Scope("COMPOSITE")  
319     public class ClientComponentImpl implements Client {  
320         // ...  
321     }
```

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

324 A Java component implementation specifies init and destroy callbacks by using @Init and
325 @Destroy respectively. For example:

```
326  
327     public class ClientComponentImpl implements Client {  
328  
329         @Init  
330         public void init() {  
331             //...  
332         }  
333  
334         @Destroy  
335         public void destroy() {  
336             //...  
337         }  
338     }  
339
```

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]

341

7 Accessing a Callback Service

342

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.

343

344

8 Component Type of a Java Implementation

345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389

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 attribute 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

Deleted: name implied by

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 \[JAVABEANS\]](#)

390 | corresponding to the setter method name, depending on what element of the class is annotated
391 | by the @Property (note: for a constructor parameter, the @Property annotation is required to
392 | have a name parameter)

Deleted: name implied by

- 393 | • value attribute is omitted
- 394 | • type attribute which is set to the XML type implied by the JAXB mapping of the Java type of the
395 | field or the Java type defined by the parameter of the setter method. Where the type of the field
396 | or of the setter method is an array, the element type of the array is used. Where the type of the
397 | field or of the setter method is an java.util.Collection, the parameterized type of the Collection or
398 | its member type is used. If the JAXB mapping is to a global element rather than a type (JAXB
399 | @XMLRootElement annotation), the type attribute is omitted.
- 400 | • element attribute is omitted unless the JAXB mapping of the Java type of the field or the Java
401 | type defined by the parameter of the setter method is to a global element (JAXB
402 | @XMLRootElement annotation). In this case, the element attribute has the value of the name of
403 | the XSD global element implied by the JAXB mapping.
- 404 | • many attribute set to "false" unless the type of the field or of the setter method is an array or a
405 | java.util.Collection, in which case it is set to "true".
- 406 | • mustSupply attribute set to "true" unless the @Property annotation has required=false, in which
407 | case it is set to "false"

408 8.1 Component Type of an Implementation with no @Service 409 annotations

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

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

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

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

- 421 | • name attribute is the simple name of the interface or the simple name of the class
- 422 | • requires attribute is omitted
- 423 | • policySets attribute is omitted
- 424 | • interface child element is present with the interface attribute set to the fully qualified name of the
425 | interface class or to the fully qualified name of the class itself
- 426 | • binding child element is omitted
- 427 | • callback child element is omitted

428

429 8.2 ComponentType of an Implementation with no @Reference or 430 @Property annotations

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

435

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

438 The following setter methods and fields are taken into consideration:

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

442

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

- 444 • its type is an interface annotated with @Remotable
- 445 • its type is an array where the element type of the array is an interface annotated with
446 @Remotable
- 447 • its type is a java.util.Collection where the parameterized type of the Collection or its
448 member type is an interface annotated with @Remotable

449 The reference in the component type has:

- 450 • name attribute with the value of the name of the field or the [JavaBeans property name](#)
451 [\[JAVABEANS\] corresponding to the setter method name](#)
- 452 • multiplicity attribute is (1..1) for the case where the type is an interface
453 multiplicity attribute is (1..n) for the cases where the type is an array or is a
454 java.util.Collection
- 455 • interface child element the interface attribute set to the fully qualified name of the
456 interface class which types the field or setter method
- 457
- 458 • all other attributes and child elements of the reference are omitted

Deleted: name implied by the name of

459

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

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

- 462 • name attribute with the value of the name of the field or the [JavaBeans property name](#)
463 [\[JAVABEANS\] corresponding to the setter method name](#)
- 464 • type attribute and element attribute set as described for a property declared via a
465 @Property annotation
- 466 • value attribute omitted
- 467 • many attribute set to "false" unless the type of the field or of the setter method is an array
468 or a java.util.Collection, in which case it is set to "true".
- 469 • mustSupply attribute set to true

Deleted: name implied by the name of

470

471 8.3 Java Implementation with conflicting setter methods

Comment: Issue 117

Formatted: Heading 2,H2

472 If a Java implementation class, with or without @Property and @Reference annotations, has more than
473 one setter method with the same JavaBeans property name [JAVABEANS] corresponding to the setter
474 method name which results in either an SCA property or an SCA reference, the SCA runtime MUST raise
475 an error and MUST NOT instantiate the implementation class.

476

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

479

```

480 /** Illegal since two setter methods with same JavaBeans property name
481 are annotated with @Property annotation. Note the difference in case
482 between the two setter method names.*/
483 public class IllegalImpl1 {
484     @Property
485     public void setSomeProperty(String someProperty) {...}
486
487     @Property
488     public void setsomeProperty(String someProperty) {...}
489 }
490
491 /** Illegal since setter methods with same JavaBeans property name are
492 annotated with @Reference annotation. Note the difference in case
493 between the two setter method names.*/
494 public class IllegalImpl2 {
495     @Reference
496     public void setSomeReference(SomeService service) {...}
497
498     @Reference
499     public void setsomeReference(SomeService service) {...}
500 }
501
502 /** Illegal since two setter methods with same JavaBeans property name
503 are resulting in an SCA property. Implementation has no @Property or
504 @Reference annotations. Note the difference in case between the two
505 setter method names.*/
506 public class IllegalImpl3 {
507     public void setSomeOtherProperty(String someProperty) {...}
508
509     public void setsomeOtherProperty(String someProperty) {...}
510 }
511
512 /** Illegal since two setter methods with same JavaBeans property name
513 are resulting in an SCA reference. Implementation has no @Property or
514 @Reference annotations. Note the difference in case between the two
515 setter method names.*/
516 public class IllegalImpl4 {
517     public void setSomeOtherReference(SomeService service) {...}
518
519     public void setsomeOtherReference(SomeService service) {...}
520 }
521

```

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

```
525 |  
526 | /** Two setter methods with same JavaBeans property name, but one is  
527 | annotated with @Property and the other is annotated with @Reference  
528 | annotation. */  
529 | public class WeirdButLegalImpl {  
530 |     @Property  
531 |     public void setFoo(String foo) {...}  
532 |  
533 |     @Reference  
534 |     public void setfoo(SomeService service) {...}  
535 | }  
536 |
```

← - - - - Formatted: Normal

537 **9 Specifying the Java Implementation Type in an**
538 **Assembly**

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

540

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

542

543 The implementation.java element has the following attributes:

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

545

546

547

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 \[JAVACAA\]](#).¶

Comment: Issue 60

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

548

10 Java Packaging and Deployment Model

549
550
551

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.

552
553
554
555
556

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.

Formatted: Heading 2,H2

557

10.1 Contribution Metadata Extensions

558
559
560
561
562
563
564

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:

565

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

566

567

The import.java element has the following attributes:

568
569
570

- **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 ",".

Formatted: Bullets and Numbering

571

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:

575
576

```
package="com.acme.package1;version=1.4.1"  
package="com.acme.package2;version=[1.2.1.3]"
```

577

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

578

[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

579

(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.

581

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)

582

583

584

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.

585

586

587

588

589

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

590

591

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.

592

593

594

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.

595

596

597

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:

598

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

600

601 The export.java element has the following attributes:

602 • **package : string (0..1)** – The name of one or more Java package(s) to expose for sharing by Formatted: Bullets and Numbering
603 another contribution. Where there is more than one package, the package names are
604 separated by a comma ",".

605 The package can have a **version number** appended to it, separated from the package name
606 by a semicolon ":" followed by the text "version=" and the version number:
607 package="com.acme.package1;version=1.4.1"

608
609 The package can have a **uses directive** appended to it, separated from the package name by
610 a semicolon ":" followed by the text "uses=" which is then followed by a list of package names
611 contained within single quotes "" (required as the list contains commas).

612
613 The uses directive indicates that any SCA contribution that imports this package from this
614 exporting contribution MUST also import the same version as is used by this exporting
615 contribution of any of the packages contained in the uses directive. Typically, the packages in
616 the uses directive are packages used in the interface to the package being exported (eg as
617 parameters or as classes/interfaces that are extended by the exported package). Example:

618
619 package="com.acme.package1;uses='com.acme.package2,com.acme.package3'"
620

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

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

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

627 For example, a contribution that wants to:

- 628 • use classes from the *some.package* package from another contribution (any version) Formatted: Bullets and Numbering
- 629 • use classes of the *some.other.package* package from another contribution, at exactly version
630 2.0.0
- 631 • expose the *my.package* package from its own contribution, with version set to 1.0.0

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

633

634 `<?xml version="1.0" encoding="ASCII"?>`

635 `<contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712>`

636 `...`

637 `<import.java package="some.package"/>`

638 `<import.java package="some.other.package;version=[2.0.0]"/>`

639 `<export.java package="my.package;version=1.0.0"/>`

640 `</contribution>`

641

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

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

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

645

646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662

663
664
665
666
667
668
669
670
671
672
673

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.

Formatted: Heading 2,H2

10.3 Classloader Model

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.

674
675
676
677
678
679
680

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

681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714

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

715

B. Acknowledgements

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

718 **Participants:**

719 [Participant Name, Affiliation | Individual Member]

720 [Participant Name, Affiliation | Individual Member]

721

C. Non-Normative Text

723

D. Revision History

724

[optional; should not be included in OASIS Standards]

725

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	<ul style="list-style-type: none"> • Accepted all changes from wd02 • Applied 60, 87, 117, 126 • Removed conversations

Formatted: Bullets and Numbering

726

727

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)