



Service Component Architecture Java Component Implementation Specification Version 1.1 ~~109~~ +simon

Working Draft 108

3027th April 2009

Specification URIs:

This Version:

<http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec-wd08.html>
<http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec-wd08.doc>
<http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec-wd08.pdf>

Previous Version:

Latest Version:

<http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec.html>
<http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec.doc>
<http://docs.oasis-open.org/sca-j/sca-javaci-1.1-spec.pdf>

Latest Approved Version:

Technical Committee:

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

Chair(s):

David Booz, IBM
Mark Combellack, Avaya

Editor(s):

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

Related work:

This specification replaces or supersedes:

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

This specification is related to:

- Service Component Architecture Assembly Model Specification Version 1.1
- Service Component Architecture Policy Framework Specification Version 1.1
- [Service Component Architecture Java Common Annotations and APIs Specification Version 1.1](#)

Declared XML Namespace(s):

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

Abstract:

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

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

Status:

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

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

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

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

Notices

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

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.....	8
2.5	Callback Services.....	8
3	References.....	9
3.1	Reference Injection.....	9
3.2	Dynamic Reference Access.....	9
4	Properties.....	10
4.1	Property Injection.....	10
4.2	Dynamic Property Access.....	10
5	Implementation Instance Creation.....	11
6	Implementation Scopes and Lifecycle Callbacks.....	13
7	Accessing a Callback Service.....	14
8	Component Type of a Java Implementation.....	15
8.1	Component Type of an Implementation with no @Service annotations.....	16
8.2	ComponentType of an Implementation with no @Reference or @Property annotations.....	17
8.3	Component Type Introspection Examples.....	18
8.4	Java Implementation with conflicting setter methods.....	19
9	Specifying the Java Implementation Type in an Assembly.....	21
10	Java Packaging and Deployment Model.....	22
10.1	Contribution Metadata Extensions.....	22
10.2	Java Artifact Resolution.....	24
10.3	Class loader Model.....	24
11	Conformance.....	25
11.1	SCA Java Component Implementation Composite Document.....	25
11.2	SCA Java Component Implementation Contribution Document.....	25
11.3	SCA Runtime.....	25
A.	XML Schemas.....	26
A.1	sca-contribution-java.xsd.....	26
A.2	sca-implementation-java.xsd.....	26
B.	Conformance Items.....	28
C.	Acknowledgements.....	31
D.	Non-Normative Text.....	33
E.	Revision History.....	34

1 Introduction

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

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

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

1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.2 Normative References

- [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.
- [ASSEMBLY] SCA Assembly [Model Specification Version 1.1](#), <http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf><http://www.oasis-open.org/committees/download.php/31722/sca-assembly-1.1-spec-cd03.pdf>
- [POLICY] SCA Policy [Framework Specification Version 1.1](#), <http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd02.pdf><http://www.oasis-open.org/committees/download.php/31608/sca-policy-1.1-spec-cd02.pdf>
- [JAVACAA] SCA Java Common Annotations and APIs [Specification Version 1.1](#), <http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01.pdf>[DAB1]
- [WSDL] WSDL Specification, WSDL 1.1: <http://www.w3.org/TR/wSDL>
- [OSGi Core] OSGi Service Platform Core Specification, Version 4.0.1 <http://www.osgi.org/download/r4v41/r4.core.pdf>
- [JAVABEANS] JavaBeans 1.01 Specification, <http://java.sun.com/javase/technologies/desktop/javabeans/api/>

1.3 Non-Normative References

- TBD TBD

2 Service

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

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

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

[JCI20001]

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

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

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

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

2.1 Use of @Service

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

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

Interface:

```
package services.hello;
public interface HelloService {
    String hello(String message);
}
```

87 Implementation class:

```
88 @Service(HelloService.class)
89 public class HelloServiceImpl implements HelloService {
90
91     public String hello(String message) {
92         ...
93     }
94 }
95
```

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

```
98
99 <?xml version="1.0" encoding="UTF-8"?>
100 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903">
101     <service name="HelloService">
102         <interface.java interface="services.hello.HelloService"/>
103     </service>
104 </componentType>
105
106 </componentType>
107
```

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

```
112
113 package services.hello;
114
115 @Service(HelloServiceImpl.class)
116 public class HelloServiceImpl implements AnotherInterface {
117
118     public String hello(String message) {
119         ...
120     }
121     ...
122 }
```

123
124 In the above example, HelloServiceImpl offers one service as defined by the public methods of the
125 implementation class. The interface AnotherInterface in this case does not specify a service offered by
126 the component. The following is an XML representation of the introspected component type:

```
127 <?xml version="1.0" encoding="UTF-8"?>
128 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903">
129     <service name="HelloServiceImpl">
130         <interface.java interface="services.hello.HelloServiceImpl"/>
131     </service>
132 </componentType>
133
134 </componentType>
135
```

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

138

```

139     @Service(interfaces={HelloService.class, AnotherInterface.class})
140     public class HelloServiceImpl implements HelloService, AnotherInterface
141     {
142
143         public String hello(String message) {
144             ...
145         }
146         ...
147     }
148

```

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

```

150     <?xml version="1.0" encoding="UTF-8"?>
151     <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903">
152
153         <service name="HelloService">
154             <interface.java interface="services.hello.HelloService"/>
155         </service>
156         <service name="AnotherService">
157             <interface.java interface="services.hello.AnotherService"/>
158         </service>
159
160     </componentType>

```

161 2.2 Local and Remotable Services

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

```

166     package services.hello;
167
168     @Remotable
169     public interface HelloService {
170
171         String hello(String message);
172     }
173

```

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

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

179 2.3 Introspecting Services Offered by a Java Implementation

180 The services offered by a Java implementation class are determined through introspection, as defined
181 in the section "[Component Type of a Java Implementation](#)".

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

188 **2.4 Non-Blocking Service Operations**

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

193 **2.5 Callback Services**

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

198 3 References

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

202 3.1 Reference Injection

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

```
205  
206     public class ClientComponentImpl implements Client {  
207         private HelloService service;  
208  
209         @Reference  
210         public void setHelloService(HelloService service) {  
211             this.service = service;  
212         }  
213     }  
214
```

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

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

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

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

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

234 3.2 Dynamic Reference Access

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

238

4 Properties

239

4.1 Property Injection

240

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

241

242

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

245

246

247

248

249

250

251

252

253

254

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

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

255

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

256

257

258

259

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

260

261

262

263

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

264

265

266

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

267

268

269

270

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

271

272

273

4.2 Dynamic Property Access

274

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

275

276

277

5 Implementation Instance Creation

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

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

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

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

300 [JCI50004]

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

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

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

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

- 309 • If any reference in the cycle is optional, then the container can inject a null value during
310 construction, followed by injection of a reference to the target before invoking any service.
- 311 • The container can inject a proxy to the target service; invocation of methods on the proxy can
312 result in a ServiceUnavailableException. Cyclic references between components MUST be handled by the
313 SCA runtime in one of two ways:
 - 314 • If any reference in the cycle is optional, then the container can inject a null value during
315 construction, followed by injection of a reference to the target before invoking any service.
 - 316 • The container can inject a proxy to the target service; invocation of methods on the proxy can
317 result in a ServiceUnavailableException. Cyclic references between components MUST be handled by the
318 SCA runtime in one of two ways:
 - 319 • If any reference in the cycle is optional, then the container can inject a null value during
320 construction, followed by injection of a reference to the target before invoking any service.
 - 321 • The container can inject a proxy to the target service; invocation of methods on the proxy can
322 result in a ServiceUnavailableException.

323 [JCI50003]

324 The following are examples of legal Java component constructor declarations:

```
325     /** Constructor declared using @Constructor annotation */
326     public class Impl1 {
327         private String someProperty;
328         @Constructor
329         public Impl1( @Property("someProperty") String propval ) {...}
330     }
331
332     /** Declared constructor unambiguously identifying all Property */
333     /** -and Reference values */
334     public class Impl2 {
335         private String someProperty;
336         private SomeService someReference;
337         public Impl2( @Property("someProperty") String a,
338                     @Reference("someReference") SomeService b )
339             {...}
340     }
341
342
343     /** Declared constructor unambiguously identifying all Property */
344     /** and Reference values
345     plus an additional Property injected */
346     /** via a setter method */
347     public class Impl3 {
348         private String someProperty;
349         private String anotherProperty;
350         private SomeService someReference;
351         public Impl3( @Property("someProperty") String a,
352                     @Reference("someReference") SomeService b )
353             {...}
354         @Property
355         public void setAnotherProperty( String anotherProperty ) {...}
356     }
357
358     /** No-arg constructor */
359     public class Impl4 {
360         @Property
361         public String someProperty;
362         @Reference
363         public SomeService someReference;
```

```
364         public Impl4() {...}
365     }
366
367     /** Unannotated implementation with no-arg constructor */
368     public class Impl5 {
369         public String someProperty;
370         public SomeService someReference;
371         public Impl5() {...}
372     }
```

373 6 Implementation Scopes and Lifecycle Callbacks

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

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

```
378     @Scope ("COMPOSITE")  
379     public class ClientComponentImpl implements Client {  
380         // ...  
381     }  
382 }
```

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

385 A Java component implementation specifies init and destroy [methodscallbacks](#) by using the @Init and
386 @Destroy annotations respectively, as described in the Java Common Annotations and APIs
387 specification [JAVACAA].

388 For example:

```
389     public class ClientComponentImpl implements Client {  
390  
391         @Init  
392         public void init() {  
393             //...  
394         }  
395  
396         @Destroy  
397         public void destroy() {  
398             //...  
399         }  
400     }  
401 }
```

402 **7 Accessing a Callback Service**

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

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

409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453

8 Component Type of a Java Implementation

An SCA runtime MUST introspect the componentType of a Java implementation class following the rules defined in the section "Component Type of a Java Implementation". [JC180001]

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

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

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

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

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

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

- 456 • binding child element is omitted
- 457 • callback child element is omitted

458

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

- 460 • name attribute has the value of the name parameter of the @Property annotation, if present,
461 otherwise it is the name of the field or the JavaBeans property name [JAVABEANS]
462 corresponding to the setter method name, depending on what element of the class is annotated
463 by the @Property (note: for a constructor parameter, the @Property annotation needs to have a
464 name parameter)
- 465 • value attribute is omitted
- 466 • type attribute which is set to the XML type implied by the JAXB mapping of the Java type of the
467 field or the Java type defined by the parameter of the setter method. Where the type of the field
468 or of the setter method is an array, the element type of the array is used. Where the type of the
469 field or of the setter method is a java.util.Collection, the parameterized type of the Collection or
470 its member type is used. If the JAXB mapping is to a global element rather than a type (JAXB
471 @XMLRootElement annotation), the type attribute is omitted.
- 472 • element attribute is omitted unless the JAXB mapping of the Java type of the field or the Java
473 type defined by the parameter of the setter method is to a global element (JAXB
474 @XMLRootElement annotation). In this case, the element attribute has the value of the name of
475 the XSD global element implied by the JAXB mapping.
- 476 • many attribute is set to "false" unless the type of the field or of the setter method is an array or a
477 java.util.Collection, in which case it is set to "true".
- 478 • mustSupply attribute is set to "true" unless the @Property annotation has required=false, in which
479 case it is set to "false"

480

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

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

489 8.1 Component Type of an Implementation with no @Service 490 Annotations

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

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

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

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

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

515 8.2 ComponentType of an Implementation with no @Reference or

516 @Property Annotations

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

521

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

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

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

528

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

- 530 • its type is an interface annotated with @Remotable
- 531 • its type is an array where the element type of the array is an interface annotated with
- 532 @Remotable
- 533 • its type is a java.util.Collection where the parameterized type of the Collection or its
- 534 member type is an interface annotated with @Remotable

535 The reference in the component type has:

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

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

552

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

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

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

563 8.3 Component Type Introspection Examples

564 Example 8.1 shows how intent annotations can be applied to service and reference interfaces and

565 methods as well as to a service implementation class.

```

566 // Service interface
567 package test;
568 import org.oasisopen.sca.annotation.Authentication;
569 import org.oasisopen.sca.annotation.Confidentiality;
570
571 @Authentication
572 public interface MyService {
573     @Confidentiality
574     void mymethod();
575 }
576
577 // Reference interface
578 package test;
579 import org.oasisopen.sca.annotation.Integrity;
580
581 public interface MyRefInt {
582     @Integrity
583     void mymethod1();
584 }
585
586 // Service implementation class
587 package test;
588 import static org.oasisopen.sca.Constants.SCA_PREFIX;
589 import org.oasisopen.sca.annotation.Confidentiality;
590 import org.oasisopen.sca.annotation.Reference;
591 import org.oasisopen.sca.annotation.Service;
592 @Service(MyService.class)
593 @Requires(SCA_PREFIX+"managedTransaction")
594 public class MyServiceImpl {
595     @Confidentiality
596     @Reference
597     protected MyRefInt myRef;
598
599     public void mymethod() {...}

```

600 }

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

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

```
604 <componentType xmlns:sca=  
605     "http://docs.oasis-open.org/ns/opencsa/sca/200903">  
606     <implementation.java class="test.MyServiceImpl"  
607         requires="sca:managedTransaction"/>  
608     <service name="MyService" requires="sca:managedTransaction">  
609         <interface.java interface="test.MyService"/>  
610     </service>  
611     <reference name="myRef" requires="sca:confidentiality">  
612         <interface.java interface="test.MyRefInt"/>  
613     </reference>  
614 </componentType>
```

615 Example 8.2. Introspected component type with intents.

616 8.4 Java Implementation with **C**onflicting **S**etter **M**ethods

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

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

624

```
625     /** Illegal since two setter methods with same JavaBeans property name  
626     are annotated with @Property annotation. */  
627     public class IllegalImpl1 {  
628         // Setter method with upper case initial letter 'S'  
629         @Property  
630         public void setSomeProperty(String someProperty) {...}  
631  
632         // Setter method with lower case initial letter 's'  
633         @Property  
634         public void setsomeProperty(String someProperty) {...}  
635     }  
636  
637     /** Illegal since setter methods with same JavaBeans property name are  
638     annotated with @Reference annotation. */  
639     public class IllegalImpl2 {  
640         // Setter method with upper case initial letter 'S'  
641         @Reference  
642         public void setSomeReference(SomeService service) {...}  
643  
644         // Setter method with lower case initial letter 's'  
645         @Reference  
646         public void setsomeReference(SomeService service) {...}  
647     }  
648  
649     /** Illegal since two setter methods with same JavaBeans property name  
650     are resulting in an SCA property. Implementation has no @Property or  
651     @Reference annotations. */  
652     public class IllegalImpl3 {
```

```

653         // Setter method with upper case initial letter 'S'
654         public void setSomeOtherProperty(String someProperty) {...}
655
656         // Setter method with lower case initial letter 's'
657         public void setsomeOtherProperty(String someProperty) {...}
658     }
659
660     /** Illegal since two setter methods with same JavaBeans property name
661     are resulting in an SCA reference. Implementation has no @Property or
662     @Reference annotations. */
663     public class IllegalImpl4 {
664         // Setter method with upper case initial letter 'S'
665         public void setSomeOtherReference(SomeService service) {...}
666
667         // Setter method with lower case initial letter 's'
668         public void setsomeOtherReference(SomeService service) {...}
669     }
670

```

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

```

674
675     /** Two setter methods with same JavaBeans property name, but one is
676     annotated with @Property and the other is annotated with @Reference
677     annotation. */
678     public class WeirdButLegalImpl {
679         // Setter method with upper case initial letter 'F'
680         @Property
681         public void setFoo(String foo) {...}
682
683         // Setter method with lower case initial letter 'f'
684         @Reference
685         public void setfoo(SomeService service) {...}
686     }
687

```

688 9 Specifying the Java Implementation Type in an 689 Assembly

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

692

```
693 <implementation.java class="xs:NCName"  
694             requires="list of xs:QName"?  
695             policySets="list of xs:QName"?/>  
696
```

697 The implementation.java element has the following attributes:

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

703

704 The <implementation.java> element MUST conform to the schema defined in sca-implementation-
705 java.xsd. [\[JCI90001\]](#)

706

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

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

712

10 Java Packaging and Deployment Model

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

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

721 10.1 Contribution Metadata Extensions

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

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

730

731 The `import.java` element has the following attributes:

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

735

736 The package can have a **version number range** appended to it, separated from the package
737 name by a semicolon ";" followed by the text "version=" and the version number range, for
738 example:

```
739 package="com.acme.package1;version=1.4.1"
```

```
740 package="com.acme.package2;version=[1.2,1.3]"
```

741

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

743

744 [1.2,1.3] - enclosing square brackets - inclusive range meaning any version in the range from
745 the lowest to the highest, including the lowest and the highest

746 (1.3.1,2.4.1) - enclosing round brackets - exclusive range meaning any version in the range
747 from the lowest to the highest but not including the lowest or the highest.

748 1.4.1 - no enclosing brackets - implies any version at or later than the specified version
749 number is acceptable - equivalent to [1.4.1, infinity)

750

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

753

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

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

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

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

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

766

767 The export.java element has the following attributes:

- 768 • **package : string (01..1)** – The name of one or more Java package(s) to expose for sharing
769 by another contribution. Where there is more than one package, the package names are
770 separated by a comma ",".

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

774

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

778

779 The uses directive indicates that the SCA runtime MUST ensure that any SCA contribution that
780 imports this package from this exporting contribution also imports the same version as is used by
781 this exporting contribution of any of the packages contained in the uses directive. [JCI100003]

782 Typically, the packages in the uses directive are packages used in the interface to the package
783 being exported (eg as parameters or as classes/interfaces that are extended by the exported
784 package). Example:

785

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

787

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

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

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

795 For example, a contribution that wants to:

- 796 • use classes from the *some.package* package from another contribution (any version)
- 797 • use classes of the *some.other.package* package from another contribution, at exactly version
798 2.0.0
- 799 • expose the *my.package* package from its own contribution, with version set to 1.0.0

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

801

```
802 <?xml version="1.0" encoding="UTF-8"?>  
803 <contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200903>  
804 ...  
805 <import.java package="some.package"/>  
806 <import.java package="some.other.package;version=[2.0.0]"/>  
807 <export.java package="my.package;version=1.0.0"/>  
808 </contribution>
```

809

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

812

813 10.2 Java Artifact Resolution

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

857 [JCI100008]

858 10.3 Class Loader Model

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

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

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

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

866 evaluates to true.

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

869

870 11 Conformance

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

874
875 There are three categories of artifacts that this specification defines conformance for: SCA Java
876 Component Implementation Composite Document, SCA Java Component Implementation Contribution
877 Document and SCA Runtime.

878 11.1 SCA Java Component Implementation Composite Document

879 An SCA Java Component Implementation Composite Document is an SCA Composite Document, as
880 defined by the SCA Assembly [Model Specification](#) Section 13.1 [ASSEMBLY], that uses the
881 <implementation.java> element. Such an SCA Java Component Implementation Composite Document
882 MUST be a conformant SCA Composite Document, as defined by [ASSEMBLY], and MUST comply with
883 the requirements specified in Section 9 of this specification.

884 11.2 SCA Java Component Implementation Contribution Document

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

890 11.3 SCA Runtime

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

- 892
893 1. The implementation MUST meet all the conformance requirements defined by the SCA
894 Assembly Model Specification [ASSEMBLY].
- 895 2. The implementation MUST reject an SCA Java Composite Document that does not conform to
896 the sca-implementation-java.xsd schema.
- 897 3. The implementation MUST reject an SCA Java Contribution Document that does not conform to
898 the sca-contribution-java.xsd schema.
- 899 4. The implementation MUST meet all the conformance requirements, specified in 'Section 11
900 Conformance', from the SCA Java Common Annotations and APIs Specification [JAVACAA].
- 901 5. This specification permits an implementation class to use any and all the APIs and annotations
902 defined in the Java Common Annotations and APIs Specification [JAVACAA], therefore the
903 implementation MUST comply with all the statements in Appendix B: Conformance Items of
904 [JAVACAA], notably all mandatory statements have to be implemented.
- 905 6. The implementation MUST comply with all statements related to an SCA Runtime, specified in
906 'Appendix B~~X~~: Conformance Items' of this specification, notably all mandatory statements have to
907 be implemented.

908

909

A. XML Schemas

910

A.1 sca-contribution-java.xsd

```
911 <?xml version="1.0" encoding="UTF-8"?>
912 <!-- Copyright(C) OASIS(R) 2005,2009. All Rights Reserved. OASIS trademark,
913 IPR and other policies apply. -->
914 <schema xmlns="http://www.w3.org/2001/XMLSchema"
915         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200903"
916         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200903"
917         elementFormDefault="qualified">
918
919     <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>
920
921     <!-- Import.java -->
922     <element name="import.java" type="sca:JavaImportType"/>
923     <complexType name="JavaImportType">
924         <complexContent>
925             <extension base="sca:Import">
926                 <attribute name="package" type="NCName" use="required"/>
927                 <attribute name="location" type="anyURI" use="optional"/>
928             </extension>
929         </complexContent>
930     </complexType>
931
932     <!-- Export.java -->
933     <element name="export.java" type="sca:JavaExportType"/>
934     <complexType name="JavaExportType">
935         <complexContent>
936             <extension base="sca:Export">
937                 <attribute name="package" type="NCName" use="required"/>
938             </extension>
939         </complexContent>
940     </complexType>
941
942 </schema>
```

943

A.2 sca-implementation-java.xsd

```
944 <?xml version="1.0" encoding="UTF-8"?>
945 <!-- Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
946 OASIS trademark, IPR and other policies apply. -->
947 <schema xmlns="http://www.w3.org/2001/XMLSchema"
948         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200903"
949         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200903"
950         elementFormDefault="qualified">
951
952     <include schemaLocation="sca-core-1.1-cd03.xsd"/>
953
954     <!-- Java Implementation -->
955     <element name="implementation.java" type="sca:JavaImplementation"
956             substitutionGroup="sca:implementation"/>
957     <complexType name="JavaImplementation">
958         <complexContent>
959             <extension base="sca:Implementation">
```

```
960         <sequence>
961             <any namespace="##other" processContents="lax" minOccurs="0"
962                 maxOccurs="unbounded"/>
963         </sequence>
964         <attribute name="class" type="NCName" use="required"/>
965         <anyAttribute namespace="##other" processContents="lax"/>
966     </extension>
967 </complexContent>
968 </complexType>
969
970 </schema>
```

971

B. Conformance Items

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

974

Conformance ID	Description
[JCI20001]	The services provided by a Java-based implementation MUST have an interface defined in one of the following ways: <ul style="list-style-type: none"> • A Java interface • A Java class • A Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType.
[JCI20002]	Java implementation classes MUST implement all the operations defined by the service interface.
[JCI50001]	A Java implementation class MUST provide a public or protected constructor that can be used by the SCA runtime to create the implementation instance.
[JCI50002]	The @Constructor annotation MUST only be specified on one constructor; the SCA container MUST raise an error if multiple constructors are annotated with @Constructor.
[JCI50003]	Cyclic references between components MUST be handled by the SCA runtime in one of two ways: <ul style="list-style-type: none"> • If any reference in the cycle is optional, then the container can inject a null value during construction, followed by injection of a reference to the target before invoking any service. • The container can inject a proxy to the target service; invocation of methods on the proxy can result in a ServiceUnavailableException
[JCI50004]	The constructor to use for the creation of an implementation instance MUST be selected by the SCA runtime using the sequence: <ol style="list-style-type: none"> 1. A declared constructor annotated with a @Constructor annotation. 2. A declared constructor, all of whose parameters are annotated with either @Property or @Reference. 3. A no-argument constructor.
[JCI50005]	The SCA runtime MUST raise an error if there are multiple constructors that are not annotated with @Constructor and have a non-empty parameter list with all parameters annotated with either @Property or @Reference.
[JCI60001]	The SCA runtime MUST support the STATELESS and COMPOSITE implementation scopes.
[JCI80001]	An SCA runtime MUST introspect the componentType of a Java implementation class following the rules defined in the section "Component Type of a Java Implementation".
[JCI80002]	If a Java implementation class, with or without @Property and @Reference annotations, has more than one setter method with the same JavaBeans property name [JAVABEANS] corresponding to the setter method name, then if more than

	one method is inferred to set the same SCA property or to set the same SCA reference, the SCA runtime MUST raise an error and MUST NOT instantiate the implementation class.
[JCI90001]	The <implementation.java> element MUST conform to the schema defined in sca-implementation-java.xsd.
[JCI90002]	The fully qualified name of the Java class referenced by the @class attribute of <implementation.java/> MUST resolve to a Java class, using the artifact resolution rules defined in Section 10.2, that can be used as a Java component implementation.
[JCI90003]	The Java class referenced by the @class attribute of <implementation.java/> MUST conform to Java SE version 5.0.
[JCI100001]	Each Java package that is imported into the contribution MUST be included in one and only one import.java element.
[JCI100002]	The SCA runtime MUST ensure that the package used to satisfy an import matches the package name, the version number or version number range and (if present) the location specified on the import.java element.
[JCI100003]	The uses directive indicates that the SCA runtime MUST ensure that any SCA contribution that imports this package from this exporting contribution also imports the same version as is used by this exporting contribution of any of the packages contained in the uses directive.
[JCI100004]	Each Java package that is exported from the contribution MUST be included in one and only one export.java element.
[JCI100007]	A Java package that is specified on an export element MUST be contained within the contribution containing the export element.
[JCI100008]	<p>The SCA runtime MUST ensure that within a contribution, Java classes are resolved according to the following steps in the order specified:</p> <ol style="list-style-type: none"> 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: <ol style="list-style-type: none"> 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. <p>If the java package is not found, continue to step 3.</p> 3. The contribution itself is searched using the archive resolution rules defined by the Java Language.
[JCI100009]	The SCA runtime MUST set the thread context class loader of a component implementation class to the class loader of its containing contribution.
[JCI100010]	The SCA runtime MUST ensure that the Java classes used by a contribution are all loaded by a class loader that is unique for each contribution in the Domain.
[JCI100011]	The SCA runtime MUST ensure that Java classes that are imported into a

975

	contribution are loaded by the exporting contribution's class loader
--	--

976

C. Acknowledgements

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

979 **Participants:**

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

Scott Vorthmann
Feng Wang
Robin Yang

TIBCO Software Inc.
Primeton Technologies, Inc.
Primeton Technologies, Inc.

980

D. Non-Normative Text

982

E. Revision History

983

[optional; should not be included in OASIS Standards]

984

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, 123
wd04	2009-03-20	Mike Edwards	Accepted all changes from wd03 Issue 105 - RFC 2119 Language added - covers most of the specification. Accepted all changes after RFC 2119 language added. Editorial fix to ensure the term "class loader" is used consistently
wd05	2009-03-24	David Booz	Applied resolution for issues: 119, 137
wd06	2009-03-27	David Booz	Accepted all previous changes and applied issues 145,146,147,151
wd07	2009-04-06	David Booz	Editorial cleanup, namespace changes, changed XML encoding to UTF-8 in examples, applied 144
wd08	2009-04-27	David Booz	Applied issue 98, 152
<u>wd09</u>	<u>2009-04-29</u>	<u>David Booz</u>	<u>Editorial fixes throughout (capitalization, quotes, fonts, spec references, etc.)</u>
<u>wd10</u>	<u>2009-04-30</u>	<u>David Booz</u>	<u>Editorial fixes, indention, etc.</u>

985

986

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