



Service Component Architecture POJO Component Implementation Specification Version 1.1

Committee Specification Draft 03

8 November 2010

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csd03.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csd03.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csd03.pdf> (Authoritative)

Previous Version:

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

Latest Version:

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

Technical Committee:

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

Chair(s):

David Booz, IBM
Anish Karmarkar, Oracle Corporation

Editor(s):

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

Related work:

This specification replaces or supersedes:

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

This specification is related to:

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

Declared XML Namespace(s):

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

Abstract:

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

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

Status:

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

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

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

Citation Format:

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

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

Notices

Copyright © OASIS® 2005, 2010. 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", "SCA" and "Service Component Architecture" are trademarks of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <http://www.oasis-open.org/who/trademark.php> for above guidance.

Table of Contents

1	Introduction	6
1.1	Terminology.....	6
1.2	Normative References.....	6
2	Service	7
2.1	Use of @Service.....	7
2.2	Local and Remotable Services.....	9
2.3	Introspecting Services Offered by a Java Implementation.....	11
2.4	Non-Blocking Service Operations.....	11
2.5	Callback Services	11
3	References.....	12
3.1	Reference Injection.....	12
3.2	Dynamic Reference Access.....	12
4	Properties.....	13
4.1	Property Injection.....	13
4.2	Dynamic Property Access	13
5	Implementation Instance Creation.....	14
6	Implementation Scopes and Lifecycle Callbacks	16
7	Accessing a Callback Service	17
8	Component Type of a Java Implementation.....	18
8.1	Component Type of an Implementation with no @Service, @Reference or @Property Annotations	19
8.2	Impact of JAX-WS Annotations on ComponentType	21
8.2.1	@WebService	21
8.2.2	@WebMethod	21
8.2.3	@WebParam.....	21
8.2.4	@WebResult	22
8.2.5	@SOAPBinding.....	22
8.2.6	@WebServiceProvider	22
8.2.7	Web Service Binding.....	22
8.3	Component Type Introspection Examples.....	23
8.4	Java Implementation with Conflicting Setter Methods	24
9	Specifying the Java Implementation Type in an Assembly	26
10	Java Packaging and Deployment Model	27
10.1	Contribution Metadata Extensions.....	27
10.2	Java Artifact Resolution.....	29
10.3	Class Loader Model.....	29
11	Conformance.....	30
11.1	SCA Java Component Implementation Composite Document	30
11.2	SCA Java Component Implementation Contribution Document	30
11.3	SCA Runtime	30
A.	XML Schemas	31
A.1	sca-contribution-java.xsd	31
A.2	sca-implementation-java.xsd.....	31

B. Conformance Items	33
C. Acknowledgements	35
D. Revision History	37

1 Introduction

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

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

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

1.1 Terminology

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

1.2 Normative References

- | | |
|-------------|---|
| [RFC2119] | S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , http://www.ietf.org/rfc/rfc2119.txt , IETF RFC 2119, March 1997. |
| [ASSEMBLY] | OASIS Committee Draft 06, <i>SCA Assembly Model Specification Version 1.1</i> , January 2010.
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd06.pdf |
| [POLICY] | OASIS Committee Draft 04, <i>SCA Policy Framework Specification Version 1.1</i> , September 2010.
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd04.pdf |
| [JAVACAA] | OASIS Committee Draft 04, <i>Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1</i> , February 2010.
http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd04.pdf |
| [WSDL] | WSDL Specification, WSDL 1.1: http://www.w3.org/TR/wsd |
| [OSGi Core] | OSGi Service Platform Core Specification, Version 4.0.1
http://www.osgi.org/download/r4v41/r4.core.pdf |
| [JAVABEANS] | JavaBeans 1.01 Specification,
http://java.sun.com/javase/technologies/desktop/javabeans/api/ |
| [JAX-WS] | JAX-WS 2.1 Specification (JSR-224),
http://www.jcp.org/en/jsr/detail?id=224 |
| [WSBINDING] | OASIS Committee Draft 04, <i>SCA Web Service Binding Specification Version 1.1</i> , May 2010.
http://docs.oasis-open.org/opencsa/sca-bindings/sca-wsbinding-1.1-spec-cd04.pdf |

2 Service

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

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

- A Java interface
- A Java class
- A Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType.

[JCI20001]

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

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

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

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

2.1 Use of @Service

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

Snippet 2-1 and **Error! Reference source not found.** are an example of a Java service interface and a Java implementation which provides a service using that interface:

Interface:

```
package services.hello;

public interface HelloService {

    String hello(String message);
}
```

Snippet 2-1: Example Java Service Interface

Implementation class:

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

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

Snippet 2-2: Example Java Component Implementation

83 The XML representation of the component type for this implementation is shown in Snippet 2-3 for
84 illustrative purposes. There is no need to author the component type as it is introspected from the Java
85 class.

86

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

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

96

97 Another possibility is to use the Java implementation class itself to define a service offered by a
98 component and the interface of the service. In this case, the `@Service` annotation can be used to
99 explicitly declare the implementation class defines the service offered by the implementation. In this case,
100 a component will only offer services declared by `@Service`. Snippet 2-4 illustrates this:

101

```
102 package services.hello;  
103  
104 @Service(HelloServiceImpl.class)  
105 public class HelloServiceImpl implements AnotherInterface {  
106  
107     public String hello(String message) {  
108         ...  
109     }  
110     ...  
111 }
```

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

113

114 In Snippet 2-4, `HelloServiceImpl` offers one service as defined by the public methods of the
115 implementation class. The interface `AnotherInterface` in this case does not specify a service offered by
116 the component. Snippet 2-5 is an XML representation of the introspected component type:

```
117 <?xml version="1.0" encoding="UTF-8"?>  
118 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">  
119     <service name="HelloServiceImpl">  
120         <interface.java interface="services.hello.HelloServiceImpl"/>  
121     </service>  
122 </componentType>
```

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

126

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

129

```
130 @Service(interfaces={HelloService.class, AnotherInterface.class})  
131 public class HelloServiceImpl implements HelloService, AnotherInterface  
132 {  
133  
134     public String hello(String message) {  
135         ...  
136     }  
137 }
```



```
137     ...
138 }
```

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

140

141 Snippet 2-7 shows the introspected component type for this implementation.

```
142 <?xml version="1.0" encoding="UTF-8"?>
143 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
144
145     <service name="HelloService">
146         <interface.java interface="services.hello.HelloService"/>
147     </service>
148     <service name="AnotherService">
149         <interface.java interface="services.hello.AnotherService"/>
150     </service>
151
152 </componentType>
```

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

154 2.2 Local and Remotable Services

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

159 Interface:

```
160 package services.hello;
161
162 @Remotable
163 public interface HelloService {
164
165     String hello(String message);
166 }
```

167 *Snippet 2-8: Example Remotable Interface*

168

169 Implementation class:

```
170 package services.hello;
171
172 @Service(HelloService.class)
173 public class HelloServiceImpl implements HelloService {
174
175     public String hello(String message) {
176         ...
177     }
178 }
```

179 *Snippet 2-9: Implementation for Remotable Interface*

180

181 Snippet 2-10 shows the introspected component type for this implementation.

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

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

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

191 If a service is defined by a Java implementation class instead of a Java interface, the @Remotable
192 annotation can be used on the implementation class to indicate that the service is remotable. Snippet
193 2-11 demonstrates this:

```
194 package services.hello;  
195  
196 @Remotable  
197 @Service(HelloServiceImpl.class)  
198 public class HelloServiceImpl {  
199  
200     public String hello(String message) {  
201         ...  
202     }  
203 }
```

204 *Snippet 2-11: Remotable Interface Defined by a Class*

205

206 Snippet 2-12 shows the introspected component type for this implementation.

```
207 <?xml version="1.0" encoding="UTF-8"?>  
208 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">  
209     <service name="HelloServiceImpl">  
210         <interface.java interface="services.hello.HelloServiceImpl"/>  
211     </service>  
212 </componentType>
```

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

214

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

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

220 Interface:

```
221 package services.hello;  
222  
223 public interface HelloService {  
224  
225     String hello(String message);  
226 }
```

227 *Snippet 2-13: Interface without @Remotable*

228

229 Implementation class:

```
230 package services.hello;  
231  
232 @Remotable  
233 @Service(HelloService.class)  
234 public class HelloServiceImpl implements HelloService {  
235  
236     public String hello(String message) {  
237         ...  
238     }  
239 }
```

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

241

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

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

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

252

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

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

260 2.3 Introspecting Services Offered by a Java Implementation

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

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

270 2.4 Non-Blocking Service Operations

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

274 2.5 Callback Services

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

279 3 References

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

283 3.1 Reference Injection

284 A Java implementation type can explicitly specify its references through the use of the @Reference
285 annotation as in Snippet 3-1:

```
286  
287 public class ClientComponentImpl implements Client {  
288     private HelloService service;  
289  
290     @Reference  
291     public void setHelloService(HelloService service) {  
292         this.service = service;  
293     }  
294 }
```

295 *Snippet 3-1: Specifying a Reference*

296

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

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

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

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

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

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

319 3.2 Dynamic Reference Access

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

323 4 Properties

324 4.1 Property Injection

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

328 A Java implementation type can explicitly specify its properties through the use of the @Property
329 annotation as in Snippet 4-1:

```
330  
331 public class ClientComponentImpl implements Client {  
332     private int maxRetries;  
333  
334     @Property  
335     public void setMaxRetries(int maxRetries) {  
336         this.maxRetries = maxRetries;  
337     }  
338 }
```

339 *Snippet 4-1: Specifying a Property*

340

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

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

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

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

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

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

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

366 4.2 Dynamic Property Access

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

370 5 Implementation Instance Creation

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

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

- 379 1. **A declared constructor annotated with a @Constructor annotation.**
- 380 2. **A declared constructor, all of whose parameters are annotated with either @Property or**
381 **@Reference.**
- 382 3. **A no-argument constructor.**

383 [JCI50004]

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

385 **In the absence of an @Constructor annotation, there MUST NOT be more than one constructor that has**
386 **a non-empty parameter list with all parameters annotated with either @Property or @Reference.**
387 [JCI50005]

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

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

393 Snippet 5-1 shows examples of legal Java component constructor declarations:

```
394 /** Constructor declared using @Constructor annotation */  
395 public class Impl1 {  
396     private String someProperty;  
397     @Constructor  
398     public Impl1( @Property("someProperty") String propval ) {...}  
399 }  
400  
401 /** Declared constructor unambiguously identifying all Property  
402  * and Reference values */  
403 public class Impl2 {  
404     private String someProperty;  
405     private SomeService someReference;  
406     public Impl2( @Property("someProperty") String a,  
407                  @Reference("someReference") SomeService b )  
408         {...}  
409 }  
410  
411 /** Declared constructor unambiguously identifying all Property  
412  * and Reference values plus an additional Property injected  
413  * via a setter method */  
414 public class Impl3 {  
415     private String someProperty;  
416     private String anotherProperty;  
417     private SomeService someReference;  
418     public Impl3( @Property("someProperty") String a,  
419                  @Reference("someReference") SomeService b)  
420         {...}  
421     @Property  
422     public void setAnotherProperty( String anotherProperty ) {...}
```

```
423     }
424
425     /** No-arg constructor */
426     public class Impl4 {
427         @Property
428         public String someProperty;
429         @Reference
430         public SomeService someReference;
431         public Impl4() {...}
432     }
433
434     /** Unannotated implementation with no-arg constructor */
435     public class Impl5 {
436         public String someProperty;
437         public SomeService someReference;
438         public Impl5() {...}
439     }
```

440 *Snippet 5-1: Examples of Valid Constructors*

441 6 Implementation Scopes and Lifecycle Callbacks

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

445 Implementations specify their scope through the use of the @Scope annotation as shown in Snippet 6-1:
446

```
447 @Scope("COMPOSITE")  
448 public class ClientComponentImpl implements Client {  
449     // ...  
450 }
```

451 *Snippet 6-1: Specifying the Scope of an Implementation*

452
453 When the @Scope annotation is not specified on an implementation class, its scope is defaulted to
454 STATELESS.

455 A Java component implementation specifies init and destroy methods by using the @Init and @Destroy
456 annotations respectively, as described in the SCA-J Common Annotations and APIs specification
457 [JAVACAA].

458 For example:

```
459 public class ClientComponentImpl implements Client {  
460  
461     @Init  
462     public void init() {  
463         //...  
464     }  
465  
466     @Destroy  
467     public void destroy() {  
468         //...  
469     }  
470 }
```

471 *Snippet 6-2: Example Init and Destroy Methods*

472 **7 Accessing a Callback Service**

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

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

479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522

8 Component Type of a Java Implementation

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

The component type of a Java Implementation is introspected from the implementation class using the rules:

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

- name attribute is the simple name of the interface or implementation class (i.e., without the package name)
- requires attribute is omitted unless the service implementation class is annotated with general or specific intent annotations - in this case, the requires attribute is present with a value equivalent to the intents declared by the service implementation class.
- policySets attribute is omitted unless the service implementation class is annotated with @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by the @PolicySets annotation.
- <interface.java> child element is present with the interface attribute set to the fully qualified name of the interface or implementation class identified by the @Service annotation. See the SCA-J Common Annotations and APIs specification [JAVACAA] for a definition of how policy annotations on Java interfaces, Java classes, and methods of Java interfaces are handled.
- remotable attribute of <interface.java> child element is omitted unless the service is defined by a Java interface with no @Remotable annotation and the service implementation class is annotated with @Remotable, in which case the <interface.java> element has remotable="true".
- binding child element is omitted
- callback child element is omitted

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

- name attribute has the value of the name parameter of the @Reference annotation, if present, otherwise it is the name of the field or the JavaBeans property name [JAVABEANS] corresponding to the setter method name, depending on what element of the class is annotated by the @Reference (note: for a constructor parameter, the @Reference annotation needs to have a name parameter)
- autowire attribute is omitted
- wiredByImpl attribute is omitted
- target attribute is omitted
- the multiplicity attribute is set according to the rules in section "@Reference" of the SCA Common Annotations and APIs Specification [JAVACAA]
- requires attribute is omitted unless the field, setter method or parameter is also annotated with general or specific intent annotations - in this case, the requires attribute is present with a value equivalent to the intents declared by the Java reference.
- policySets attribute is omitted unless the field, setter method or parameter is also annotated with @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by the @PolicySets annotation.
- <interface.java> child element with the interface attribute set to the fully qualified name of the interface class which types the field or setter method or constructor parameter. See the SCA-J Common Annotations and APIs specification [JAVACAA] for a definition of how policy annotations on Java interfaces and methods of Java interfaces are handled.

- 523 • remotable attribute of <interface.java> child element is omitted unless the interface class has no
- 524 • @Remotable annotation and there is a @Remotable annotation on the field, setter method or
- 525 • constructor parameter, in which case the <interface.java> element has remotable="true".
- 526 • binding child element is omitted
- 527 • callback child element is omitted
- 528 A <property/> element exists for each @Property annotation:
 - 529 • name attribute has the value of the name parameter of the @Property annotation, if present,
 - 530 • otherwise it is the name of the field or the JavaBeans property name [JAVABEANS] corresponding to
 - 531 • the setter method name, depending on what element of the class is annotated by the @Property
 - 532 • (note: for a constructor parameter, the @Property annotation needs to have a name parameter)
 - 533 • value attribute is omitted
 - 534 • type attribute which is set to the XML type implied by the JAXB mapping of the Java type of the field
 - 535 • or the Java type defined by the parameter of the setter method. Where the type of the field or of the
 - 536 • setter method is an array, the element type of the array is used. Where the type of the field or of the
 - 537 • setter method is a java.util.Collection, the parameterized type of the Collection or its member type is
 - 538 • used. If the JAXB mapping is to a global element rather than a type (JAXB @XMLRootElement
 - 539 • annotation), the type attribute is omitted. Note that JAXB mapping is the default mapping, but that
 - 540 • other mappings are possible, where supported by the SCA runtime
 - 541 • (for example, SDO). How such alternative mappings are indicated is not described in this
 - 542 • specification.
 - 543 • element attribute is omitted unless the JAXB mapping of the Java type of the field or the Java type
 - 544 • defined by the parameter of the setter method is to a global element (JAXB @XMLRootElement
 - 545 • annotation). In this case, the element attribute has the value of the name of the XSD global element
 - 546 • implied by the JAXB mapping.
 - 547 • many attribute is set according to the rules in section “@Property” of the SCA Common Annotations
 - 548 • and APIs Specification [JAVACAA].
 - 549 • mustSupply attribute is set to "true" unless the @Property annotation has required=false, in which
 - 550 • case it is set to "false"
- 551 An <implementation.java/> element exists if the service implementation class is annotated with general or
- 552 • specific intent annotations or with @PolicySets:
 - 553 • requires attribute is omitted unless the service implementation class is annotated with general or
 - 554 • specific intent annotations - in this case, the requires attribute is present with a value equivalent to the
 - 555 • intents declared by the service implementation class.
 - 556 • policySets attribute is omitted unless the service implementation class is annotated with @PolicySets
 - 557 • - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by
 - 558 • the @PolicySets annotation.

559 **8.1 Component Type of an Implementation with no @Service,**

560 **@Reference or @Property Annotations**

561 The section defines the rules for determining the services of a Java component implementation that

562 contains no @Service annotations, no @Reference annotations, and no @Property annotations. If the

563 implementation class contains any @Service, @Reference or @Property annotations, the rules in this

564 section do not apply.

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

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

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

572 • name attribute is the simple name of the interface or the simple name of the class

573 • requires attribute is omitted unless the service implementation class is annotated with general or

574 specific intent annotations - in this case, the requires attribute is present with a value equivalent to the

575 intents declared by the service implementation class.

576 • policySets attribute is omitted unless the service implementation class is annotated with @PolicySets

577 - in this case, the policySets attribute is present with a value equivalent to the policy sets declared by

578 the @PolicySets annotation.

579 • <interface.java> child element is present with the interface attribute set to the fully qualified name of

580 the interface class or to the fully qualified name of the class itself. See the SCA-J Common

581 Annotations and APIs specification [JAVACAA] for a definition of how policy annotations on Java

582 interfaces, Java classes, and methods of Java interfaces are handled.

583 • remotable attribute of <interface.java> child element is omitted

584 • binding child element is omitted

585 • callback child element is omitted

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

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

588 1. Public setter methods that are not part of the implementation of an SCA service (either explicitly

589 marked with @Service or implicitly defined as described above)

590 2. Public or protected fields unless there is a public setter method for the same name

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

592 • its type is an interface annotated with @Remotable

593 • its type is an array where the element type of the array is an interface annotated with @Remotable

594 • its type is a java.util.Collection where the parameterized type of the Collection or its member type is

595 an interface annotated with @Remotable

596 The reference in the component type has:

597 • name attribute with the value of the name of the field or the JavaBeans property name [JAVABEANS]

598 corresponding to the setter method name

599 • multiplicity attribute is (1..1) for the case where the type is an interface

600 multiplicity attribute is (1..n) for the cases where the type is an array or is a java.util.Collection

601 • <interface.java> child element with the interface attribute set to the fully qualified name of the

602 interface class which types the field or setter method. See the SCA-J Common Annotations and APIs

603 specification [JAVACAA] for a definition of how policy annotations on Java interfaces and methods of

604 Java interfaces are handled.

605 • remotable attribute of <interface.java> child element is omitted

606 • requires attribute is omitted unless the field or setter method is also annotated with general or

607 specific intent annotations - in this case, the requires attribute is present with a value equivalent

608 to the intents declared by the Java reference.

609 • policySets attribute is omitted unless the field or setter method is also annotated with

610 @PolicySets - in this case, the policySets attribute is present with a value equivalent to the policy

611 sets declared by the @PolicySets annotation.

612 • all other attributes and child elements of the reference are omitted

613 An unannotated field or setter method is a **property** if it is not a reference using the immediately

614 preceding rules.

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

616 • name attribute with the value of the name of the field or the JavaBeans property name [JAVABEANS]

617 corresponding to the setter method name

- 618 • type attribute and element attribute are set as described for a property declared via a `@Property`
619 annotation, following the JAXB mapping of the Java type of the field or setter method by default. Note
620 that other mappings are possible, where supported by the SCA runtime (for example, SDO). How
621 such alternative mappings are indicated is not described in this specification.
- 622 • value attribute omitted
- 623 • many attribute set to “false” unless the type of the field or of the setter method is an array or a
624 `java.util.Collection`, in which case it is set to “true”.
- 625 • `mustSupply` attribute set to true

626 8.2 Impact of JAX-WS Annotations on ComponentType

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

631 8.2.1 @WebService

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

- 634 • The value of the name property of the `@WebService` annotation is used as the name of the
635 `<service/>` element
- 636 • If the `endpointInterface` property of the `@WebService` annotation has a non-default value, then the
637 interface attribute of the `<interface.java/>` child element of the `<service/>` element is set to the
638 interface identified by the `endpointInterface` property.
- 639 • The `<interface.java/>` child element of the `<service/>` has the `remotable` attribute set to “true”.
- 640 • If the `wsdlLocation` property of the `@WebService` annotation has a non-default value, then the
641 `<service/>` element has an `<interface.wsdl/>` child element instead of an `<interface.java/>` child
642 element. The value of the `@interface` attribute of the `<interface.wsdl/>` element is constructed by
643 pointing to the `portType`, in the WSDL definition pointed to by `@wsdlLocation`, which resulted from the
644 JAX-WS mapping for the annotated class or interface.
- 645 • If both the `endpointInterface` and `wsdlLocation` properties of the `@WebService` annotation have
646 default values and there is no `@Service` annotation, then the interface attribute of the
647 `<interface.java/>` child element of the `<service/>` element is set to the fully qualified name of the
648 interface or implementation class.

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

651 8.2.2 @WebMethod

- 652 • The value of the name property of the `@WebMethod` annotation is used when testing interface
653 compatibility.
- 654 • If the value of the `exclude` property of the `@WebMethod` annotation is “true”, then the method is
655 excluded from the SCA interface.

656 8.2.3 @WebParam

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

661 8.2.4 @WebResult

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

664 8.2.5 @SOAPBinding

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

668 8.2.6 @WebServiceProvider

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

- 671 • Where the Java implementation class implements a Java interface that is annotated with
672 @Remotable:
 - 673 ○ The @name attribute of the <service/> element in the component type is the simple name of
674 the Java interface class where the Java implementation class implements the Java interface
675 marked with @Remotable.
 - 676 ○ The <service/> element has a <interface.java/> subelement with an @interface set to the fully
677 qualified name of the Java interface class.
- 678 • Where the Java implementation class does not implement a Java interface that is annotated with
679 @Remotable:
 - 680 ○ The @name attribute of the <service/> element in the component type is the simple name of
681 the Java implementation class.
 - 682 ○ The <service/> element has a <interface.java/> subelement with an @interface set to the fully
683 qualified name of the Java implementation class and the @remotable attribute is set to "true".
- 684 • If the wsdlLocation property of the @WebServiceProvider annotation has a non-default value, then
685 the <service/> element has an <interface.wsdl/> child element instead of an <interface.java/> child
686 element. The value of the @interface attribute of the <interface.wsdl/> element is constructed by
687 pointing to the portType, in the WSDL definition pointed to by @wsdlLocation, which resulted from the
688 JAX-WS mapping for the annotated class or interface.

689 8.2.7 Web Service Binding

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

693 Therefore, the presence of **any** JAX-WS annotations in an SCA implementation or in an interface class
694 requires that any SCA services exposed by an implementation class are made available using the SOAP
695 1.1 HTTP WSDL binding by default. As a result, the respective <service/> elements in the component
696 type of the implementation class each have a <binding.ws/> subelement [WSBINDING] with the
697 SOAP.v1_1 intent added to the requires attribute of the <binding.ws/> subelement.

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

700 8.2.7.1 @BindingType

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

706

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

707 Table 8-1: Intents for WSDL Bindings

708

8.3 Component Type Introspection Examples

709 Snippet 8-1 shows how intent annotations can be applied to service and reference interfaces and
710 methods as well as to a service implementation class.

```

711 // Service interface
712 package test;
713 import org.oasisopen.sca.annotation.Authentication;
714 import org.oasisopen.sca.annotation.Confidentiality;
715
716 @Authentication
717 public interface MyService {
718     @Confidentiality
719     void mymethod();
720 }
721
722 // Reference interface
723 package test;
724 import org.oasisopen.sca.annotation.Integrity;
725
726 public interface MyRefInt {
727     @Integrity
728     void mymethod1();
729 }
730
731 // Service implementation class
732 package test;
733 import static org.oasisopen.sca.Constants.SCA_PREFIX;
734 import org.oasisopen.sca.annotation.Confidentiality;
735 import org.oasisopen.sca.annotation.Reference;
736 import org.oasisopen.sca.annotation.Service;
737 @Service(MyService.class)
738 @Requires(SCA_PREFIX+"managedTransaction")
739 public class MyServiceImpl {
740     @Confidentiality
741     @Reference
742     protected MyRefInt myRef;
743
744     public void mymethod() {...}
745 }

```

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

747

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

```

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

```



```

752 <implementation.java class="test.MyServiceImpl"
753     requires="sca:managedTransaction"/>
754 <service name="MyService" requires="sca:managedTransaction">
755     <interface.java interface="test.MyService"/>
756 </service>
757 <reference name="myRef" requires="sca:confidentiality">
758     <interface.java interface="test.MyRefInt"/>
759 </reference>
760 </componentType>

```

761 *Snippet 8-2: Introspected Component Type with Intents*

762 8.4 Java Implementation with Conflicting Setter Methods

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

768 Snippet 8-3 shows examples of illegal Java implementation due to the presence of more than one setter
769 method resulting in either an SCA property or an SCA reference with the same name:

770

```

771 /** Illegal since two setter methods with same JavaBeans property name
772  * are annotated with @Property annotation. */
773 public class IllegalImpl1 {
774     // Setter method with upper case initial letter 'S'
775     @Property
776     public void setSomeProperty(String someProperty) {...}
777
778     // Setter method with lower case initial letter 's'
779     @Property
780     public void setsomeProperty(String someProperty) {...}
781 }
782
783 /** Illegal since setter methods with same JavaBeans property name
784  * are annotated with @Reference annotation. */
785 public class IllegalImpl2 {
786     // Setter method with upper case initial letter 'S'
787     @Reference
788     public void setSomeReference(SomeService service) {...}
789
790     // Setter method with lower case initial letter 's'
791     @Reference
792     public void setsomeReference(SomeService service) {...}
793 }
794
795 /** Illegal since two setter methods with same JavaBeans property name
796  * are resulting in an SCA property. Implementation has no @Property
797  * or @Reference annotations. */
798 public class IllegalImpl3 {
799     // Setter method with upper case initial letter 'S'
800     public void setSomeOtherProperty(String someProperty) {...}
801
802     // Setter method with lower case initial letter 's'
803     public void setsomeOtherProperty(String someProperty) {...}
804 }
805
806 /** Illegal since two setter methods with same JavaBeans property name
807  * are resulting in an SCA reference. Implementation has no @Property
808  * or @Reference annotations. */
809 public class IllegalImpl4 {
810     // Setter method with upper case initial letter 'S'

```



```
811     public void setSomeOtherReference(SomeService service) {...}
812
813     // Setter method with lower case initial letter 's'
814     public void setsomeOtherReference(SomeService service) {...}
815 }
```

816 *Snippet 8-3: Example Conflicting Setter Methods*

817

818 Snippet 8-4 is an example of a legal Java implementation in spite of the implementation class having two
819 setter methods with same JavaBeans property name [JAVABEANS] corresponding to the setter method
820 name:

821

```
822 /** Two setter methods with same JavaBeans property name, but one is
823  * annotated with @Property and the other is annotated with @Reference
824  * annotation. */
825 public class WeirdButLegalImpl {
826     // Setter method with upper case initial letter 'F'
827     @Property
828     public void setFoo(String foo) {...}
829
830     // Setter method with lower case initial letter 'f'
831     @Reference
832     public void setfoo(SomeService service) {...}
833 }
```

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

835 9 Specifying the Java Implementation Type in an 836 Assembly

837 Snippet 9-1 shows the pseudo-schema that defines the implementation element schema used for the
838 Java implementation type:

839

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

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

844

845 The implementation.java element has the attributes:

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

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

853

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

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

859 10 Java Packaging and Deployment Model

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

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

868 10.1 Contribution Metadata Extensions

869 SCA contributions can be self contained such that all the code and metadata needed to execute the
870 components defined by the contribution is contained within the contribution. However, in larger projects,
871 there is often a need to share artifacts across contributions. This is accomplished through the use of the
872 import and export extension points as defined in the sca-contribution.xml document. An SCA contribution
873 that needs to use a Java class from another contribution can declare the dependency via an
874 <import.java/> extension element, contained within a <contribution/> element, as shown in Snippet 10-1:

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

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

877

878 The import.java element has the attributes:

- 879 • **package : string (1..1)** – The name of one or more Java package(s) to use from another
880 contribution. Where there is more than one package, the package names are separated by a comma
881 ",".

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

884 package="com.acme.package1;version=1.4.1"

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

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

887 [1.2,1.3] - enclosing square brackets - inclusive range meaning any version in the range from the
888 lowest to the highest, including the lowest and the highest

889 (1.3.1,2.4.1) - enclosing round brackets - exclusive range meaning any version in the range from the
890 lowest to the highest but not including the lowest or the highest.

891 1.4.1 - no enclosing brackets - implies any version at or later than the specified version number is
892 acceptable - equivalent to [1.4.1, infinity)

893 If no version is specified for an imported package, then it is assumed to have a version range of
894 [0.0.0, infinity) - ie any version is acceptable.

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

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

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

903 An SCA contribution that wants to allow a Java package to be used by another contribution can declare
904 the exposure via an <export.java/> extension element as shown in Snippet 10-2:

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

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

907

908 The export.java element has the attributes:

909 • **package : string (1..1)** – The name of one or more Java package(s) to expose for sharing by another
910 contribution. Where there is more than one package, the package names are separated by a comma
911 ",".

912 The package can have a **version number** appended to it, separated from the package name by a
913 semicolon ";" followed by the text "version=" and the version number:

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

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

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

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

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

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

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

930 For example, a contribution that wants to:

931 use classes from the *some.package* package from another contribution (any version)

932 use classes of the *some.other.package* package from another contribution, at exactly version 2.0.0

933 expose the *my.package* package from its own contribution, with version set to 1.0.0

934 would specify an sca-contribution.xml file shown in Snippet 10-3 :

935

```
936 <?xml version="1.0" encoding="UTF-8"?>  
937 <contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200912>  
938 ...  
939 <import.java package="some.package"/>  
940 <import.java package="some.other.package;version=[2.0.0]"/>  
941 <export.java package="my.package;version=1.0.0"/>  
942 </contribution>
```

943 *Snippet 10-3: Example Imports and Exports*

944

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

947

948 10.2 Java Artifact Resolution

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

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

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

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

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

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

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

964 [JCI100008]

965 10.3 Class Loader Model

966 The SCA runtime MUST ensure that the Java classes used by a contribution are all loaded by a class
967 loader that is unique for each contribution in the Domain. [JCI100010] The SCA runtime MUST ensure
968 that Java classes that are imported into a contribution are loaded by the exporting contribution's class
969 loader [JCI100011], as described in the section "Contribution Metadata Extensions"

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

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

973 *Snippet 10-4: Example Class Loader Use*

974 evaluates to true.

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

977 **11 Conformance**

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

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

984 **11.1 SCA Java Component Implementation Composite Document**

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

990 **11.2 SCA Java Component Implementation Contribution Document**

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

996 **11.3 SCA Runtime**

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

- 998 1. The implementation MUST meet all the conformance requirements defined by the SCA Assembly
999 Model Specification [ASSEMBLY].
- 1000 2. The implementation MUST reject an SCA Java Composite Document that does not conform to the
1001 sca-implementation-java.xsd schema.
- 1002 3. The implementation MUST reject an SCA Java Contribution Document that does not conform to the
1003 sca-contribution-java.xsd schema.
- 1004 4. The implementation MUST meet all the conformance requirements, specified in 'Section 11
1005 Conformance', from the SCA-J Common Annotations and APIs Specification [JAVACAA].
- 1006 5. This specification permits an implementation class to use any and all the APIs and annotations
1007 defined in the SCA-J Common Annotations and APIs Specification [JAVACAA], therefore the
1008 implementation MUST comply with all the statements in Appendix B: Conformance Items of
1009 [JAVACAA], notably all mandatory statements have to be implemented.
- 1010 6. The implementation MUST comply with all statements related to an SCA Runtime, specified in
1011 'Appendix B: Conformance Items' of this specification, notably all mandatory statements have to
1012 be implemented.

1013 A. XML Schemas

1014 A.1 sca-contribution-java.xsd

```
1015 <?xml version="1.0" encoding="UTF-8"?>
1016 <!-- Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
1017 OASIS trademark, IPR and other policies apply. -->
1018 <schema xmlns="http://www.w3.org/2001/XMLSchema"
1019 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
1020 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
1021 elementFormDefault="qualified">
1022
1023 <include schemaLocation="sca-contribution-1.1-cd06.xsd"/>
1024
1025 <!-- Import.java -->
1026 <element name="import.java" type="sca:JavaImportType"
1027 substitutionGroup="sca:importBase" />
1028 <complexType name="JavaImportType">
1029 <complexContent>
1030 <extension base="sca:Import">
1031 <attribute name="package" type="string" use="required"/>
1032 <attribute name="location" type="anyURI" use="optional"/>
1033 </extension>
1034 </complexContent>
1035 </complexType>
1036
1037 <!-- Export.java -->
1038 <element name="export.java" type="sca:JavaExportType"
1039 substitutionGroup="sca:exportBase" />
1040 <complexType name="JavaExportType">
1041 <complexContent>
1042 <extension base="sca:Export">
1043 <attribute name="package" type="string" use="required"/>
1044 </extension>
1045 </complexContent>
1046 </complexType>
1047
1048 </schema>
```

1049 A.2 sca-implementation-java.xsd

```
1050 <?xml version="1.0" encoding="UTF-8"?>
1051 <!-- Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
1052 OASIS trademark, IPR and other policies apply. -->
1053 <schema xmlns="http://www.w3.org/2001/XMLSchema"
1054 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
1055 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
1056 elementFormDefault="qualified">
1057
1058 <include schemaLocation="sca-core-1.1-cd06.xsd"/>
1059
1060 <!-- Java Implementation -->
1061 <element name="implementation.java" type="sca:JavaImplementation"
1062 substitutionGroup="sca:implementation"/>
1063 <complexType name="JavaImplementation">
```

```
1064     <complexContent>
1065         <extension base="sca:Implementation">
1066             <sequence>
1067                 <any namespace="##other" processContents="lax"
1068                     minOccurs="0" maxOccurs="unbounded"/>
1069             </sequence>
1070             <attribute name="class" type="NCName" use="required"/>
1071         </extension>
1072     </complexContent>
1073 </complexType>
1074
1075 </schema>
```


1076

B. Conformance Items

1077 This section contains a list of conformance items for the SCA Java Component Implementation
1078 specification.

1079

Conformance ID	Description
[JCI20001]	<p>The services provided by a Java-based implementation MUST have an interface defined in one of the following ways:</p> <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]	<p>Java implementation classes MUST implement all the operations defined by the service interface.</p>
[JCI40001]	<p>For an unannotated field or setter method that is introspected as a property and where the Java type of the field or setter method is a JAXB [JAXB] annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition into an instance of the property's Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled.</p>
[JCI50001]	<p>A Java implementation class MUST provide a public or protected constructor that can be used by the SCA runtime to create the implementation instance.</p>
[JCI50002]	<p>The @Constructor annotation MUST NOT appear on more than one constructor.</p>
[JCI50004]	<p>The constructor to use for the creation of an implementation instance MUST be selected by the SCA runtime using the sequence:</p> <ol style="list-style-type: none"> 7. A declared constructor annotated with a @Constructor annotation. 8. A declared constructor, all of whose parameters are annotated with either @Property or @Reference. 9. A no-argument constructor.
[JCI50005]	<p>In the absence of an @Constructor annotation, there MUST NOT be more than one constructor that has a non-empty parameter list with all parameters annotated with either @Property or @Reference.</p>
[JCI60001]	<p>The SCA runtime MUST support the STATELESS and COMPOSITE implementation scopes.</p>
[JCI80001]	<p>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".</p>
[JCI80002]	<p>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.</p>

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

1081

C. Acknowledgements

1082 The following individuals have participated in the creation of this specification and are gratefully
1083 acknowledged:

1084 **Participants:**

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

Michael Rowley
Vladimir Savchenko
Pradeep Simha
Raghav Srinivasan
Scott Vorthmann
Feng Wang

Paul Yang

Active Endpoints, Inc.
SAP AG*
TIBCO Software Inc.
Oracle Corporation
TIBCO Software Inc.
Primeton Technologies, Inc.
Changfeng Open Standards
Platform Software

1085

1086

D. Revision History

1087 [optional; should not be included in OASIS Standards]

1088

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
wd09	2009-04-29	David Booz	Editorial fixes throughout (capitalization, quotes, fonts, spec references, etc.)
wd10	2009-04-30	David Booz	Editorial fixes, indentation, etc.
cd01	2009-05-04	David Booz	Final editorial fixes for CD and PRD
cd01-rev1	2009-08-12	David Booz	Editorial fixes, applied issues: 143,153,176
cd01-rev2	2009-09-14	David Booz	Applied issues: 157,162
cd01-rev3	2010-01-18	David Booz	Upgraded namespace to latest 200912 Applied issues: 168, 171, 181, 184, 186, 192,193
cd01-rev4	2010-01-20	Bryan Aupperle	Editorial updates to match OASIS document standards

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

1089