



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Working Draft 05

03 October 2008

Specification URIs:

This Version:

<http://docs.oasis-open.org/sca-j/sca-javacaa-1.1-spec-WD05.html>
<http://docs.oasis-open.org/sca-j/sca-javacaa-1.1-spec-WD05.doc>
<http://docs.oasis-open.org/sca-j/sca-javacaa-1.1-spec-WD05.pdf>

Previous Version:

Latest Version:

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

Latest Approved Version:

Technical Committee:

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

Chair(s):

Simon Nash,	IBM
Michael Rowley,	BEA Systems
Mark Combellack,	Avaya

Editor(s):

Ron Barack,	SAP
David Booz,	IBM
Mark Combellack,	Avaya
Mike Edwards,	IBM
Anish Karmarkar,	Oracle
Ashok Malhotra,	Oracle
Peter Peshev,	SAP

Related work:

This specification replaces or supersedes:

- Service Component Architecture Java Annotations and APIs Specification Version 1.00, March 21 2007

This specification is related to:

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

Declared XML Namespace(s):

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

Abstract:

The SCA Java Common Annotation and APIs specify a Java syntax for programming concepts defined in the SCA Assembly Model Specification. It specifies a set of APIs and annotations that may be used by Java-based SCA specifications.

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous and conversational services
4. Metadata for callbacks
5. Definitions of standard component implementation scopes
6. Java to WSDL and WSDL to Java mappings
7. Security policy annotations

Note that individual programming models may chose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

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, 2008. 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.....	7
1.1	Terminology.....	7
1.2	Normative References.....	7
1.3	Non-Normative References.....	8
2	Implementation Metadata.....	9
2.1	Service Metadata.....	9
2.1.1	@Service.....	9
2.1.2	Java Semantics of a Remotable Service.....	9
2.1.3	Java Semantics of a Local Service.....	9
2.1.4	@Reference.....	10
2.1.5	@Property.....	10
2.2	Implementation Scopes: @Scope, @Init, @Destroy.....	10
2.2.1	Stateless scope.....	11
2.2.2	Request scope.....	11
2.2.3	Composite scope.....	11
2.2.4	Conversation scope.....	11
3	Interface.....	12
3.1	Java interface element ("interface.java").....	12
3.2	@Remotable.....	12
3.3	@Conversational.....	12
4	Client API.....	13
4.1	Accessing Services from an SCA Component.....	13
4.1.1	Using the Component Context API.....	13
4.2	Accessing Services from non-SCA component implementations.....	13
4.2.1	ComponentContext.....	13
5	Error Handling.....	14
6	Asynchronous and Conversational Programming.....	15
6.1	@OneWay.....	15
6.2	Conversational Services.....	15
6.2.1	ConversationAttributes.....	15
6.2.2	@EndsConversation.....	16
6.3	Passing Conversational Services as Parameters.....	16
6.4	Conversational Client.....	16
6.5	Conversation Lifetime Summary.....	17
6.6	Conversation ID.....	18
6.6.1	Application Specified Conversation IDs.....	18
6.6.2	Accessing Conversation IDs from Clients.....	18
6.7	Callbacks.....	18
6.7.1	Stateful Callbacks.....	18
6.7.2	Stateless Callbacks.....	20
6.7.3	Implementing Multiple Bidirectional Interfaces.....	21
6.7.4	Accessing Callbacks.....	21
6.7.5	Customizing the Callback.....	22

6.7.6 Customizing the Callback Identity	22
6.7.7 Bindings for Conversations and Callbacks.....	23
7 Java API	24
7.1 Component Context.....	24
7.2 Request Context	25
7.3 CallableReference	26
7.4 ServiceReference	26
7.5 Conversation.....	27
7.6 ServiceRuntimeException.....	27
7.7 NoRegisteredCallbackException	28
7.8 ServiceUnavailableException	28
7.9 InvalidServiceException.....	28
7.10 ConversationEndedException	28
8 Java Annotations	30
8.1 @AllowsPassByReference.....	30
8.2 @Callback	30
8.3 @ComponentName	32
8.4 @Constructor.....	32
8.5 @Context.....	33
8.6 @Conversational	34
8.7 @ConversationAttributes.....	34
8.8 @ConversationID	35
8.9 @Destroy	36
8.10 @EagerInit.....	36
8.11 @EndsConversation.....	37
8.12 @Init.....	37
8.13 @OneWay	38
8.14 @Property.....	39
8.15 @Reference.....	40
8.15.1 Reinjection.....	42
8.16 @Remotable.....	44
8.17 @Scope	45
8.18 @Service	46
9 WSDL to Java and Java to WSDL	48
9.1 JAX-WS Client Asynchronous API for a Synchronous Service.....	48
10 Policy Annotations for Java	50
10.1 General Intent Annotations.....	50
10.2 Specific Intent Annotations	52
10.2.1 How to Create Specific Intent Annotations.....	53
10.3 Application of Intent Annotations	54
10.3.1 Inheritance And Annotation	55
10.4 Relationship of Declarative And Annotated Intents	56
10.5 Policy Set Annotations.....	57
10.6 Security Policy Annotations	57
10.6.1 Security Interaction Policy	57

10.6.2 Security Implementation Policy	60
A. Acknowledgements	64
B. Non-Normative Text	65
C. Revision History.....	66

1 Introduction

The SCA Common Annotation, APIs, Client and Implementation Model specifies a Java syntax for programming concepts defined in the SCA Assembly Model Specification [1]. It specifies a set of APIs and annotations that may be used by Java-based SCA specifications.

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous and conversational services
4. Metadata for callbacks
5. Definitions of standard component implementation scopes
6. Java to WSDL and WSDL to Java mappings
7. Security policy annotations

Note that individual programming models may chose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

The goal of specifying the annotations, APIs, client and implementation model in this specification is to promote consistency and reduce duplication across various Java-related component implementation type specifications. The annotations, APIs, client and implementation model defined in this specification are designed to be used by other SCA Java-related specifications in either a partial or complete fashion.

This document defines implementation metadata using the annotation capability from Java™ 2 Standard Edition (J2SE) 5. However, SCA also allows service clients and implementations to be written using J2SE 1.4. All metadata that is represented by annotations can also be expressed using a component type side file, as defined in the SCA Assembly Specification [1].

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.

TBD TBD

[1] SCA Assembly Specification

<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf>

[2] SDO 2.1 Specification

<http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf>

[3] JAXB Specification

<http://www.jcp.org/en/jsr/detail?id=31>

- 41 [4] WSDL Specification
42 WSDL 1.1: <http://www.w3.org/TR/wsdl>
43 WSDL 2.0: <http://www.w3.org/TR/wsdl20/>
44 [5] SCA Policy Framework
45 http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf
46 [6] Common Annotation for Java Platform specification (JSR-250)
47 <http://www.jcp.org/en/jsr/detail?id=250>
48 [7] JAX-WS Specification (JSR-224)
49 <http://www.jcp.org/en/jsr/detail?id=224>
50

51 **1.3 Non-Normative References**

52 **TBD** **TBD**

53 2 Implementation Metadata

54 This section describes SCA Java-based metadata, which applies to Java-based implementation
55 types.

56 2.1 Service Metadata

57 2.1.1 @Service

58
59 The **@Service annotation** is used on a Java class to specify the interfaces of the services
60 implemented by the implementation. Service interfaces are defined in one of the following ways:

- 61 • As a Java interface
- 62 • As a Java class
- 63 • As a Java interface generated from a Web Services Description Language [4] (WSDL)
64 portType (Java interfaces generated from a WSDL portType are always **remotable**)

65 2.1.2 Java Semantics of a Remotable Service

66 A **remotable service** is defined using the @Remotable annotation on the Java interface that
67 defines the service. Remotable services are intended to be used for **coarse grained** services, and
68 the parameters are passed **by-value**. Remotable Services are not allowed to make use of method
69 **overloading**.

70 The following snippet shows an example of a Java interface for a remote service:

```
71 package services.hello;  
72 @Remotable  
73 public interface HelloService {  
74     String hello(String message);  
75 }  
76
```

77 2.1.3 Java Semantics of a Local Service

78 A **local service** can only be called by clients that are deployed within the same address space as
79 the component implementing the local service.

80 A local interface is defined by a Java interface with no @Remotable annotation or it is defined by a
81 Java class.

82 The following snippet shows an example of a Java interface for a local service:

```
83  
84 package services.hello;  
85 public interface HelloService {  
86     String hello(String message);  
87 }  
88
```

89 The style of local interfaces is typically **fine grained** and is intended for **tightly coupled**
90 interactions.

91 The data exchange semantic for calls to local services is **by-reference**. This means that code must
92 be written with the knowledge that changes made to parameters (other than simple types) by
93 either the client or the provider of the service are visible to the other.

94 2.1.4 @Reference

95 Accessing a service using reference injection is done by defining a field, a setter method
96 parameter, or a constructor parameter typed by the service interface and annotated with an
97 **@Reference** annotation.

98 2.1.5 @Property

99 Implementations can be configured with data values through the use of properties, as defined in
100 the SCA Assembly specification [1]. The **@Property** annotation is used to define an SCA property .

101 2.2 Implementation Scopes: @Scope, @Init, @Destroy

102 Component implementations can either manage their own state or allow the SCA runtime to do so.
103 In the latter case, SCA defines the concept of **implementation scope**, which specifies a visibility
104 and lifecycle contract an implementation has with the SCA runtime. Invocations on a service
105 offered by a component will be dispatched by the SCA runtime to an **implementation instance**
106 according to the semantics of its implementation scope.

107 Scopes are specified using the **@Scope** annotation on the implementation class.

108 This document defines four scopes:

- 109 • STATELESS
- 110 • REQUEST
- 111 • CONVERSATION
- 112 • COMPOSITE

113 Java-based implementation types can choose to support any of these scopes, and they may define
114 new scopes specific to their type.

115 An implementation type may allow component implementations to declare **lifecycle methods** that
116 are called when an implementation is instantiated or the scope is expired.

117 **@Init** denotes a method called upon first use of an instance during the lifetime of the scope
118 (except for composite scoped implementation marked to eagerly initialize, see [section Composite](#)
119 [Scope](#)).

120 **@Destroy** specifies a method called when the scope ends.

121 Note that only no-argument methods may be annotated as lifecycle methods.

122 The following snippet is an example showing a fragment of a service implementation annotated
123 with lifecycle methods:

```
124  
125     @Init  
126     public void start() {  
127         ...  
128     }  
129  
130     @Destroy  
131     public void stop() {  
132         ...  
133     }
```

134

135 The following sections specify four standard scopes, which a Java-based implementation type may
136 support.

137 2.2.1 Stateless scope

138 For stateless scope components, there is no implied correlation between implementation
139 instances used to dispatch service requests.

140 2.2.2 Request scope

141 The lifecycle of request scope extends from the point a request on a remotable interface enters
142 the SCA runtime and a thread processes that request until the thread completes synchronously
143 processing the request. During that time, all service requests are delegated to the same
144 implementation instance of a request-scoped component.

145 There are times when a local request scoped service is called without there being a remotable
146 service earlier in the call stack, such as when a local service is called from a non-SCA entity. In
147 these cases, a remote request is always considered to be present, but the lifetime of the request is
148 implementation dependent. For example, a timer event could be treated as a remote request.

149 2.2.3 Composite scope

150 All service requests are dispatched to the same implementation instance for the lifetime of the
151 containing composite. The lifetime of the containing composite is defined as the time it becomes
152 active in the runtime to the time it is deactivated, either normally or abnormally.

153 A composite scoped implementation may also specify eager initialization using the **@EagerInit**
154 annotation. When marked for eager initialization, the composite scoped instance is created when
155 its containing component is started. If a method is marked with the **@Init** annotation, it is called
156 when the instance is created.

157 2.2.4 Conversation scope

158 A **conversation** is defined as a series of correlated interactions between a client and a target
159 service. A conversational scope starts when the first service request is dispatched to an
160 implementation instance offering a conversational service. A conversational scope completes after
161 an end operation defined by the service contract is called and completes processing or the
162 conversation expires. A conversation may be long-running (for example, hours, days or weeks)
163 and the SCA runtime may choose to passivate implementation instances. If this occurs, the
164 runtime must guarantee that implementation instance state is preserved.

165 Note that in the case where a conversational service is implemented by a Java class marked as
166 conversation scoped, the SCA runtime will transparently handle implementation state. It is also
167 possible for an implementation to manage its own state. For example, a Java class having a
168 stateless (or other) scope could implement a conversational service.

169 A conversational scoped class **MUST NOT** expose a service using a non-conversational interface.
170 When a service has a conversational interface it **MUST** be implemented by a conversation-scoped
171 component. If no scope is specified on the implementation, then conversation scope is implied.

172 3 Interface

173 This section describes the SCA Java interface element and the SCA metadata for Java interfaces.

174 3.1 Java interface element ("interface.java")

175 The following snippet shows the schema for the Java interface element.

176

```
177 <interface.java interface="NCName" ... />
```

178

179 The interface.java element has the following attributes:

- 180 • **interface** – the fully qualified name of the Java interface

181

182 The following snippet shows an example of the Java interface element:

183

```
184 <interface.java interface="services.stockquote.StockQuoteService" />
```

185

186 Here, the Java interface is defined in the Java class file
187 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the
188 contribution in which the interface exists.

189 For the Java interface type system, **arguments and return values** of the service methods are
190 described using Java classes or simple Java types. [Service Data Objects \[2\]](#) are the preferred form
191 of Java class because of their integration with XML technologies.

192

193 3.2 @Remotable

194 The **@Remotable** annotation on a Java interface indicates that the interface is designed to be
195 used for remote communication. Remotable interfaces are intended to be used for **coarse**
196 **grained** services. Operations' parameters and return values are passed **by-value**. Remotable
197 Services are not allowed to make use of method **overloading**.

198 3.3 @Conversational

199 Java service interfaces may be annotated to specify whether their contract is conversational as
200 described in [the Assembly Specification \[1\]](#) by using the **@Conversational** annotation. A
201 conversational service indicates that requests to the service are correlated in some way.

202 When @Conversational is not specified on a service interface, the service contract is **stateless**.

203 4 Client API

204 This section describes how SCA services may be programmatically accessed from components and
205 also from non-managed code, i.e. code not running as an SCA component.

206 4.1 Accessing Services from an SCA Component

207 An SCA component may obtain a service reference either through injection or programmatically
208 through the **ComponentContext** API. Using reference injection is the recommended way to
209 access a service, since it results in code with minimal use of middleware APIs. The
210 ComponentContext API is provided for use in cases where reference injection is not possible.

211 4.1.1 Using the Component Context API

212 When a component implementation needs access to a service where the reference to the service is
213 not known at compile time, the reference can be located using the component's
214 ComponentContext.

215 4.2 Accessing Services from non-SCA component implementations

216 This section describes how Java code not running as an SCA component that is part of an SCA
217 composite accesses SCA services via references.

218 4.2.1 ComponentContext

219 Non-SCA client code can use the ComponentContext API to perform operations against a
220 component in an SCA domain. How client code obtains a reference to a ComponentContext is
221 runtime specific.

222 The following example demonstrates the use of the component Context API by non-SCA code:

223

```
224 ComponentContext context = // obtained through host environment-specific means  
225 HelloService helloService =  
226     context.getService(HelloService.class, "HelloService");  
227 String result = helloService.hello("Hello World!");
```

228 5 Error Handling

229 Clients calling service methods may experience business exceptions and SCA runtime exceptions.

230 Business exceptions are thrown by the implementation of the called service method, and are
231 defined as checked exceptions on the interface that types the service.

232 SCA runtime exceptions are raised by the SCA runtime and signal problems in management of
233 component execution or problems interacting with remote services. The SCA runtime exceptions
234 are [defined in the Java API section](#).

235 6 Asynchronous and Conversational Programming

236 Asynchronous programming of a service is where a client invokes a service and carries on
237 executing without waiting for the service to execute. Typically, the invoked service executes at
238 some later time. Output from the invoked service, if any, must be fed back to the client through a
239 separate mechanism, since no output is available at the point where the service is invoked. This is
240 in contrast to the call-and-return style of synchronous programming, where the invoked service
241 executes and returns any output to the client before the client continues. The SCA asynchronous
242 programming model consists of:

- 243 • support for non-blocking method calls
- 244 • conversational services
- 245 • callbacks

246 Each of these topics is discussed in the following sections.

247 Conversational services are services where there is an ongoing sequence of interactions between
248 the client and the service provider, which involve some set of state data – in contrast to the
249 simple case of stateless interactions between a client and a provider. Asynchronous services may
250 often involve the use of a conversation, although this is not mandatory.

251 6.1 @OneWay

252 **Nonblocking calls** represent the simplest form of asynchronous programming, where the client of
253 the service invokes the service and continues processing immediately, without waiting for the
254 service to execute.

255 Any method with a void return type and has no declared exceptions may be marked with an
256 **@OneWay** annotation. This means that the method is non-blocking and communication with the
257 service provider may use a binding that buffers the requests and sends it at some later time.

258 For a Java client to make a non-blocking call to methods that either return values or which throw
259 exceptions, a Java client can use the JAX-WS asynchronous client API model that is described in
260 section 9. It is considered to be a best practice that service designers define one-way methods as
261 often as possible, in order to give the greatest degree of binding flexibility to deployers.

262 6.2 Conversational Services

263 A service may be declared as conversational by marking its Java interface with an
264 **@Conversational** annotation. If a service interface is not marked with **@Conversational**, it is
265 stateless.

266 6.2.1 ConversationAttributes

267 A Java-based implementation class may be marked with an **@ConversationAttributes** annotation,
268 which is used to specify the expiration rules for conversational implementation instances.

269 An example of **@ConversationAttributes** is shown below:

```
270 package com.bigbank;  
271 import org.osoa.sca.annotations.ConversationAttributes;  
272  
273 @ConversationAttributes(maxAge="30 days");  
274 public class LoanServiceImpl implements LoanService {  
275  
276 }
```

277 6.2.2 @EndsConversation

278 A method of a conversational interface may be marked with an @EndsConversation annotation.
279 Once a method marked with @EndsConversation has been called, the conversation between client
280 and service provider is at an end, which implies no further methods may be called on that service
281 within the same conversation. This enables both the client and the service provider to free up
282 resources that were associated with the conversation.

283 It is also possible to mark a method on a callback interface (described later) with
284 @EndsConversation, in order for the service provider to be the party that chooses to end the
285 conversation.

286 If a conversation is ended with an explicit outbound call to an @EndsConversation method or
287 through a call to the ServiceReference.endConversation() method, then any subsequent call to an
288 operation on the service reference will start a new conversation. If the conversation ends for any
289 other reason (e.g. a timeout occurred), then until ServiceReference.getConversation().end() is
290 called, the ConversationEndedException is thrown by any conversational operation.

291 6.3 Passing Conversational Services as Parameters

292 The service reference which represents a single conversation can be passed as a parameter to
293 another service, even if that other service is remote. This may be used to allow one component to
294 continue a conversation that had been started by another.

295 A service provider may also create a service reference for itself that it can pass to other services.
296 A service implementation does this with a call to the createSelfReference(...) method:

```
297     interface ComponentContext{  
298         ...  
299         <B> ServiceReference<B> createSelfReference(Class  
300             businessInterface);  
301         <B> ServiceReference<B> createSelfReference(Class  
302             businessInterface, String serviceName);  
303     }
```

304
305 The second variant, which takes an additional **serviceName** parameter, must be used if the
306 component implements multiple services.

307 This capability may be used to support complex callback patterns, such as when a callback is
308 applicable only to a subset of a larger conversation. Simple callback patterns are handled by the
309 built-in callback support described later.

310 6.4 Conversational Client

311 The client of a conversational service does not need to be coded in a special way. The client can
312 take advantage of the conversational nature of the interface through the relationship of the
313 different methods in the interface and any data they may share in common. If the service is
314 asynchronous, the client may like to use a feature such as the conversationID to keep track of any
315 state data relating to the conversation.

316 The developer of the client knows that the service is conversational by introspecting the service
317 contract. The following shows how a client accesses the conversational service described above:

```
318  
319     @Reference  
320     LoanService loanService;  
321     // Known to be conversational because the interface is marked as  
322     // conversational
```



```

323     public void applyForMortgage(Customer customer, HouseInfo houseInfo,
324                               int term)
325     {
326         LoanApplication loanApp;
327         loanApp = createApplication(customer, houseInfo);
328         loanService.apply(loanApp);
329         loanService.lockCurrentRate(term);
330     }
331
332     public boolean isApproved() {
333         return loanService.getLoanStatus().equals("approved");
334     }
335     public LoanApplication createApplication(Customer customer,
336                                           HouseInfo houseInfo) {
337         return ...;
338     }

```

339 6.5 Conversation Lifetime Summary

340 ***Starting conversations***

341 Conversations start on the client side when one of the following occur:

- 342 • A @Reference to a conversational service is injected
- 343 • A call is made to CompositeContext.getServiceReference and then a method of the service
- 344 is called.

346 ***Continuing conversations***

347 The client can continue an existing conversation, by:

- 348 • Holding the service reference that was created when the conversation started
- 349 • Getting the service reference object passed as a parameter from another service, even
- 350 remotely
- 351 • Loading a service reference that had been written to some form of persistent storage

353 ***Ending conversations***

354 A conversation ends, and any state associated with the conversation is freed up, when:

- 355 • A service operation that has been annotated @EndsConversation has been called
- 356 • The server calls an @EndsConversation method on the @Callback reference
- 357 • The server's conversation lifetime timeout occurs
- 358 • The client calls Conversation.end()
- 359 • Any non-business exception is thrown by a conversational operation

360
361 If a method is invoked on a service reference after an @EndsConversation method has been called
362 then a new conversation will automatically be started. If
363 ServiceReference.getConversationID() is called after the @EndsConversation method is called,
364 but before the next conversation has been started, it returns null.

365 If a service reference is used after the service provider's conversation timeout has caused the
366 conversation to be ended, then ConversationEndedException is thrown. In order to use that
367 service reference for a new conversation, its endConversation () method must be called.
368

369 6.6 Conversation ID

370 Every conversation has a **conversation ID**. The conversation ID can be generated by the system,
371 or it can be supplied by the client component.

372 If a field or setter method is annotated with **@ConversationID**, then the conversation ID for the
373 conversation is injected. The type of the field is not necessarily String. System generated
374 conversation IDs are always strings, but application generated conversation IDs may be other
375 complex types.

376 6.6.1 Application Specified Conversation IDs

377 It is possible to take advantage of the state management aspects of conversational services while
378 using a client-provided conversation ID. To do this, the client does not use reference injection,
379 but uses the **ServiceReference.setConversationID()** API.

380 The conversation ID that is passed into this method should be an instance of either a String or of
381 an object that is serializable into XML. The ID must be unique to the client component over all
382 time. If the client is not an SCA component, then the ID must be globally unique.

383 Not all conversational service bindings support application-specified conversation IDs or may only
384 support application-specified conversation IDs that are Strings.

385 6.6.2 Accessing Conversation IDs from Clients

386 Whether the conversation ID is chosen by the client or is generated by the system, the client may
387 access the conversation ID by calling **getConversationID()** on the current conversation
388 object.

389 If the conversation ID is not application specified, then the
390 **ServiceReference.getConversationID()** method is only guaranteed to return a valid value
391 after the first operation has been invoked, otherwise it returns null.

392 6.7 Callbacks

393 A **callback service** is a service that is used for **asynchronous** communication from a service
394 provider back to its client, in contrast to the communication through return values from
395 synchronous operations. Callbacks are used by **bidirectional services**, which are services that
396 have two interfaces:

- 397 • an interface for the provided service
- 398 • a callback interface that must be provided by the client

399 Callbacks may be used for both remotable and local services. Either both interfaces of a
400 bidirectional service must be remotable, or both must be local. It is illegal to mix the two. There
401 are two basic forms of callbacks: stateless callbacks and stateful callbacks.

402 A callback interface is declared by using an **@Callback** annotation on a service interface, with the
403 Java Class object of the interface as a parameter. The annotation may also be applied to a method
404 or to a field of an implementation, which is used in order to have a callback injected, as explained
405 in the next section.

406 6.7.1 Stateful Callbacks

407 A **stateful** callback represents a specific implementation instance of the component that is the
408 client of the service. The interface of a stateful callback should be marked as **conversational**.

409 The following example interfaces show an interaction over a stateful callback.

```
410 package somepackage;
411 import org.osoa.sca.annotations.Callback;
412 import org.osoa.sca.annotations.Conversational;
413 import org.osoa.sca.annotations.Remotable;
414 @Remotable
415 @Conversational
416 @Callback(MyServiceCallback.class)
417 public interface MyService {
418
419     void someMethod(String arg);
420 }
421
422 @Remotable
423 @Conversational
424 public interface MyServiceCallback {
425
426     void receiveResult(String result);
427 }
428
```

429 An implementation of the service in this example could use the `@Callback` annotation to request
430 that a stateful callback be injected. The following is a fragment of an implementation of the
431 example service. In this example, the request is passed on to some other component, so that the
432 example service acts essentially as an intermediary. If the example service is conversation
433 scoped, the callback will still be available when the backend service sends back its asynchronous
434 response.

435 When an interface and its callback interface are both marked as conversational, then there is only
436 one conversation that applies in both directions and it has the same lifetime. In this case, if both
437 interfaces declare a `@ConversationAttributes` annotation, then only the annotation on the main
438 interface applies.

```
439 @Callback
440 protected MyServiceCallback callback;
441
442 @Reference
443 protected MyService backendService;
444
445 public void someMethod(String arg) {
446     backendService.someMethod(arg);
447 }
448
449 public void receiveResult(String result) {
450     callback.receiveResult(result);
451 }
452
```

453
454 This fragment must come from an implementation that offers two services, one that it offers to its
455 clients (`MyService`) and one that is used for receiving callbacks from the back end
456 (`MyServiceCallback`). The code snippet below is taken from the client of this service, which also
457 implements the methods defined in `MyServiceCallback`.

458

```

459
460     private MyService myService;
461
462     @Reference
463     public void setMyService(MyService service) {
464         myService = service;
465     }
466
467     public void aClientMethod() {
468         ...
469         myService.someMethod(arg);
470     }
471
472     public void receiveResult(String result) {
473         // code to process the result
474     }
475

```

476 Stateful callbacks support some of the same use cases as are supported by the ability to pass
477 service references as parameters. The primary difference is that stateful callbacks do not require
478 any additional parameters be passed with service operations. This can be a great convenience. If
479 the service has many operations and any of those operations could be the first operation of the
480 conversation, it would be unwieldy to have to take a callback parameter as part of every
481 operation, just in case it is the first operation of the conversation. It is also more natural than
482 requiring application developers to invoke an explicit operation whose only purpose is to pass the
483 callback object that should be used.

484 6.7.2 Stateless Callbacks

485 A stateless callback interface is a callback whose interface is not marked as **conversational**.
486 Unlike stateful services, a client that uses stateless callbacks will not have callback methods
487 routed to an instance of the client that contains any state that is relevant to the conversation. As
488 such, it is the responsibility of such a client to perform any persistent state management itself.
489 The only information that the client has to work with (other than the parameters of the callback
490 method) is a callback ID object that is passed with requests to the service and is guaranteed to be
491 returned with any callback.

492 The following is a repeat of the client code fragment above, but with the assumption that in this
493 case the MyServiceCallback is stateless. The client in this case needs to set the callback ID before
494 invoking the service and then needs to get the callback ID when the response is received.

```

495
496     private ServiceReference<MyService> myService;
497
498     @Reference
499     public void setMyService(ServiceReference<MyService> service) {
500         myService = service;
501     }
502
503     public void aClientMethod() {
504         String someKey = "1234";
505         ...
506
507         myService.setCallbackID(someKey);
508         myService.getService().someMethod(arg);
509     }
510     public void receiveResult(String result) {
511         Object key = myService.getCallbackID();
512         // Lookup any relevant state based on "key"
513         // code to process the result

```

514 }

515

516 Just as with stateful callbacks, a service implementation gets access to the callback object by
517 annotating a field or setter method with the `@Callback` annotation, such as the following:

518

519 `@Callback`

520 `protected MyServiceCallback callback;`

521

522 The difference for stateless services is that the callback field would not be available if the
523 component is servicing a request for anything other than the original client. So, the technique
524 used in the previous section, where there was a response from the backendService which was
525 forwarded as a callback from MyService would not work because the callback field would be null
526 when the message from the backend system was received.

527 6.7.3 Implementing Multiple Bidirectional Interfaces

528 Since it is possible for a single implementation class to implement multiple services, it is also
529 possible for callbacks to be defined for each of the services that it implements. The service
530 implementation can include an injected field for each of its callbacks. The runtime injects the
531 callback onto the appropriate field based on the type of the callback. The following shows the
532 declaration of two fields, each of which corresponds to a particular service offered by the
533 implementation.

534

535 `@Callback`

536 `protected MyService1Callback callback1;`

537

538 `@Callback`

539 `protected MyService2Callback callback2;`

540

541 If a single callback has a type that is compatible with multiple declared callback fields, then all of
542 them will be set.

543 6.7.4 Accessing Callbacks

544 In addition to injecting a reference to a callback service, it is also possible to obtain a reference to
545 a Callback instance by annotating a field or method with the `@Callback` annotation.

546

547 A reference implementing the callback service interface may be obtained using
548 `CallableReference.getService()`.

549 The following example fragments come from a service implementation that uses the callback API:

550

551 `@Callback`

552 `protected CallableReference<MyCallback> callback;`

553

554 `public void someMethod() {`

555

556 `MyCallback myCallback = callback.getCallback(); ...`

557

558 `myCallback.receiveResult(theResult);`

559 `}`

560

561 Alternatively, a callback may be retrieved programmatically using the `RequestContext` API. The
562 snippet below shows how to retrieve a callback in a method programmatically:

563

```
564 public void someMethod() {
565     MyCallback myCallback =
566         ComponentContext.getRequestContext().getCallback();
567     ...
568     myCallback.receiveResult(theResult);
569 }
570
571
572
573
```

574 On the client side, the service that implements the callback can access the callback ID (i.e.,
575 reference parameters) that was returned with the callback operation by accessing the request
576 context, as follows:

577

```
578 @Context
579 protected RequestContext requestContext;
580
581 void receiveResult(Object theResult) {
582     Object refParams =
583         requestContext.getServiceReference().getCallbackID();
584     ...
585 }
586
```

587

588 On the client side, the object returned by the `getServiceReference()` method represents the
589 service reference that was used to send the original request. The object returned by
590 `getCallbackID()` represents the identity associated with the callback, which may be a single
591 String or may be an object (as described below in “Customizing the Callback Identity”).

592 6.7.5 Customizing the Callback

593 By default, the client component of a service is assumed to be the callback service for the
594 bidirectional service. However, it is possible to change the callback by using the
595 **`ServiceReference.setCallback()`** method. The object passed as the callback should implement
596 the interface defined for the callback, including any additional SCA semantics on that interface
597 such as whether or not it is remotable.

598 Since a service other than the client can be used as the callback implementation, SCA does not
599 generate a deployment-time error if a client does not implement the callback interface of one of its
600 references. However, if a call is made on such a reference without the `setCallback()` method
601 having been called, then a **`NoRegisteredCallbackException`** is thrown on the client.

602 A callback object for a stateful callback interface has the additional requirement that it must be
603 serializable. The SCA runtime may serialize a callback object and persistently store it.

604 A callback object may be a service reference to another service. In that case, the callback
605 messages go directly to the service that has been set as the callback. If the callback object is not
606 a service reference, then callback messages go to the client and are then routed to the specific
607 instance that has been registered as the callback object. However, if the callback interface has a
608 stateless scope, then the callback object **must** be a service reference.

609 6.7.6 Customizing the Callback Identity

610 The identity that is used to identify a callback request is initially generated by the system.
611 However, it is possible to provide an application specified identity to identify the callback by calling
612 the **`ServiceReference.setCallbackID()`** method. This can be used both for stateful and for

613 stateless callbacks. The identity is sent to the service provider, and the binding must guarantee
614 that the service provider will send the ID back when any callback method is invoked.

615 The callback identity has the same restrictions as the conversation ID. It should either be a string
616 or an object that can be serialized into XML. Bindings determine the particular mechanisms to use
617 for transmission of the identity and these may lead to further restrictions when using a given
618 binding.

619 **6.7.7 Bindings for Conversations and Callbacks**

620 There are potentially many ways of representing the conversation ID for conversational services
621 depending on the type of binding that is used. For example, it may be possible WS-RM sequence
622 ids for the conversation ID if reliable messaging is used in a Web services binding. WS-Eventing
623 uses a different technique (the wse:Identity header). There is also a WS-Context OASIS TC that
624 is creating a general purpose mechanism for exactly this purpose.

625 SCA's programming model supports conversations, but it leaves up to the binding the means by
626 which the conversation ID is represented on the wire.

627 7 Java API

628 This section provides a reference for the Java API offered by SCA.

629 7.1 Component Context

630 The following Java code defines the **ComponentContext** interface:

```
631
632 package org.osoa.sca;
633
634 public interface ComponentContext {
635     String getURI();
636
637     <B> B getService(Class<B> businessInterface, String referenceName);
638
639     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
640                                             String referenceName);
641
642     <B> Collection<B> getServices(Class<B> businessInterface,
643                               String referenceName);
644
645     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
646                                                         businessInterface, String referenceName);
647
648     <B> ServiceReference<B> createSelfReference(Class<B>
649                                               businessInterface);
650
651     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
652                                               String serviceName);
653
654     <B> B getProperty(Class<B> type, String propertyName);
655
656     <B, R extends CallableReference<B>> R cast(B target)
657                                     throws IllegalArgumentException;
658
659     RequestContext getRequestContext();
660
661     <B> ServiceReference<B> cast(B target) throws IllegalArgumentException;
662 }
```

- 663
- 664 • **getURI()** - returns the absolute URI of the component within the SCA domain
 - 665 • **getService(Class businessInterface, String referenceName)** – Returns a proxy for
666 the reference defined by the current component. The getService() method takes as its
667 input arguments the Java type used to represent the target service on the client and the
668 name of the service reference. It returns an object providing access to the service. The
669 returned object implements the Java interface the service is typed with. This method
670 MUST throw an IllegalArgumentException if the reference has multiplicity greater than
671 one.
 - 672 • **getServiceReference(Class businessInterface, String referenceName)** – Returns a
673 ServiceReference defined by the current component. This method MUST throw an
674 IllegalArgumentException if the reference has multiplicity greater than one.

- 675 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
676 typed service proxies for a business interface type and a reference name.
- 677 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
678 list typed service references for a business interface type and a reference name.
- 679 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
680 be used to invoke this component over the designated service.
- 681 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
682 ServiceReference that can be used to invoke this component over the designated service.
683 Service name explicitly declares the service name to invoke
- 684 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
685 property defined by this component.
- 686 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
687 there is no current request or if the context is unavailable. This method MUST return non-
688 null when invoked during the execution of a Java business method for a service operation
689 or callback operation, on the same thread that the SCA runtime provided, and MUST
690 return null in all other cases.
- 691 • **cast(B target)** - Casts a type-safe reference to a CallableReference

692 A component may access its component context by defining a field or setter method typed by
693 **org.osoa.sca.ComponentContext** and annotated with **@Context**. To access the target service,
694 the component uses **ComponentContext.getService(..)**.

695
696 The following shows an example of component context usage in a Java class using the @Context
697 annotation.

```
698 private ComponentContext componentContext;
699
700 @Context
701 public void setContext(ComponentContext context) {
702     componentContext = context;
703 }
704
705 public void doSomething() {
706     HelloWorld service =
707         componentContext.getService(HelloWorld.class, "HelloWorldComponent");
708     service.hello("hello");
709 }
710
```

711 Similarly, non-SCA client code can use the ComponentContext API to perform operations against a
712 component in an SCA domain. How the non-SCA client code obtains a reference to a
713 ComponentContext is runtime specific.

714 7.2 Request Context

715 The following shows the **RequestContext** interface:

```
716
717 package org.osoa.sca;
718
719 import javax.security.auth.Subject;
720
721 public interface RequestContext {
722
723     Subject getSecuritySubject();
724
```

```

725     String getServiceName();
726     <CB> CallableReference<CB> getCallbackReference();
727     <CB> CB getCallback();
728     <B> CallableReference<B> getServiceReference();
729
730 }
731

```

732 The RequestContext interface has the following methods:

- 733 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 734 • **getServiceName()** – Returns the name of the service on the Java implementation the
735 request came in on
- 736 • **getCallbackReference()** – Returns a callable reference to the callback as specified by the
737 caller
- 738 • **getCallback()** – Returns a proxy for the callback as specified by the caller
- 739 • **getServiceReference()** – When invoked during the execution of a service operation, this
740 method MUST return a CallableReference that represents the service that was invoked.
741 When invoked during the execution of a callback operation, this method MUST return a
742 CallableReference that represents the callback that was invoked.

743 7.3 CallableReference

744 The following Java code defines the **CallableReference** interface:

```

745
746 package org.osoa.sca;
747
748 public interface CallableReference<B> extends java.io.Serializable {
749
750     B getService();
751     Class<B> getBusinessInterface();
752     boolean isConversational();
753     Conversation getConversation();
754     Object getCallbackID();
755 }
756

```

757 The CallableReference interface has the following methods:

- 758
- 759 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
760 returned is guaranteed to implement the business interface for this reference. The value
761 returned is a proxy to the target that implements the business interface associated with this
762 reference.
- 763 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
764 this reference.
- 765 • **isConversational()** – Returns true if this reference is conversational.
- 766 • **getConversation()** – Returns the conversation associated with this reference. Returns null if
767 no conversation is currently active.
- 768 • **getCallbackID()** – Returns the callback ID.

769 7.4 ServiceReference

770

771 ServiceReferences may be injected using the @Reference annotation on a field, a setter method,
772 or constructor parameter taking the type ServiceReference. The detailed description of the usage
773 of these methods is described in the section on Asynchronous Programming in this document.

774 The following Java code defines the ServiceReference interface:

```
775  
776 package org.osoa.sca;  
777  
778 public interface ServiceReference<B> extends CallableReference<B> {  
779  
780     Object getConversationID();  
781     void setConversationID(Object conversationId) throws  
782         IllegalStateException;  
783     void setCallbackID(Object callbackID);  
784     Object getCallback();  
785     void setCallback(Object callback);  
786 }  
787
```

788 The ServiceReference interface has the methods of CallableReference plus the following:

- 789
- 790 • **getConversationID()** - Returns the id supplied by the user that will be associated with
791 future conversations initiated through this reference, or null if no ID has been set by the
792 user.
 - 793 • **setConversationID(Object conversationId)** – Set the ID, supplied by the user, to associate
794 with any future conversation started through this reference. If the value supplied is null then
795 the id will be generated by the implementation. Throws an IllegalStateException if a
796 conversation is currently associated with this reference.
 - 797 • **setCallbackID(Object callbackID)** – Sets the callback ID.
 - 798 • **getCallback()** – Returns the callback object.
 - 799 • **setCallback(Object callback)** – Sets the callback object.

800 7.5 Conversation

801 The following snippet defines Conversation:

```
802  
803 package org.osoa.sca;  
804  
805 public interface Conversation {  
806     Object getConversationID();  
807     void end();  
808 }  
809
```

810 The Conversation interface has the following methods:

- 811 • **getConversationID()** – Returns the identifier for this conversation. If a user-defined identity
812 had been supplied for this reference then its value will be returned; otherwise the identity
813 generated by the system when the conversation was initiated will be returned.
- 814 • **end()** – Ends this conversation.

815 7.6 ServiceRuntimeException

816 The following snippet shows the **ServiceRuntimeException**.

817

```
818 package org.osoa.sca;  
819  
820 public class ServiceRuntimeException extends RuntimeException {  
821     ...  
822 }  
823
```

824 This exception signals problems in the management of SCA component execution.

825 7.7 NoRegisteredCallbackException

826 The following snippet shows the *NoRegisteredCallbackException*.

```
827 package org.osoa.sca;  
828  
829 public class NoRegisteredCallbackException extends  
830     ServiceRuntimeException {  
831     ...  
832 }  
833
```

834 This exception signals a problem where an attempt is made to invoke a callback when a client
835 does not implement the Callback interface and no valid custom Callback has been specified via a
836 call to *ServiceReference.setCallback()*.

837 7.8 ServiceUnavailableException

838 The following snippet shows the *ServiceUnavailableException*.

```
839 package org.osoa.sca;  
840  
841 public class ServiceUnavailableException extends ServiceRuntimeException {  
842     ...  
843 }  
844  
845
```

846 This exception signals problems in the interaction with remote services. These are exceptions
847 that may be transient, so retrying is appropriate. Any exception that is a
848 ServiceRuntimeException that is *not* a ServiceUnavailableException is unlikely to be resolved by
849 retrying the operation, since it most likely requires human intervention

850 7.9 InvalidServiceException

851 The following snippet shows the *InvalidServiceException*.

```
852 package org.osoa.sca;  
853  
854 public class InvalidServiceException extends ServiceRuntimeException {  
855     ...  
856 }  
857  
858
```

859 This exception signals that the ServiceReference is no longer valid. This can happen when the
860 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
861 be resolved by retrying the operation and will most likely require human intervention.

862 7.10 ConversationEndedException

863 The following snippet shows the *ConversationEndedException*.

```
864
865     package org.osoa.sca;
866
867     public class ConversationEndedException extends ServiceRuntimeException {
868         ...
869     }
870
```

871 8 Java Annotations

872 This section provides definitions of all the Java annotations which apply to SCA.

873 8.1 @AllowsPassByReference

874 The following Java code defines the **@AllowsPassByReference** annotation:

```
875
876 package org.osoa.sca.annotations;
877
878 import static java.lang.annotation.ElementType.TYPE;
879 import static java.lang.annotation.ElementType.METHOD;
880 import static java.lang.annotation.RetentionPolicy.RUNTIME;
881 import java.lang.annotation.Retention;
882 import java.lang.annotation.Target;
883
884 @Target({TYPE, METHOD})
885 @Retention(RUNTIME)
886 public @interface AllowsPassByReference {
887
888 }
889
```

890 The **@AllowsPassByReference** annotation is used on implementations of remotable interfaces to
891 indicate that interactions with the service from a client within the same address space are allowed
892 to use pass by reference data exchange semantics. The implementation promises that its by-value
893 semantics will be maintained even if the parameters and return values are actually passed by-
894 reference. This means that the service will not modify any operation input parameter or return
895 value, even after returning from the operation. Either a whole class implementing a remotable
896 service or an individual remotable service method implementation can be annotated using the
897 **@AllowsPassByReference** annotation.

898 **@AllowsPassByReference** has no attributes

899

900 The following snippet shows a sample where **@AllowsPassByReference** is defined for the
901 implementation of a service method on the Java component implementation class.

902

```
903 @AllowsPassByReference
904 public String hello(String message) {
905     ...
906 }
```

907 8.2 @Callback

908 The following Java code defines shows the **@Callback** annotation:

909

```
910 package org.osoa.sca.annotations;
911
912 import static java.lang.annotation.ElementType.TYPE;
913 import static java.lang.annotation.ElementType.METHOD;
914 import static java.lang.annotation.ElementType.FIELD;
915 import static java.lang.annotation.RetentionPolicy.RUNTIME;
916 import java.lang.annotation.Retention;
```

```

917     import java.lang.annotation.Target;
918
919     @Target(TYPE, METHOD, FIELD)
920     @Retention(RUNTIME)
921     public @interface Callback {
922
923         Class<?> value() default Void.class;
924     }

```

925
926
927 The @Callback annotation is used to annotate a service interface with a callback interface, which
928 takes the Java Class object of the callback interface as a parameter.

929 The @Callback annotation has the following attribute:

- 930 • **value** – the name of a Java class file containing the callback interface

931

932 The @Callback annotation may also be used to annotate a method or a field of an SCA
933 implementation class, in order to have a callback object injected

934

935 The following snippet shows a callback annotation on an interface:

936

```

937     @Remotable
938     @Callback(MyServiceCallback.class)
939     public interface MyService {
940
941         void someAsyncMethod(String arg);
942     }
943

```

944 An example use of the @Callback annotation to declare a callback interface follows:

945

```

946     package somepackage;
947     import org.oesa.sca.annotations.Callback;
948     import org.oesa.sca.annotations.Remotable;
949     @Remotable
950     @Callback(MyServiceCallback.class)
951     public interface MyService {
952
953         void someMethod(String arg);
954     }
955
956     @Remotable
957     public interface MyServiceCallback {
958
959         void receiveResult(String result);
960     }

```

961

962 In this example, the implied component type is:

963

```

964     <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
965         <service name="MyService">
966             <interface.java interface="somepackage.MyService"

```

```
968         callbackInterface="somepackage.MyServiceCallback" />
969     </service>
970 </componentType>
```

971 8.3 @ComponentName

972 The following Java code defines the **@ComponentName** annotation:

```
973
974 package org.osoa.sca.annotations;
975
976 import static java.lang.annotation.ElementType.METHOD;
977 import static java.lang.annotation.ElementType.FIELD;
978 import static java.lang.annotation.RetentionPolicy.RUNTIME;
979 import java.lang.annotation.Retention;
980 import java.lang.annotation.Target;
981
982 @Target({METHOD, FIELD})
983 @Retention(RUNTIME)
984 public @interface ComponentName {
985
986 }
987
```

988 The @ComponentName annotation is used to denote a Java class field or setter method that is
989 used to inject the component name.

990

991 The following snippet shows a component name field definition sample.

992

```
993 @ComponentName
994 private String componentName;
995
```

996 The following snippet shows a component name setter method sample.

997

```
998 @ComponentName
999 public void setComponentName(String name) {
1000     //...
1001 }
```

1002 8.4 @Constructor

1003 The following Java code defines the **@Constructor** annotation:

1004

```
1005 package org.osoa.sca.annotations;
1006
1007 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1008 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1009 import java.lang.annotation.Retention;
1010 import java.lang.annotation.Target;
1011
1012 @Target(CONSTRUCTOR)
1013 @Retention(RUNTIME)
1014 public @interface Constructor {
1015     String[] value() default "";
1016 }
```


1017

1018 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1019 Java component implementation.

1020 The @Constructor annotation has the following attribute:

- 1021 • **value (optional)** – identifies the property/reference names that correspond to each of the
1022 constructor arguments. The position in the array determines which of the arguments are
1023 being named.

1024 The following snippet shows a sample for the Constructor annotation.

1025

```
1026 public class HelloServiceImpl implements HelloService {  
1027     public HelloServiceImpl(){  
1028         ...  
1029     }  
1030  
1031     @Constructor  
1032     public HelloServiceImpl( String someProperty ){  
1033         ...  
1034     }  
1035  
1036     public String hello(String message) {  
1037         ...  
1038     }  
1039 }  
1040 }
```

1041 8.5 @Context

1042 The following Java code defines the **@Context** annotation:

1043

```
1044 package org.osoa.sca.annotations;  
1045  
1046 import static java.lang.annotation.ElementType.METHOD;  
1047 import static java.lang.annotation.ElementType.FIELD;  
1048 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1049 import java.lang.annotation.Retention;  
1050 import java.lang.annotation.Target;  
1051  
1052 @Target({METHOD, FIELD})  
1053 @Retention(RUNTIME)  
1054 public @interface Context {  
1055  
1056 }  
1057 }
```

1058 The @Context annotation is used to denote a Java class field or a setter method that is used to
1059 inject a composite context for the component. The type of context to be injected is defined by the
1060 type of the Java class field or type of the setter method input argument; the type is either
1061 **ComponentContext** or **RequestContext**.

1062 The @Context annotation has no attributes.

1063

1064 The following snippet shows a ComponentContext field definition sample.

1065

```
1066 @Context
1067 protected ComponentContext context;
1068
```

1069 The following snippet shows a RequestContext field definition sample.

```
1070
1071 @Context
1072 protected RequestContext context;
```

1073 8.6 @Conversational

1074 The following Java code defines the **@Conversational** annotation:

```
1075
1076 package org.osoa.sca.annotations;
1077
1078 import static java.lang.annotation.ElementType.TYPE;
1079 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1080 import java.lang.annotation.Retention;
1081 import java.lang.annotation.Target;
1082 @Target(TYPE)
1083 @Retention(RUNTIME)
1084 public @interface Conversational {
1085 }
1086
```

1087 The @Conversational annotation is used on a Java interface to denote a conversational service
1088 contract.

1089 The @Conversational annotation has no attributes.

1090 The following snippet shows a sample for the Conversational annotation.

```
1091 package services.hello;
1092
1093 import org.osoa.sca.annotations.Conversational;
1094
1095 @Conversational
1096 public interface HelloService {
1097     void setName(String name);
1098     String sayHello();
1099 }
```

1100 8.7 @ConversationAttributes

1101 The following Java code defines the **@ConversationAttributes** annotation:

```
1102
1103 package org.osoa.sca.annotations;
1104
1105 import static java.lang.annotation.ElementType.TYPE;
1106 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1107 import java.lang.annotation.Retention;
1108 import java.lang.annotation.Target;
1109
1110 @Target(TYPE)
1111 @Retention(RUNTIME)
1112 public @interface ConversationAttributes {
1113
1114     String maxIdleTime() default " ";
```

```

1115     String maxAge() default "";
1116     boolean singlePrincipal() default false;
1117 }
1118

```

1119 The `@ConversationAttributes` annotation is used to define a set of attributes which apply to
1120 conversational interfaces of services or references of a Java class. The annotation has the following
1121 attributes:

- 1122 • ***maxIdleTime (optional)*** - The maximum time that can pass between successive
1123 operations within a single conversation. If more time than this passes, then the container
1124 may end the conversation.
- 1125 • ***maxAge (optional)*** - The maximum time that the entire conversation can remain active.
1126 If more time than this passes, then the container may end the conversation.
- 1127 • ***singlePrincipal (optional)*** – If true, only the principal (the user) that started the
1128 conversation has authority to continue the conversation. The default value is false.

1129

1130 The two attributes that take a time express the time as a string that starts with an integer, is
1131 followed by a space and then one of the following: "seconds", "minutes", "hours", "days" or
1132 "years".

1133

1134 Not specifying timeouts means that timeouts are defined by the SCA runtime implementation,
1135 however it chooses to do so.

1136

1137 The following snippet shows the use of the `@ConversationAttributes` annotation to set the
1138 maximum age for a Conversation to be 30 days.

1139

```

1140 package service.shoppingcart;
1141
1142 import org.osoa.sca.annotations.ConversationAttributes;
1143
1144 @ConversationAttributes (maxAge="30 days");
1145 public class ShoppingCartServiceImpl implements ShoppingCartService {
1146     ...
1147 }

```

1148 8.8 @ConversationID

1149 The following Java code defines the `@ConversationID` annotation:

1150

```

1151 package org.osoa.sca.annotations;
1152
1153 import static java.lang.annotation.ElementType.METHOD;
1154 import static java.lang.annotation.ElementType.FIELD;
1155 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1156 import java.lang.annotation.Retention;
1157 import java.lang.annotation.Target;
1158
1159 @Target({METHOD, FIELD})
1160 @Retention(RUNTIME)
1161 public @interface ConversationID {
1162
1163 }

```

1164

1165 The @ConversationID annotation is used to annotate a Java class field or setter method that is
1166 used to inject the conversation ID. System generated conversation IDs are always strings, but
1167 application generated conversation IDs may be other complex types.

1168 The following snippet shows a conversation ID field definition sample.

1169

```
1170 @ConversationID  
1171 private String conversationID;
```

1172

1173 The type of the field is not necessarily String.

1174

1175 8.9 @Destroy

1176 The following Java code defines the **@Destroy** annotation:

1177

```
1178 package org.osoa.sca.annotations;  
1179  
1180 import static java.lang.annotation.ElementType.METHOD;  
1181 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1182 import java.lang.annotation.Retention;  
1183 import java.lang.annotation.Target;  
1184  
1185 @Target(METHOD)  
1186 @Retention(RUNTIME)  
1187 public @interface Destroy {  
1188  
1189 }  
1190
```

1191 The @Destroy annotation is used to denote a single Java class method that will be called when the
1192 scope defined for the implementation class ends. The method MAY have any access modifier and
1193 MUST have a void return value and no arguments.

1194 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1195 when the scope defined for the implementation class ends. If the implementation class has a
1196 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1197 NOT instantiate the implementation class.

1198

1199 The following snippet shows a sample for a destroy method definition.

1200

```
1201 @Destroy  
1202 void myDestroyMethod() {  
1203     ...  
1204 }
```

1205 8.10 @EagerInit

1206 The following Java code defines the **@EagerInit** annotation:

1207

```
1208 package org.osoa.sca.annotations;  
1209
```

```

1210     import static java.lang.annotation.ElementType.TYPE;
1211     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1212     import java.lang.annotation.Retention;
1213     import java.lang.annotation.Target;
1214
1215     @Target(TYPE)
1216     @Retention(RUNTIME)
1217     public @interface EagerInit {
1218
1219     }

```

1220 8.11 @EndsConversation

1221 The following Java code defines the **@EndsConversation** annotation:

```

1222
1223     package org.osoa.sca.annotations;
1224
1225     import static java.lang.annotation.ElementType.METHOD;
1226     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1227     import java.lang.annotation.Retention;
1228     import java.lang.annotation.Target;
1229
1230     @Target(METHOD)
1231     @Retention(RUNTIME)
1232     public @interface EndsConversation {
1233
1234     }
1235
1236

```

1237 The @EndsConversation annotation is used to denote a method on a Java interface that is called
1238 to end a conversation.

1239 The @EndsConversation annotation has no attributes.

1240 The following snippet shows a sample using the @EndsConversation annotation.

```

1241     package services.shoppingbasket;
1242
1243     import org.osoa.sca.annotations.EndsConversation;
1244
1245     public interface ShoppingBasket {
1246         void addItem(String itemID, int quantity);
1247
1248         @EndsConversation
1249         void buy();
1250     }

```

1251 8.12 @Init

1252 The following Java code defines the **@Init** annotation:

```

1253
1254     package org.osoa.sca.annotations;
1255
1256     import static java.lang.annotation.ElementType.METHOD;
1257     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1258     import java.lang.annotation.Retention;
1259     import java.lang.annotation.Target;

```

```

1260
1261 @Target(METHOD)
1262 @Retention(RUNTIME)
1263 public @interface Init {
1264
1265
1266 }
1267

```

1268 The @Init annotation is used to denote a single Java class method that is called when the scope
1269 defined for the implementation class starts. The method MAY have any access modifier and MUST
1270 have a void return value and no arguments.

1271 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1272 after all property and reference injection is complete. If the implementation class has a method
1273 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1274 instantiate the implementation class.

1275 The following snippet shows an example of an init method definition.

```

1276
1277 @Init
1278 public void myInitMethod() {
1279     ...
1280 }

```

1281 8.13 @OneWay

1282 The following Java code defines the **@OneWay** annotation:

```

1283
1284 package org.osoa.sca.annotations;
1285
1286 import static java.lang.annotation.ElementType.METHOD;
1287 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1288 import java.lang.annotation.Retention;
1289 import java.lang.annotation.Target;
1290
1291 @Target(METHOD)
1292 @Retention(RUNTIME)
1293 public @interface OneWay {
1294
1295
1296 }
1297

```

1298 The @OneWay annotation is used on a Java interface or class method to indicate that invocations
1299 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1300 Programming.

1301 The @OneWay annotation has no attributes.

1302 The following snippet shows the use of the @OneWay annotation on an interface.

```

1303 package services.hello;
1304
1305 import org.osoa.sca.annotations.OneWay;
1306
1307 public interface HelloService {
1308     @OneWay
1309     void hello(String name);
1310 }

```

1311 8.14 @Property

1312 The following Java code defines the **@Property** annotation:

1313

```
1314 package org.osoa.sca.annotations;
1315
1316 import static java.lang.annotation.ElementType.METHOD;
1317 import static java.lang.annotation.ElementType.FIELD;
1318 import static java.lang.annotation.ElementType.PARAMETER;
1319 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1320 import java.lang.annotation.Retention;
1321 import java.lang.annotation.Target;
1322
1323 @Target({METHOD, FIELD, PARAMETER})
1324 @Retention(RUNTIME)
1325 public @interface Property {
1326
1327     String name() default "";
1328     boolean required() default false;
1329 }
1330
```

1331 The @Property annotation is used to denote a Java class field or a setter method that is used to
1332 inject an SCA property value. The type of the property injected, which can be a simple Java type
1333 or a complex Java type, is defined by the type of the Java class field or the type of the setter
1334 method input argument.

1335 The @Property annotation may be used on fields, on setter methods or on a constructor method
1336 parameter.

1337 Properties may also be injected via setter methods even when the @Property annotation is not
1338 present. However, the @Property annotation must be used in order to inject a property onto a
1339 non-public field. In the case where there is no @Property annotation, the name of the property is
1340 the same as the name of the field or setter.

1341 Where there is both a setter method and a field for a property, the setter method is used.

1342

1343 The @Property annotation has the following attributes:

- 1344 • **name (optional)** – the name of the property, defaults to the name of the field of the Java
1345 class
- 1346 • **required (optional)** – specifies whether injection is required, defaults to false

1347

1348 The following snippet shows a property field definition sample.

1349

```
1350 @Property(name="currency", required=true)
1351 protected String currency;
```

1352

1353 The following snippet shows a property setter sample

1354

```
1355 @Property(name="currency", required=true)
1356 public void setCurrency( String theCurrency ) {
1357     ....
```

1358 }

1359

1360 If the property is defined as an array or as any type that extends or implements
1361 **java.util.Collection**, then the implied component type has a property with a **many** attribute set to
1362 true.

1363

1364 The following snippet shows the definition of a configuration property using the @Property
1365 annotation for a collection.

1366

```
1367 ...  
1368 private List<String> helloConfigurationProperty;  
1369  
1370 @Property(required=true)  
1371 public void setHelloConfigurationProperty(List<String> property) {  
1372     helloConfigurationProperty = property;  
1373 }  
1374 ...
```

1375 8.15 @Reference

1376 The following Java code defines the **@Reference** annotation:

1377

```
1378 package org.osoa.sca.annotations;  
1379  
1380 import static java.lang.annotation.ElementType.METHOD;  
1381 import static java.lang.annotation.ElementType.FIELD;  
1382 import static java.lang.annotation.ElementType.PARAMETER;  
1383 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1384 import java.lang.annotation.Retention;  
1385 import java.lang.annotation.Target;  
1386 @Target({METHOD, FIELD, PARAMETER})  
1387 @Retention(RUNTIME)  
1388 public @interface Reference {  
1389  
1390     String name() default "";  
1391     boolean required() default true;  
1392 }  
1393
```

1394 The @Reference annotation is used to denote a Java class field, a setter method, or a constructor
1395 parameter that is used to inject a service that resolves the reference. The interface of the service
1396 injected is defined by the type of the Java class field or the type of the setter method input
1397 argument.

1398 References may also be injected via setter methods even when the @Reference annotation is not
1399 present. However, the @Reference annotation must be used in order to inject a reference onto a
1400 non-public field. In the case where there is no @Reference annotation, the name of the reference
1401 is the same as the name of the field or setter.

1402 Where there is both a setter method and a field for a reference, the setter method is used.

1403 The @Reference annotation has the following attributes:

- 1404 • **name (optional)** – the name of the reference, defaults to the name of the field of the Java
1405 class
- 1406 • **required (optional)** – whether injection of service or services is required. Defaults to true.

1407

1408 The following snippet shows a reference field definition sample.

1409

```
1410 @Reference(name="stockQuote", required=true)
1411 protected StockQuoteService stockQuote;
```

1412

1413 The following snippet shows a reference setter sample

1414

```
1415 @Reference(name="stockQuote", required=true)
1416 public void setStockQuote( StockQuoteService theSQService );
```

1417

1418 The following fragment from a component implementation shows a sample of a service reference
1419 using the @Reference annotation. The name of the reference is "helloService" and its type is
1420 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1421 helloService reference.

1422

```
1423 package services.hello;
```

1424

```
1425 private HelloService helloService;
```

1426

```
1427 @Reference(name="helloService", required=true)
```

```
1428 public setHelloService(HelloService service) {
```

```
1429     helloService = service;
```

```
1430 }
```

1431

```
1432 public void clientMethod() {
```

```
1433     String result = helloService.hello("Hello World!");
```

```
1434     ...
```

```
1435 }
```

1436

1437 The presence of a @Reference annotation is reflected in the componentType information that the
1438 runtime generates through reflection on the implementation class. The following snippet shows
1439 the component type for the above component implementation fragment.

1440

```
1441 <?xml version="1.0" encoding="ASCII"?>
```

```
1442 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
```

1443

```
1444     <!-- Any services offered by the component would be listed here -->
```

```
1445     <reference name="helloService" multiplicity="1..1">
```

```
1446         <interface.java interface="services.hello.HelloService"/>
```

```
1447     </reference>
```

1448

```
1449 </componentType>
```

1450

1451 If the reference is not an array or collection, then the implied component type has a reference
1452 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**
1453 attribute – 1..1 applies if required=true.

1454

1455 If the reference is defined as an array or as any type that extends or implements **java.util.Collection**,
1456 then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending

1457 on whether the **required** attribute of the **@Reference** annotation is set to true or false – 1..n applies if
1458 required=true.

1459
1460 The following fragment from a component implementation shows a sample of a service reference
1461 definition using the **@Reference** annotation on a java.util.List. The name of the reference is
1462 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1463 services referenced by the helloServices reference. In this case, at least one HelloService should
1464 be present, so **required** is true.

```
1465     @Reference(name="helloServices", required=true)  
1466     protected List<HelloService> helloServices;  
1467  
1468     public void clientMethod() {  
1469  
1470         ...  
1471         for (int index = 0; index < helloServices.size(); index++) {  
1472             HelloService helloService =  
1473                 (HelloService)helloServices.get(index);  
1474             String result = helloService.hello("Hello World!");  
1475         }  
1476         ...  
1477     }  
1478 }  
1479
```

1480 The following snippet shows the XML representation of the component type reflected from for the
1481 former component implementation fragment. There is no need to author this component type in
1482 this case since it can be reflected from the Java class.

```
1483  
1484 <?xml version="1.0" encoding="ASCII"?>  
1485 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
1486  
1487     <!-- Any services offered by the component would be listed here -->  
1488     <reference name="helloServices" multiplicity="1..n">  
1489         <interface.java interface="services.hello.HelloService"/>  
1490     </reference>  
1491  
1492 </componentType>
```

1493
1494 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1495 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1496 of 0..N must be an empty array or collection.

1497 8.15.1 Reinjection

1498 References MAY be reinjected after the initial creation of a component if the reference target
1499 changes due to a change in wiring that has occurred since the component was initialized. In order
1500 for reinjection to occur, the following MUST be true:

- 1501 1. The component MUST NOT be STATELESS or REQUEST scoped.
- 1502 2. The reference MUST use either field-based injection or setter injection. References that are
1503 injected through constructor injection MUST NOT be changed. Setter injection allows for
1504 code in the setter method to perform processing in reaction to a change.
- 1505 3. If the reference has a conversational interface, then reinjection MUST NOT occur while the
1506 conversation is active.

1507 If a reference target changes and the reference is not reinjected, the reference MUST continue to
1508 work as if the reference target was not changed.

1509 If an operation is called on a reference where the target of that reference has been undeployed,
 1510 the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference
 1511 where the target of the reference has become unavailable for some reason, the SCA runtime
 1512 SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the
 1513 reference MAY continue to work, depending on the runtime and the type of change that was made.
 1514 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1515 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
 1516 corresponds to the reference that is passed as a parameter to cast(). If the reference is
 1517 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
 1518 to work as if the reference target was not changed. If the target of a ServiceReference has been
 1519 undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is
 1520 invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the
 1521 SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the
 1522 ServiceReference. If the target of a ServiceReference is changed, the reference MAY continue to
 1523 work, depending on the runtime and the type of change that was made. If it doesn't work, the
 1524 exception thrown will depend on the runtime and the cause of the failure.

1525 A reference or ServiceReference accessed through the component context by calling getService()
 1526 or getServiceReference() MUST correspond to the current configuration of the domain. This
 1527 applies whether or not reinjection has taken place. If the target has been undeployed or has
 1528 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
 1529 and attempts to call business methods SHOULD throw an exception as described above. If the
 1530 target has changed, the result SHOULD be a reference to the changed service.

1531 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This
 1532 means that in the cases listed above where reference reinjection is not allowed, the array or
 1533 Collection for the reference MUST NOT change its contents. In cases where the contents of a
 1534 reference collection MAY change, then for references that use setter injection, the setter method
 1535 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be
 1536 the same array or Collection object previously injected to the component.

1537

<u>Change event</u>	<u>Effect on</u>		
	Reference	Existing ServiceReference Object	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	MAY be reinjected (if other conditions* apply). If not reinjected, then it MUST continue to work as if the reference target was not changed.	MUST continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.
Target service undeployed	Business methods SHOULD throw InvalidServiceException.	Business methods SHOULD throw InvalidServiceException.	Result SHOULD be a reference to the undeployed or unavailable service. Business methods SHOULD throw InvalidServiceException.
Target service changed	MAY continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the	MAY continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the	Result SHOULD be a reference to the changed service.

	failure.	failure.	
<p>* Other conditions:</p> <ol style="list-style-type: none"> 1. The component MUST NOT be STATELESS or REQUEST scoped. 2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed. <p>** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().</p>			

1538

1539 8.16 @Remotable

1540 The following Java code defines the **@Remotable** annotation:

1541

```
1542 package org.osoa.sca.annotations;
```

1543

```
1544 import static java.lang.annotation.ElementType.TYPE;
```

```
1545 import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
1546 import java.lang.annotation.Retention;
```

```
1547 import java.lang.annotation.Target;
```

1548

1549

```
1550 @Target(TYPE)
```

```
1551 @Retention(RUNTIME)
```

```
1552 public @interface Remotable {
```

1553

```
1554 }
```

1555

1556 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable
1557 service can be published externally as a service and must be translatable into a WSDL portType.

1558 The @Remotable annotation has no attributes.

1559

1560 The following snippet shows the Java interface for a remotable service with its @Remotable
1561 annotation.

```
1562 package services.hello;
```

1563

```
1564 import org.osoa.sca.annotations.*;
```

1565

```
1566 @Remotable
```

```
1567 public interface HelloService {
```

1568

```
1569     String hello(String message);
```

1570

```
1571 }
```

1572 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
1573 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

1574

1575 Complex data types exchanged via remotable service interfaces must be compatible with the
1576 marshalling technology used by the service binding. For example, if the service is going to be
1577 exposed using the standard web service binding, then the parameters must be Service Data
1578 Objects (SDOs) 2.0 or 2.1 [2] or JAXB 2.0 [3] types.

1579 Independent of whether the remotable service is called from outside of the composite that
1580 contains it or from another component in the same composite, the data exchange semantics are
1581 **by-value**.

1582 Implementations of remotable services may modify input data during or after an invocation and
1583 may modify return data after the invocation. If a remotable service is called locally or remotely,
1584 the SCA container is responsible for making sure that no modification of input data or post-
1585 invocation modifications to return data are seen by the caller.

1586

1587 The following snippet shows a remotable Java service interface.

1588

```
1589 package services.hello;
1590
1591 import org.osoa.sca.annotations.*;
1592
1593 @Remotable
1594 public interface HelloService {
1595     String hello(String message);
1596 }
1597
1598 package services.hello;
1599
1600 import org.osoa.sca.annotations.*;
1601
1602 @Service(HelloService.class)
1603 public class HelloServiceImpl implements HelloService {
1604     public String hello(String message) {
1605         ...
1606     }
1607 }
1608
1609 }
```

1610 8.17 @Scope

1611 The following Java code defines the **@Scope** annotation:

1612

```
1613 package org.osoa.sca.annotations;
1614
1615 import static java.lang.annotation.ElementType.TYPE;
1616 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1617 import java.lang.annotation.Retention;
1618 import java.lang.annotation.Target;
1619
1620 @Target(TYPE)
1621 @Retention(RUNTIME)
1622 public @interface Scope {
1623     String value() default "STATELESS";
1624 }
1625
```

1626 The @Scope annotation may only be used on a service's implementation class. It is an error to use
1627 this annotation on an interface.

1628 The @Scope annotation has the following attribute:

- 1629 • **value** – the name of the scope.
1630 The default value is 'STATELESS'. For 'STATELESS' implementations, a different

1631 implementation instance may be used to service each request. Implementation instances
1632 may be newly created or be drawn from a pool of instances.
1633 SCA defines the following scope names, but others can be defined by particular Java-
1634 based implementation types:
1635 STATELESS
1636 REQUEST
1637 COMPOSITE
1638 CONVERSATION

1639 The following snippet shows a sample for a CONVERSATION scoped service implementation:

```
1640 package services.hello;  
1641  
1642 import org.osoa.sca.annotations.*;  
1643  
1644 @Service(HelloService.class)  
1645 @Scope("CONVERSATION")  
1646 public class HelloServiceImpl implements HelloService {  
1647     public String hello(String message) {  
1648         ...  
1649     }  
1650 }  
1651  
1652
```

1653 8.18 @Service

1654 The following Java code defines the **@Service** annotation:

```
1655  
1656 package org.osoa.sca.annotations;  
1657  
1658 import static java.lang.annotation.ElementType.TYPE;  
1659 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1660 import java.lang.annotation.Retention;  
1661 import java.lang.annotation.Target;  
1662  
1663 @Target(TYPE)  
1664 @Retention(RUNTIME)  
1665 public @interface Service {  
1666  
1667     Class<?>[] interfaces() default {};  
1668     Class<?> value() default Void.class;  
1669 }  
1670
```

1671 The @Service annotation is used on a component implementation class to specify the SCA services
1672 offered by the implementation. The class need not be declared as implementing all of the
1673 interfaces implied by the services, but all methods of the service interfaces must be present. A
1674 class used as the implementation of a service is not required to have an @Service annotation. If a
1675 class has no @Service annotation, then the rules determining which services are offered and what
1676 interfaces those services have are determined by the specific implementation type.

1677 The @Service annotation has the following attributes:

- 1678 • **interfaces** – The value is an array of interface or class objects that should be exposed as
1679 services by this component.
- 1680 • **value** – A shortcut for the case when the class provides only a single service interface.

1681 Only one of these attributes should be specified.

1682

1683 A `@Service` annotation with no attributes is meaningless, it is the same as not having the
1684 annotation there at all.

1685 The **service names** of the defined services default to the names of the interfaces or class, without
1686 the package name.

1687 If a Java implementation needs to realize two services with the same interface, then this is
1688 achieved through subclassing of the interface. The subinterface must not add any methods. Both
1689 interfaces are listed in the `@Service` annotation of the Java implementation class.

1690 The following snippet shows an implementation of the `HelloService` marked with the `@Service`
1691 annotation.

1692 `package` services.hello;

1693

1694 `import` org.osoa.sca.annotations.Service;

1695

1696 `@Service`(HelloService.class)

1697 `public class` HelloServiceImpl `implements` HelloService {

1698

1699 `public void` hello(String name) {

1700 `System.out.println`("Hello " + name);

1701 }

1702 }

1703

1704

1705

9 WSDL to Java and Java to WSDL

1706 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL
1707 mapping rules as defined by the JAX-WS specification [7] for generating remotable Java interfaces
1708 from WSDL portTypes and vice versa.

1709 For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a
1710 @WebService annotation on the class, even if it doesn't, and the org.osoa.annotations.OneWay
1711 annotation should be treated as a synonym for javax.jws.OneWay. For the WSDL-to-Java
1712 mapping, the generated @WebService annotation implies that the interface is @Remotable.

1713 For the mapping from Java types to XML schema types SCA supports both the SDO 2.1 [2]
1714 mapping and the JAXB 2.0 [3] mapping. Having a choice of binding technologies is allowed, as
1715 noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is
1716 referenced by the JAX-WS specification.

1717 The JAX-WS mappings are applied with the following restrictions:

- 1718 • No support for holders

1719

1720 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous
1721 model is used.

1722 9.1 JAX-WS Client Asynchronous API for a Synchronous Service

1723 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client
1724 application with a means of invoking that service asynchronously, so that the client can invoke a service
1725 operation and proceed to do other work without waiting for the service operation to complete its
1726 processing. The client application can retrieve the results of the service either through a polling
1727 mechanism or via a callback method which is invoked when the operation completes.

1728 For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional
1729 client-side asynchronous polling and callback methods defined by JAX-WS. For SCA service interfaces
1730 defined using interface.java, the Java interface MUST NOT contain these methods. If these methods are
1731 present, SCA Runtimes MUST NOT include them in the SCA reference interface as defined by the
1732 Assembly specification. These methods are recognized as follows.

1733 For each method M in the interface, if another method P in the interface has

- 1734 a. a method name that is M's method name with the characters "Async" appended, and
- 1735 b. the same parameter signature as M, and
- 1736 c. a return type of Response<R> where R is the return type of M

1737 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

1738 For each method M in the interface, if another method C in the interface has

- 1739 a. a method name that is M's method name with the characters "Async" appended, and
- 1740 b. a parameter signature that is M's parameter signature with an additional final parameter of type
1741 AsyncHandler<R> where R is the return type of M, and
- 1742 c. a return type of Future<?>

1743 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

1744 As an example, an interface may be defined in WSDL as follows:

```
1745 <!-- WSDL extract -->  
1746 <message name="getPrice">  
1747   <part name="ticker" type="xsd:string"/>  
1748 </message>  
1749
```



```
1750 <message name="getPriceResponse">
1751   <part name="price" type="xsd:float"/>
1752 </message>
1753
1754 <portType name="StockQuote">
1755   <operation name="getPrice">
1756     <input message="tns:getPrice"/>
1757     <output message="tns:getPriceResponse"/>
1758   </operation>
1759 </portType>
```

1760

1761 The JAX-WS asynchronous mapping will produce the following Java interface:

```
1762 // asynchronous mapping
1763 @WebService
1764 public interface StockQuote {
1765   float getPrice(String ticker);
1766   Response<Float> getPriceAsync(String ticker);
1767   Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
1768 }
```

1769

1770 For SCA interface definition purposes, this is treated as equivalent to the following:

```
1771 // synchronous mapping
1772 @WebService
1773 public interface StockQuote {
1774   float getPrice(String ticker);
1775 }
```

1776

1777 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above example, if
1778 the client implementation uses the asynchronous form of the interface, the two additional getPriceAsync()
1779 methods can be used for polling and callbacks as defined by the JAX-WS specification.

1780

10 Policy Annotations for Java

1781 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which
1782 influence how implementations, services and references behave at runtime. The policy facilities
1783 are described in [the SCA Policy Framework specification \[5\]](#). In particular, the facilities include
1784 Intents and Policy Sets, where intents express abstract, high-level policy requirements and policy
1785 sets express low-level detailed concrete policies.

1786 Policy metadata can be added to SCA assemblies through the means of declarative statements
1787 placed into Composite documents and into Component Type documents. These annotations are
1788 completely independent of implementation code, allowing policy to be applied during the assembly
1789 and deployment phases of application development.

1790 However, it can be useful and more natural to attach policy metadata directly to the code of
1791 implementations. This is particularly important where the policies concerned are relied on by the
1792 code itself. An example of this from the Security domain is where the implementation code
1793 expects to run under a specific security Role and where any service operations invoked on the
1794 implementation must be authorized to ensure that the client has the correct rights to use the
1795 operations concerned. By annotating the code with appropriate policy metadata, the developer
1796 can rest assured that this metadata is not lost or forgotten during the assembly and deployment
1797 phases.

1798 The SCA Java Common Annotations specification provides a series of annotations which provide
1799 the capability for the developer to attach policy information to Java implementation code. The
1800 annotations concerned first provide general facilities for attaching SCA Intents and Policy Sets to
1801 Java code. Secondly, there are further specific annotations that deal with particular policy intents
1802 for certain policy domains such as Security.

1803 The SCA Java Common Annotations specification supports using [the Common Annotation for Java
1804 Platform specification \(JSR-250\) \[6\]](#). An implication of adopting the common annotation for Java
1805 platform specification is that the SCA Java specification support consistent annotation and Java
1806 class inheritance relationships.

1807

10.1 General Intent Annotations

1809 SCA provides the annotation **@Requires** for the attachment of any intent to a Java class, to a
1810 Java interface or to elements within classes and interfaces such as methods and fields.

1811 The @Requires annotation can attach one or multiple intents in a single statement.

1812 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI
1813 followed by the name of the Intent. The precise form used follows the string representation used
1814 by the `javax.xml.namespace.QName` class, which is as follows:

1815 `"{" + Namespace URI + "}" + intentname`

1816 Intents may be qualified, in which case the string consists of the base intent name, followed by a
1817 ".", followed by the name of the qualifier. There may also be multiple levels of qualification.

1818 This representation is quite verbose, so we expect that reusable String constants will be defined
1819 for the namespace part of this string, as well as for each intent that is used by Java code. SCA
1820 defines constants for intents such as the following:

```
1821 public static final String SCA_PREFIX="{http://docs.oasis-  
1822 open.org/ns/opencsa/sca/200712}";
```

```
1823 public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
```

```
1824 public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
```

1825 Notice that, by convention, qualified intents include the qualifier as part of the name of the
1826 constant, separated by an underscore. These intent constants are defined in the file that defines
1827 an annotation for the intent (annotations for intents, and the formal definition of these constants,
1828 are covered in a following section).

1829 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

1830 An example of the @Requires annotation with 2 qualified intents (from the Security domain)
1831 follows:

1832

```
1833     @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1834

1835 This attaches the intents "confidentiality.message" and "integrity.message".

1836 The following is an example of a reference requiring support for confidentiality:

```
1837     package org.osoa.sca.annotation;
```

1838

```
1839     import static org.osoa.sca.annotation.Confidentiality.*;
```

1840

```
1841     public class Foo {
```

```
1842         @Requires(CONFIDENTIALITY)
```

```
1843         @Reference
```

```
1844         public void setBar(Bar bar) {
```

```
1845             ...
```

```
1846         }
```

```
1847     }
```

1848 Users may also choose to only use constants for the namespace part of the QName, so that they
1849 may add new intents without having to define new constants. In that case, this definition would
1850 instead look like this:

```
1851     package org.osoa.sca.annotation;
```

1852

```
1853     import static org.osoa.sca.Constants.*;
```

1854

```
1855     public class Foo {
```

```
1856         @Requires(SCA_PREFIX+"confidentiality")
```

```
1857         @Reference
```

```
1858         public void setBar(Bar bar) {
```

```
1859             ...
```

```
1860         }
```

```
1861     }
```

1862

1863 The formal syntax for the @Requires annotation follows:

```
1864     @Requires( "qualifiedIntent" | { "qualifiedIntent" [, "qualifiedIntent"] }
```

1865 where

1866 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2

1867

1868 The following shows the formal definition of the @Requires annotation:

1869

```
1870 package org.oesa.sca.annotation;
1871 import static java.lang.annotation.ElementType.TYPE;
1872 import static java.lang.annotation.ElementType.METHOD;
1873 import static java.lang.annotation.ElementType.FIELD;
1874 import static java.lang.annotation.ElementType.PARAMETER;
1875 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1876 import java.lang.annotation.Retention;
1877 import java.lang.annotation.Target;
1878 import java.lang.annotation.Inherited;
```

1879

```
1880 @Inherited
1881 @Retention(RUNTIME)
1882 @Target({TYPE, METHOD, FIELD, PARAMETER})
```

1883

```
1884 public @interface Requires {
1885     String[] value() default "";
1886 }
```

1887 The SCA_NS constant is defined in the Constants interface:

```
1888 package org.oesa.sca;
1889
1890 public interface Constants {
1891     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
1892     String SCA_PREFIX = "{"+SCA_NS+"}";
1893 }
```

1894

1895 10.2 Specific Intent Annotations

1896 In addition to the general intent annotation supplied by the @Requires annotation described
1897 above, it is also possible to have Java annotations that correspond to specific policy intents. SCA
1898 provides a number of these specific intent annotations and it is also possible to create new specific
1899 intent annotations for any intent.

1900 The general form of these specific intent annotations is an annotation with a name derived from
1901 the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an
1902 attribute to the annotation in the form of a string or an array of strings.

1903 For example, the SCA confidentiality intent described in [the section on General Intent Annotations](#)
1904 using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific
1905 @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent
1906 is:

1907 @Integrity

1908 An example of a qualified specific intent for the "authentication" intent is:

1909 @Authentication({"message", "transport"})

1910 This annotation attaches the pair of qualified intents: "authentication.message" and
 1911 "authentication.transport" (the sca: namespace is assumed in this both of these cases –
 1912 "http://docs.oasis-open.org/ns/opencsa/sca/200712").

1913 The general form of specific intent annotations is:

1914 @<Intent>[(qualifiers)]

1915 where Intent is an NCName that denotes a particular type of intent.

1916 Intent ::= NCName

1917 qualifiers ::= "qualifier" | { "qualifier" [, "qualifier"] }

1918 qualifier ::= NCName | NCName/qualifier

1919

10.2.1 How to Create Specific Intent Annotations

1921 SCA identifies annotations that correspond to intents by providing an @Intent annotation which
 1922 must be used in the definition of an intent annotation.

1923 The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the
 1924 String form of the QName of the intent. As part of the intent definition, it is good practice
 1925 (although not required) to also create String constants for the Namespace, the Intent and for
 1926 Qualified versions of the Intent (if defined). These String constants are then available for use with
 1927 the @Requires annotation and it should also be possible to use one or more of them as
 1928 parameters to the @Intent annotation.

1929 Alternatively, the QName of the intent may be specified using separate parameters for the
 1930 targetNamespace and the localPart for example:

```
1931           @Intent(targetNamespace=SCA_NS, localPart="confidentiality").
```

1932 The definition of the @Intent annotation is the following:

```
1933
1934        package org.osoa.sca.annotation;
1935        import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
1936        import static java.lang.annotation.RetentionPolicy.RUNTIME;
1937        import java.lang.annotation.Retention;
1938        import java.lang.annotation.Target;
1939        import java.lang.annotation.Inherited;
1940
1941        @Retention(RUNTIME)
1942        @Target(ANNOTATION_TYPE)
1943        public @interface Intent {
1944           String value() default "";
1945           String targetNamespace() default "";
1946           String localPart() default "";
1947        }
```

1948 When an intent can be qualified, it is good practice for the first attribute of the annotation to be a
1949 string (or an array of strings) which holds one or more qualifiers.

1950 In this case, the attribute's definition should be marked with the @Qualifier annotation. The
1951 @Qualifier tells SCA that the value of the attribute should be treated as a qualifier for the intent
1952 represented by the whole annotation. If more than one qualifier value is specified in an
1953 annotation, it means that multiple qualified forms are required. For example:

```
1954 @Confidentiality({"message", "transport"})
```

1955 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"
1956 are set for the element to which the confidentiality intent is attached.

1957 The following is the definition of the @Qualifier annotation.

1958

```
1959 package org.osoa.sca.annotation;  
1960 import static java.lang.annotation.ElementType.METHOD;  
1961 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1962 import java.lang.annotation.Retention;  
1963 import java.lang.annotation.Target;  
1964 import java.lang.annotation.Inherited;
```

1965

```
1966 @Retention(RetentionPolicy.RUNTIME)
```

```
1967 @Target(ElementType.METHOD)
```

```
1968 public @interface Qualifier {
```

```
1969 }
```

1970

1971 Examples of the use of the @Intent and @Qualifier annotations in the definition of specific intent
1972 annotations are shown in [the section dealing with Security Interaction Policy](#).

1973

1974 10.3 Application of Intent Annotations

1975 The SCA Intent annotations can be applied to the following Java elements:

- 1976 • Java class
- 1977 • Java interface
- 1978 • Method
- 1979 • Field

1980 Where multiple intent annotations (general or specific) are applied to the same Java element, they
1981 are additive in effect. An example of multiple policy annotations being used together follows:

```
1982 @Authentication
```

```
1983 @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1984 In this case, the effective intents are "authentication", "confidentiality.message" and
1985 "integrity.message".

1986 If an annotation is specified at both the class/interface level and the method or field level, then
1987 the method or field level annotation completely overrides the class level annotation of the same
1988 type.

1989 The intent annotation can be applied either to classes or to class methods when adding annotated
1990 policy on SCA services. Applying an intent to the setter method in a reference injection approach
1991 allows intents to be defined at references.

1992 **10.3.1 Inheritance And Annotation**

1993 The inheritance rules for annotations are consistent with the common annotation specification, JSR
1994 250.

1995 The following example shows the inheritance relations of intents on classes, operations, and super
1996 classes.

```
1997  
1998 package services.hello;  
1999 import org.osoa.sca.annotations.Remotable;  
2000 import org.osoa.sca.annotations.Integrity;  
2001 import org.osoa.sca.annotations.Authentication;  
2002  
2003 @Integrity("transport")  
2004 @Authentication  
2005 public class HelloService {  
2006     @Integrity  
2007     @Authentication("message")  
2008     public String hello(String message) {...}  
2009  
2010     @Integrity  
2011     @Authentication("transport")  
2012     public String helloThere() {...}  
2013 }  
2014  
2015 package services.hello;  
2016 import org.osoa.sca.annotations.Remotable;  
2017 import org.osoa.sca.annotations.Confidentiality;  
2018 import org.osoa.sca.annotations.Authentication;  
2019  
2020 @Confidentiality("message")  
2021 public class HelloChildService extends HelloService {  
2022     @Confidentiality("transport")  
2023     public String hello(String message) {...}  
2024     @Authentication  
2025     String helloWorld() {...}  
2026 }
```

2027 Example 2a. Usage example of annotated policy and inheritance.

2028

2029 The effective intent annotation on the helloWorld method is Integrity("transport"),
2030 @Authentication, and @Confidentiality("message").

2031 The effective intent annotation on the hello method of the HelloChildService is
 2032 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),
 2033 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity
 2034 and @Authentication("transport"), the same as in HelloService class.
 2035 The effective intent annotation on the hello method of the HelloService is @Integrity and
 2036 @Authentication("message")
 2037
 2038 The listing below contains the equivalent declarative security interaction policy of the HelloService
 2039 and HelloChildService implementation corresponding to the Java interfaces and classes shown in
 2040 Example 2a.

```

2041
2042 <?xml version="1.0" encoding="ASCII"?>
2043
2044 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2045           name="HelloServiceComposite" >
2046   <service name="HelloService" requires="integrity/transport
2047           authentication">
2048     ...
2049   </service>
2050   <service name="HelloChildService" requires="integrity/transport
2051           authentication confidentiality/message">
2052     ...
2053   </service>
2054   ...
2055
2056   <component name="HelloServiceComponent">*
2057     <implementation.java class="services.hello.HelloService" />
2058       <operation name="hello" requires="integrity
2059           authentication/message" />
2060       <operation name="helloThere"
2061 requires="integrity
2062           authentication/transport" />
2063     </component>
2064   <component name="HelloChildServiceComponent">*
2065     <implementation.java
2066 class="services.hello.HelloChildService" />
2067       <operation name="hello"
2068 requires="confidentiality/transport" />
2069       <operation name="helloThere" requires=" integrity/transport
2070           authentication" />
2071       <operation name="helloWorld" requires="authentication" />
2072     </component>
2073   ...
2074   ...
2075
2076 </composite>
  
```

2077 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.
 2078
 2079

2080 10.4 Relationship of Declarative And Annotated Intents

2081 Annotated intents on a Java class cannot be overridden by declarative intents either in a
 2082 composite document which uses the class as an implementation or by statements in a component

2083 Type document associated with the class. This rule follows the general rule for intents that they
2084 represent fundamental requirements of an implementation.

2085 An unqualified version of an intent expressed through an annotation in the Java class may be
2086 qualified by a declarative intent in a using composite document.

2087

2088 10.5 Policy Set Annotations

2089 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for
2090 example, a concrete policy is the specific encryption algorithm to use when encrypting messages
2091 when using a specific communication protocol to link a reference to a service).

2092 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.
2093 The PolicySets annotation either takes the QName of a single policy set as a string or the name of
2094 two or more policy sets as an array of strings:
2095

```
2096     @PolicySets( "<policy set QName>" |  
2097                 { "<policy set QName>" [, "<policy set QName>"] })
```

2098

2099 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

2100 An example of the @PolicySets annotation:

2101

```
2102     @Reference(name="helloService", required=true)  
2103     @PolicySets({ MY_NS + "WS_Encryption_Policy",  
2104                 MY_NS + "WS_Authentication_Policy" })  
2105     public setHelloService(HelloService service) {  
2106         . . .  
2107     }
```

2108 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
2109 using the namespace defined for the constant MY_NS.

2110 PolicySets must satisfy intents expressed for the implementation when both are present, according
2111 to the rules defined in [the Policy Framework specification \[5\]](#).

2112 The SCA Policy Set annotation can be applied to the following Java elements:

- 2113 • Java class
- 2114 • Java interface
- 2115 • Method
- 2116 • Field

2117

2118 10.6 Security Policy Annotations

2119 This section introduces annotations for SCA's security intents, as defined in [the SCA Policy](#)
2120 [Framework specification \[5\]](#).

2121

2122 10.6.1 Security Interaction Policy

2123 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply
2124 to the operation of services and references of an implementation:

- 2125 • @Integrity
- 2126 • @Confidentiality
- 2127 • @Authentication

2128 All three of these intents have the same pair of Qualifiers:

- 2129 • message
- 2130 • transport

2131 The following snippets shows the @Integrity, @Confidentiality and @Authentication annotations:

```

2132 package org.oesa.sca.annotation;
2133
2134 import java.lang.annotation.*;
2135 import static org.oesa.sca.Constants.SCA_NS;
2136
2137 @Inherited
2138 @Retention(RetentionPolicy.RUNTIME)
2139 @Target({ElementType.TYPE, ElementType.METHOD,
2140          ElementType.FIELD, ElementType.PARAMETER})
2141 @Intent(Integrity.INTEGRITY)
2142 public @interface Integrity {
2143     String INTEGRITY = SCA_NS+"integrity";
2144     String INTEGRITY_MESSAGE = INTEGRITY+".message";
2145     String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
2146     @Qualifier
2147     String[] value() default "";
2148 }
2149
2150
2151 package org.oesa.sca.annotation;
2152
2153 import java.lang.annotation.*;
2154 import static org.oesa.sca.Constants.SCA_NS;
2155
2156 @Inherited
2157 @Retention(RetentionPolicy.RUNTIME)
2158 @Target({ElementType.TYPE, ElementType.METHOD,
2159          ElementType.FIELD, ElementType.PARAMETER})
2160 @Intent(Confidentiality.CONFIDENTIALITY)
2161 public @interface Confidentiality {
2162     String CONFIDENTIALITY = SCA_NS+"confidentiality";
2163     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY+".message";

```

```

2164     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY+".transport";
2165     @Qualifier
2166     String[] value() default "";
2167 }
2168
2169
2170 package org.osoa.sca.annotation;
2171
2172 import java.lang.annotation.*;
2173 import static org.osoa.sca.Constants.SCA_NS;
2174
2175 @Inherited
2176 @Retention(RetentionPolicy.RUNTIME)
2177 @Target({ElementType.TYPE, ElementType.METHOD,
2178         ElementType.FIELD, ElementType.PARAMETER})
2179 @Intent(Authentication.AUTHENTICATION)
2180 public @interface Authentication {
2181     String AUTHENTICATION = SCA_NS+"authentication";
2182     String AUTHENTICATION_MESSAGE = AUTHENTICATION+".message";
2183     String AUTHENTICATION_TRANSPORT = AUTHENTICATION+".transport";
2184     @Qualifier
2185     String[] value() default "";
2186 }

```

2187
2188

The following example shows an example of applying an intent to the setter method used to inject a reference. Accessing the hello operation of the referenced HelloService requires both "integrity.message" and "authentication.message" intents to be honored.

```

2192
2193 //Interface for HelloService
2194 public interface service.hello.HelloService {
2195     String hello(String helloMsg);
2196 }
2197
2198 // Interface for ClientService
2199 public interface service.client.ClientService {
2200     public void clientMethod();
2201 }
2202
2203 // Implementation class for ClientService
2204 package services.client;

```

```

2205
2206     import services.hello.HelloService;
2207
2208     import org.osoa.sca.annotations.*;
2209
2210     @Service(ClientService.class)
2211     public class ClientServiceImpl implements ClientService {
2212
2213
2214         private HelloService helloService;
2215
2216         @Reference(name="helloService", required=true)
2217         @Integrity("message")
2218         @Authentication("message")
2219         public void setHelloService(HelloService service) {
2220             helloService = service;
2221         }
2222
2223         public void clientMethod() {
2224             String result = helloService.hello("Hello World!");
2225             ...
2226         }
2227     }
2228

```

2229 Example 1. Usage of annotated intents on a reference.

2230

2231 10.6.2 Security Implementation Policy

2232 SCA defines a number of security policy annotations that apply as policies to implementations
2233 themselves. These annotations mostly have to do with authorization and security identity. The
2234 following authorization and security identity annotations (as defined in JSR 250) are supported:

- 2235 • RunAs
- 2236 Takes as a parameter a string which is the name of a Security role.
2237 eg. @RunAs("Manager")
- 2238
- 2239 • Code marked with this annotation will execute with the Security permissions of the
2240 identified role.
- 2241 • RolesAllowed
- 2242 Takes as a parameter a single string or an array of strings which represent one or more
2243 role names. When present, the implementation can only be accessed by principals whose
2244 role corresponds to one of the role names listed in the @roles attribute. How role names
2245 are mapped to security principals is implementation dependent (SCA does not define this).
2246 eg. @RolesAllowed({"Manager", "Employee"})
- 2247
- 2248 • PermitAll
- 2249 No parameters. When present, grants access to all roles.
- 2250

- 2251 • DenyAll
- 2252
- 2253 No parameters. When present, denies access to all roles.
- 2254 • DeclareRoles
- 2255 Takes as a parameter a string or an array of strings which identify one or more role names
- 2256 that form the set of roles used by the implementation.
- 2257 eg. `@DeclareRoles({"Manager", "Employee", "Customer"})`
- 2258 (all these are declared in the Java package `javax.annotation.security`)
- 2259 For a full explanation of these intents, see [the Policy Framework specification \[5\]](#).

2260 10.6.2.1 Annotated Implementation Policy Example

2261 The following is an example showing annotated security implementation policy:

```
2262
2263 package services.account;
2264 @Remotable
2265 public interface AccountService {
2266     AccountReport getAccountReport(String customerID);
2267 }
```

2268

2269 The following is a full listing of the `AccountServiceImpl` class, showing the `Service` it implements,

2270 plus the service references it makes and the settable properties that it has, along with a set of

2271 implementation policy annotations:

```
2272
2273 package services.account;
2274 import java.util.List;
2275 import commonj.sdo.DataFactory;
2276 import org.oesa.sca.annotations.Property;
2277 import org.oesa.sca.annotations.Reference;
2278 import org.oesa.sca.annotations.RolesAllowed;
2279 import org.oesa.sca.annotations.RunAs;
2280 import org.oesa.sca.annotations.PermitAll;
2281 import services.accountdata.AccountDataService;
2282 import services.accountdata.CheckingAccount;
2283 import services.accountdata.SavingsAccount;
2284 import services.accountdata.StockAccount;
2285 import services.stockquote.StockQuoteService;
2286 @RolesAllowed("customers")
2287 @RunAs("accountants" )
2288 public class AccountServiceImpl implements AccountService {
2289
2290     @Property
2291     protected String currency = "USD";
2292
2293     @Reference
2294     protected AccountDataService accountDataService;
```

```

2295     @Reference
2296     protected StockQuoteService stockQuoteService;
2297
2298     @RolesAllowed({"customers", "accountants"})
2299     public AccountReport getAccountReport(String customerID) {
2300
2301         DataFactory dataFactory = DataFactory.INSTANCE;
2302         AccountReport accountReport =
2303             (AccountReport)dataFactory.create(AccountReport.class);
2304         List accountSummaries = accountReport.getAccountSummaries();
2305
2306         CheckingAccount checkingAccount =
2307             accountDataService.getCheckingAccount(customerID);
2308         AccountSummary checkingAccountSummary =
2309             (AccountSummary)dataFactory.create(AccountSummary.class);
2310
2311         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
2312         );
2313         checkingAccountSummary.setAccountType("checking");
2314         checkingAccountSummary.setBalance(fromUSDollarToCurrency
2315             (checkingAccount.getBalance()));
2316         accountSummaries.add(checkingAccountSummary);
2317
2318         SavingsAccount savingsAccount =
2319             accountDataService.getSavingsAccount(customerID);
2320         AccountSummary savingsAccountSummary =
2321             (AccountSummary)dataFactory.create(AccountSummary.class);
2322
2323         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
2324         savingsAccountSummary.setAccountType("savings");
2325         savingsAccountSummary.setBalance(fromUSDollarToCurrency
2326             (savingsAccount.getBalance()));
2327         accountSummaries.add(savingsAccountSummary);
2328
2329         StockAccount stockAccount =
2330         accountDataService.getStockAccount(customerID);
2331         AccountSummary stockAccountSummary =
2332             (AccountSummary)dataFactory.create(AccountSummary.class);
2333         stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
2334         stockAccountSummary.setAccountType("stock");
2335         float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
2336             stockAccount.getQuantity();
2337         stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
2338         accountSummaries.add(stockAccountSummary);

```

```

2339
2340     return accountReport;
2341 }
2342
2343 @PermitAll
2344 public float fromUSDollarToCurrency(float value) {
2345
2346     if (currency.equals("USD")) return value; else
2347     if (currency.equals("EURO")) return value * 0.8f; else
2348     return 0.0f;
2349 }
2350 }

```

2351 Example 3. Usage of annotated security implementation policy for the java language.

2352 In this example, the implementation class as a whole is marked:

- 2353 • @RolesAllowed("customers") - indicating that customers have access to the
- 2354 implementation as a whole
- 2355 • @RunAs("accountants") – indicating that the code in the implementation runs with the
- 2356 permissions of accountants

2357 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),
2358 which indicates that this method can be called by both customers and accountants.

2359 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method
2360 can be called by any role.

2361

2362 **A. Acknowledgements**

2363 The following individuals have participated in the creation of this specification and are gratefully
2364 acknowledged:

2365 **Participants:**

2366 [Participant Name, Affiliation | Individual Member]

2367 [Participant Name, Affiliation | Individual Member]

2368

B. Non-Normative Text

2370

C. Revision History

2371 [optional; should not be included in OASIS Standards]

2372

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-02-28	Anish Karmarkar	Applied resolution of issues: 4, 11, and 26
3	2008-04-17	Mike Edwards	Ed changes
4	2008-05-27	Anish Karmarkar David Booz Mark Combella	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	<ul style="list-style-type: none"> * Applied resolution of issue 9 (it was applied before, not sure by whom, but it was applied incorrectly) * Applied resolution of issue 12, 22, 23, 29, 31, 35, 36, 37, 44, 45 * Note that issue 33 was applied, but not noted, in a previous version * Replaced the osoa.org NS with the oasis-open.org NS
WD05	2008-10-03	Anish Karmarkar	<ul style="list-style-type: none"> * Fixed the resolution of issue 37 but re-adding the sentence: "However, the @... annotation must be used in order to inject a property onto a non-public field. -- in the @Property and @Reference section * resolution of issue 9 was applied incorrectly. Fixed that -- removed the requirement for throwing an exception on ComponentContext.getServiceReferences() when multiplicity of references > 1 * minor ed changes

2373

2374