



Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1

Committee Draft 04

06 February 2010

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd04.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd04.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd04.pdf> (Authoritative)

Previous Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd03.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd03.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd03.pdf> (Authoritative)

Latest Version:

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

Technical Committee:

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

Chair(s):

David Booz,	IBM
Mark Combellack,	Avaya

Editor(s):

David Booz,	IBM
Mark Combellack,	Avaya
Mike Edwards,	IBM
Anish Karmarkar,	Oracle

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

Compiled Java API:

<http://docs.oasis-open.org/opencsa/sca-j/sca-caa-apis-1.1-CD04.jar>

Downloadable Javadoc:

<http://docs.oasis-open.org/opencsa/sca-j/sca-j-caa-javadoc-1.1-CD04.zip>

Hosted Javadoc:

<http://docs.oasis-open.org/opencsa/sca-j/javadoc/index.html>

Declared XML Namespace(s):

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

Java Artifacts:

<http://docs.oasis-open.org/opencsa/sca-j/sca-j-common-annotations-and-apis-1.1-cd04.zip>

Abstract:

The SCA-J Common Annotations and APIs specification defines a Java syntax for programming concepts defined in the SCA Assembly Model Specification. It specifies a set of APIs and annotations that can be used by Java-based artifacts described by other SCA specifications such as [the POJO Component Implementation Specification \[JAVA_CI\]](#).

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous 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 other Java-based SCA specifications can choose 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 might 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, 2010. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a 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	10
2.2.2	Composite Scope	11
2.3	@AllowsPassByReference.....	11
2.3.1	Marking Services as “allows pass by reference”	12
2.3.2	Marking References as “allows pass by reference”	12
2.3.3	Applying “allows pass by reference” to Service Proxies	12
2.3.4	Using “allows pass by reference” to Optimize Remotable Calls	13
3	Interface.....	14
3.1	Java Interface Element – <interface.java>	14
3.2	@Remotable.....	15
3.3	@Callback	15
3.4	@AsyncInvocation.....	15
3.5	SCA Java Annotations for Interface Classes.....	16
3.6	Compatibility of Java Interfaces.....	16
4	SCA Component Implementation Lifecycle.....	17
4.1	Overview of SCA Component Implementation Lifecycle.....	17
4.2	SCA Component Implementation Lifecycle State Diagram.....	17
4.2.1	Constructing State.....	18
4.2.2	Injecting State.....	18
4.2.3	Initializing State	19
4.2.4	Running State.....	19
4.2.5	Destroying State.....	19
4.2.6	Terminated State.....	20
5	Client API.....	21
5.1	Accessing Services from an SCA Component	21
5.1.1	Using the Component Context API	21
5.2	Accessing Services from non-SCA Component Implementations	21
5.2.1	SCAClientFactory Interface and Related Classes	21
6	Error Handling	23
7	Asynchronous Programming.....	24
7.1	@OneWay	24

7.2	Callbacks	24
7.2.1	Using Callbacks.....	24
7.2.2	Callback Instance Management.....	26
7.2.3	Callback Injection	26
7.2.4	Implementing Multiple Bidirectional Interfaces.....	26
7.2.5	Accessing Callbacks	27
7.3	Asynchronous handling of Long Running Service Operations	28
7.4	SCA Asynchronous Service Interface	28
8	Policy Annotations for Java	31
8.1	General Intent Annotations.....	31
8.2	Specific Intent Annotations	33
8.2.1	How to Create Specific Intent Annotations.....	34
8.3	Application of Intent Annotations	34
8.3.1	Intent Annotation Examples	35
8.3.2	Inheritance and Annotation	37
8.4	Relationship of Declarative and Annotated Intents	38
8.5	Policy Set Annotations.....	38
8.6	Security Policy Annotations	39
8.7	Transaction Policy Annotations	40
9	Java API	42
9.1	Component Context.....	42
9.2	Request Context	47
9.3	ServiceReference Interface	49
9.4	ResponseDispatch interface.....	50
9.5	ServiceRuntimeException.....	51
9.6	ServiceUnavailableException	52
9.7	InvalidServiceException.....	52
9.8	Constants.....	52
9.9	SCAClientFactory Class	53
9.10	SCAClientFactoryFinder Interface.....	56
9.11	SCAClientFactoryFinderImpl Class	57
9.12	NoSuchDomainException.....	58
9.13	NoSuchServiceException	58
10	Java Annotations	59
10.1	@AllowsPassByReference.....	59
10.2	@AsyncFault	60
10.3	@AsyncInvocation	61
10.4	@Authentication	61
10.5	@Authorization	62
10.6	@Callback	62
10.7	@ComponentName	64
10.8	@Confidentiality.....	64
10.9	@Constructor.....	65
10.10	@Context.....	66
10.11	@Destroy.....	67

10.12 @EagerInit.....	67
10.13 @Init	68
10.14 @Integrity	68
10.15 @Intent	69
10.16 @ManagedSharedTransaction.....	70
10.17 @ManagedTransaction	70
10.18 @MutualAuthentication	71
10.19 @NoManagedTransaction.....	72
10.20 @OneWay	72
10.21 @PolicySets	73
10.22 @Property.....	73
10.23 @Qualifier.....	75
10.24 @Reference.....	75
10.24.1 Reinjection.....	78
10.25 @Remotable.....	80
10.26 @Requires.....	81
10.27 @Scope	82
10.28 @Service	83
11 WSDL to Java and Java to WSDL	85
11.1 JAX-WS Annotations and SCA Interfaces.....	85
11.2 JAX-WS Client Asynchronous API for a Synchronous Service.....	90
11.3 Treatment of SCA Asynchronous Service API	91
12 Conformance.....	92
12.1 SCA Java XML Document.....	92
12.2 SCA Java Class.....	92
12.3 SCA Runtime.....	92
A. XML Schema: sca-interface-java-1.1.xsd	93
B. Java Classes and Interfaces	94
B.1 SCAClient Classes and Interfaces	94
B.1.1 SCAClientFactory Class	94
B.1.2 SCAClientFactoryFinder interface	96
B.1.3 SCAClientFactoryFinderImpl class	97
B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?.....	102
C. Conformance Items	103
D. Acknowledgements	117
E. Revision History.....	119

1 Introduction

The SCA-J Common Annotations and APIs specification defines a Java syntax for programming concepts defined in the SCA Assembly Model Specification [ASSEMBLY]. It specifies a set of APIs and annotations that can be used by SCA Java-based 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 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

The goal of defining the annotations and APIs in this specification is to promote consistency and reduce duplication across the various SCA Java-based specifications. The annotations and APIs defined in this specification are designed to be used by other SCA Java-based specifications in either a partial or complete fashion.

1.1 Terminology

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

1.2 Normative References

- [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.
- [ASSEMBLY] OASIS, Committee Draft 05, “SCA Assembly Model Specification Version 1.1”, January 2010.
<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd05.pdf>
- [JAVA_CI] OASIS, Committee Draft 02, “SCA POJO Component Implementation Specification Version 1.1”, February 2010.
<http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-cd02.pdf>
- [SDO] SDO 2.1 Specification,
<http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf>
- [JAX-B] JAXB 2.1 Specification,
<http://www.jcp.org/en/jsr/detail?id=222>
- [WSDL] WSDL Specification,
WSDL 1.1: <http://www.w3.org/TR/wSDL>,
- [POLICY] OASIS, Committee Draft 02, “SCA Policy Framework Version 1.1”, February 2009.
<http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd02.pdf>
- [JSR-250] Common Annotations for the Java Platform specification (JSR-250),
<http://www.jcp.org/en/jsr/detail?id=250>
- [JAX-WS] JAX-WS 2.1 Specification (JSR-224),
<http://www.jcp.org/en/jsr/detail?id=224>
- [JAVABEANS] JavaBeans 1.01 Specification,
<http://java.sun.com/javase/technologies/desktop/javabeans/api/>

46 **[JAAS]** Java Authentication and Authorization Service Reference Guide
47 [http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.](http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html)
48 [html](http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html)

49 **1.3 Non-Normative References**

50 **[EBNF-Syntax]** Extended BNF syntax format used for formal grammar of constructs
51 <http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation>

2 Implementation Metadata

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

2.1 Service Metadata

2.1.1 @Service

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

- As a Java interface
- As a Java class
- As a Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType (Java interfaces generated from WSDL portTypes are always **remotable**)

2.1.2 Java Semantics of a Remotable Service

A **remotable service** is defined using the @Remotable annotation on the Java interface or Java class that defines the service, or on a service reference. Remotable services are intended to be used for **coarse grained** services, and the parameters are passed **by-value**. **Remotable Services MUST NOT make use of method overloading.** [JCA20001]

Snippet 2-1 shows an example of a Java interface for a remotable service:

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

Snippet 2-1: Remotable Java Interface

2.1.3 Java Semantics of a Local Service

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

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

Snippet 2-2 shows an example of a Java interface for a local service:

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

Snippet 2-2: Local Java Interface

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

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

92 2.1.4 @Reference

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

95 2.1.5 @Property

96 Implementations can be configured with data values through the use of properties, as defined in [the SCA
97 Assembly Model specification \[ASSEMBLY\]](#). The **@Property** annotation is used to define an SCA
98 property.

99 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

106 This specification defines two scopes:

- 107 • STATELESS
- 108 • COMPOSITE

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

111 An implementation type can allow component implementations to declare **lifecycle methods** that are
112 called when an implementation is instantiated or the scope is expired.

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

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

116 Note that only no-argument methods with a void return type can be annotated as lifecycle methods.

117 Snippet 2-3 is an example showing a fragment of a service implementation annotated with lifecycle
118 methods:

119

```
120 @Init  
121 public void start() {  
122     ...  
123 }  
124  
125 @Destroy  
126 public void stop() {  
127     ...  
128 }
```

129 *Snippet 2-3: Java Component Implementation with Lifecycle Methods*

130

131 The following sections specify the two standard scopes which a Java-based implementation type can
132 support.

133 2.2.1 Stateless Scope

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

136 The concurrency model for the stateless scope is single threaded. This means that **the SCA runtime
137 MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one**

138 thread at any one time. [JCA20002] In addition, within the SCA lifecycle of a stateless scoped
139 implementation instance, the SCA runtime MUST only make a single invocation of one business method.
140 [JCA20003] Note that the SCA lifecycle might not correspond to the Java object lifecycle due to runtime
141 techniques such as pooling.

142 2.2.2 Composite Scope

143 The meaning of "composite scope" is defined in relation to the composite containing the component.

144 It is important to distinguish between different uses of a composite, where these uses affect the numbers
145 of instances of components within the composite. There are 2 cases:

- 146 a) Where the composite containing the component using the Java implementation is the SCA Domain
147 (i.e. a deployment composite declares the component using the implementation)
- 148 b) Where the composite containing the component using the Java implementation is itself used as the
149 implementation of a higher level component (any level of nesting is possible, but the component is
150 NOT at the Domain level)

151 Where an implementation is used by a "domain level component", and the implementation is marked
152 "Composite" scope, the SCA runtime MUST ensure that all consumers of the component appear to be
153 interacting with a single runtime instance of the implementation. [JCA20004]

154 Where an implementation is marked "Composite" scope and it is used by a component that is nested
155 inside a composite that is used as the implementation of a higher level component, the SCA runtime
156 MUST ensure that all consumers of the component appear to be interacting with a single runtime instance
157 of the implementation. There can be multiple instances of the higher level component, each running on
158 different nodes in a distributed SCA runtime. [JCA20008]

159 The SCA runtime can exploit shared state technology in combination with other well known high
160 availability techniques to provide the appearance of a single runtime instance for consumers of composite
161 scoped components.

162 The lifetime of the containing composite is defined as the time it becomes active in the runtime to the time
163 it is deactivated, either normally or abnormally.

164 When the implementation class is marked for eager initialization, the SCA runtime MUST create a
165 composite scoped instance when its containing component is started. [JCA20005] If a method of an
166 implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when
167 the implementation instance is created. [JCA20006]

168 The concurrency model for the composite scope is multi-threaded. This means that the SCA runtime MAY
169 run multiple threads in a single composite scoped implementation instance object and the SCA runtime
170 MUST NOT perform any synchronization. [JCA20007]

171 2.3 @AllowsPassByReference

172 Calls to remotable services (see [section "Java Semantics of a Remotable Service"](#)) have by-value
173 semantics. This means that input parameters passed to the service can be modified by the service
174 without these modifications being visible to the client. Similarly, the return value or exception from the
175 service can be modified by the client without these modifications being visible to the service
176 implementation. For remote calls (either cross-machine or cross-process), these semantics are a
177 consequence of marshalling input parameters, return values and exceptions "on the wire" and
178 unmarshalling them "off the wire" which results in physical copies being made. For local method calls
179 within the same JVM, Java language calling semantics are by-reference and therefore do not provide the
180 correct by-value semantics for SCA remotable interfaces. To compensate for this, the SCA runtime can
181 intervene in these calls to provide by-value semantics by making copies of any mutable objects passed.

182 The cost of such copying can be very high relative to the cost of making a local call, especially if the data
183 being passed is large. Also, in many cases this copying is not needed if the implementation observes
184 certain conventions for how input parameters, return values and exceptions are used. The
185 @AllowsPassByReference annotation allows service method implementations and client references to be
186 marked as "allows pass by reference" to indicate that they use input parameters, return values and

187 exceptions in a manner that allows the SCA runtime to avoid the cost of copying mutable objects when a
188 remotable service is called locally within the same JVM.

189 **2.3.1 Marking Services as “allows pass by reference”**

190 Marking a service method implementation as “allows pass by reference” asserts that the method
191 implementation observes the following restrictions:

- 192 • Method execution will not modify any input parameter before the method returns.
- 193 • The service implementation will not retain a reference to any mutable input parameter, mutable return
194 value or mutable exception after the method returns.
- 195 • The method will observe “allows pass by reference” client semantics (see section 2.3.2) for any
196 callbacks that it makes.

197 See [section “@AllowsPassByReference”](#) for details of how the @AllowsPassByReference annotation is
198 used to mark a service method implementation as “allows pass by reference”.

199 **2.3.2 Marking References as “allows pass by reference”**

200 Marking a client reference as “allows pass by reference” asserts that method calls through the reference
201 observe the following restrictions:

- 202 • The client implementation will not modify any of the method’s input parameters before the method
203 returns. Such modifications might occur in callbacks or separate client threads.
- 204 • If the method is one-way, the client implementation will not modify any of the method’s input
205 parameters at any time after calling the method. This is because one-way method calls return
206 immediately without waiting for the service method to complete.

207 See [section “Applying “allows pass by reference” to Service Proxies”](#) for details of how the
208 @AllowsPassByReference annotation is used to mark a client reference as “allows pass by reference”.

209 **2.3.3 Applying “allows pass by reference” to Service Proxies**

210 Service method calls are made by clients using service proxies, which can be obtained by injection into
211 client references or by making API calls. A service proxy is marked as “allows pass by reference” if and
212 only if any of the following applies:

- 213 • It is injected into a reference or callback reference that is marked “allows pass by reference”.
- 214 • It is obtained by calling `ComponentContext.getService()` or `ComponentContext.getServices()` with the
215 name of a reference that is marked “allows pass by reference”.
- 216 • It is obtained by calling `RequestContext.getCallback()` from a service implementation that is marked
217 “allows pass by reference”.
- 218 • It is obtained by calling `ServiceReference.getService()` on a service reference that is marked “allows
219 pass by reference”.

220 A service reference for a remotable service call is marked “allows pass by reference” if and only if any of
221 the following applies:

- 222 • It is injected into a reference or callback reference that is marked “allows pass by reference”.
- 223 • It is obtained by calling `ComponentContext.getServiceReference()` or
224 `ComponentContext.getServiceReferences()` with the name of a reference that is marked “allows pass
225 by reference”.
- 226 • It is obtained by calling `RequestContext.getCallbackReference()` from a service implementation that is
227 marked “allows pass by reference”.
- 228 • It is obtained by calling `ComponentContext.cast()` on a proxy that is marked “allows pass by
229 reference”.

230 **2.3.4 Using “allows pass by reference” to Optimize Remotable Calls**

231 The SCA runtime MAY use by-reference semantics when passing input parameters, return values or
232 exceptions on calls to remotable services within the same JVM if both the service method implementation
233 and the service proxy used by the client are marked “allows pass by reference”. [JCA20009]

234 The SCA runtime MUST use by-value semantics when passing input parameters, return values and
235 exceptions on calls to remotable services within the same JVM if the service method implementation is
236 not marked “allows pass by reference” or the service proxy used by the client is not marked “allows pass
237 by reference”. [JCA20010]

238 3 Interface

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

240 3.1 Java Interface Element – <interface.java>

241 The Java interface element is used in SCA Documents in places where an interface is declared in terms
242 of a Java interface class. The Java interface element identifies the Java interface class and can also
243 identify a callback interface, where the first Java interface represents the forward (service) call interface
244 and the second interface represents the interface used to call back from the service to the client.

245 It is possible that the Java interface class referenced by the <interface.java/> element contains one or
246 more annotations defined by the JAX-WS specification [JAX-WS]. These annotations can affect the
247 interpretation of the <interface.java/> element. In the most extreme case, the annotations cause the
248 replacement of the <interface.java/> element with an <interface.wsdl/> element. The relevant JAX-WS
249 annotations and their effects on the <interface.java/> element are described in the section "[JAX-WS
250 Annotations and SCA Interfaces](#)".

251 **The interface.java element MUST conform to the schema defined in the sca-interface-java.xsd schema.**
252 **[JCA30004]**

253 Snippet 3-1 is the pseudo-schema for the interface.java element

254

```
255 <interface.java interface="NCName" callbackInterface="NCName"?  
256     requires="list of xs:QName"?  
257     policySets="list of xs:QName"?  
258     remotable="boolean"?/>
```

259 *Snippet 3-1: interface.java Pseudo-Schema*

260

261 The interface.java element has the attributes:

- 262 • **interface : NCName (1..1)** – the Java interface class to use for the service interface. **The value of the**
263 **@interface attribute MUST be the fully qualified name of the Java interface class [JCA30001]**
264 **If the identified class is annotated with either the JAX-WS @WebService or @WebServiceProvider**
265 **annotations and the annotation has a non-empty wsdlLocation property, then the SCA Runtime**
266 **MUST act as if an <interface.wsdl/> element is present instead of the <interface.java/> element, with**
267 **an @interface attribute identifying the portType mapped from the Java interface class and containing**
268 **@requires and @policySets attribute values equal to the @requires and @policySets attribute values**
269 **of the <interface.java/> element. [JCA30010]**
- 270 • **callbackInterface : NCName (0..1)** – the Java interface class to use for the callback interface. **The**
271 **value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used**
272 **for callbacks [JCA30002]**
- 273 • **requires : QName (0..1)** – a list of policy intents. See the [Policy Framework specification \[POLICY\]](#)
274 for a description of this attribute
- 275 • **policySets : QName (0..1)** – a list of policy sets. See the [Policy Framework specification \[POLICY\]](#)
276 for a description of this attribute.
- 277 • **remotable : boolean (0..1)** – indicates whether or not the interface is remotable. A value of “true”
278 means the interface is remotable and a value of “false” means it is not. This attribute does not have a
279 default value. If it is not specified then the remotability is determined by the presence or absence of
280 the @Remotable annotation on the interface class. The @remotable attribute applies to both the
281 interface and any optional callbackInterface. The @remotable attribute is intended as an alternative
282 to using the @Remotable annotation on the interface class. **The value of the @remotable attribute**

283 on the <interface.java/> element does not override the presence of a @Remotable annotation on the
284 interface class and so if the interface class contains a @Remotable annotation and the @remotable
285 attribute has a value of "false", then the SCA Runtime MUST raise an error and MUST NOT run the
286 component concerned. [JCA30005]

287

288 Snippet 3-2 shows an example of the Java interface element:

289

```
290 <interface.java interface="services.stockquote.StockQuoteService "  
291     callbackInterface="services.stockquote.StockQuoteServiceCallback" />
```

292 *Snippet 3-2 Example interface.java Element*

293

294 Here, the Java interface is defined in the Java class file
295 *./services/stockquote/StockQuoteService.class*, where the root directory is defined by the contribution
296 in which the interface exists. Similarly, the callback interface is defined in the Java class file
297 *./services/stockquote/StockQuoteServiceCallback.class*.

298 Note that the Java interface class identified by the @interface attribute can contain a Java @Callback
299 annotation which identifies a callback interface. If this is the case, then it is not necessary to provide the
300 @callbackInterface attribute. However, if the Java interface class identified by the @interface attribute
301 does contain a Java @Callback annotation, then the Java interface class identified by the
302 @callbackInterface attribute MUST be the same interface class. [JCA30003]

303 For the Java interface type system, parameters and return types of the service methods are described
304 using Java classes or simple Java types. It is recommended that the Java Classes used conform to the
305 requirements of either JAXB [JAX-B] or of Service Data Objects [SDO] because of their integration with
306 XML technologies.

307 3.2 @Remotable

308 The @Remotable annotation on a Java interface, a service implementation class, or a service reference
309 denotes an interface or class that is designed to be used for remote communication. Remotable
310 interfaces are intended to be used for **coarse grained** services. Operations' parameters, return values
311 and exceptions are passed **by-value**. Remotable Services are not allowed to make use of method
312 **overloading**.

313 3.3 @Callback

314 A callback interface is declared by using a @Callback annotation on a Java service interface, with the
315 Java Class object of the callback interface as a parameter. There is another form of the @Callback
316 annotation, without any parameters, that specifies callback injection for a setter method or a field of an
317 implementation.

318 3.4 @AsyncInvocation

319 An interface can be annotated with @AsyncInvocation or with the equivalent
320 @Requires("sca:asyncInvocation") annotation to indicate that request/response operations of that
321 interface are **long running** and that response messages are likely to be sent an arbitrary length of time
322 after the initial request message is sent to the target service. This is described in the [SCA Assembly
323 Specification \[ASSEMBLY\]](#).

324 For a service client, it is strongly recommended that the client uses the asynchronous form of the client
325 interface when using a reference to a service with an interface annotated with @AsyncInvocation, using
326 either polling or callbacks to receive the response message. See the sections "[Asynchronous
327 Programming](#)" and the section "[JAX-WS Client Asynchronous API for a Synchronous Service](#)" for more
328 details about the asynchronous client API.

329 For a service implementation, SCA provides an **asynchronous service** mapping of the WSDL
330 request/response interface which enables the service implementation to send the response message at
331 an arbitrary time after the original service operation is invoked. This is described in the section
332 "[Asynchronous handling of Long Running Service Operations](#)".

333 **3.5 SCA Java Annotations for Interface Classes**

334 A Java interface referenced by the @interface attribute of an <interface.java/> element MUST NOT
335 contain any of the following SCA Java annotations:

336 @AllowsPassByReference, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit,
337 @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service. [JCA30006]

338 A Java interface referenced by the @callbackInterface attribute of an <interface.java/> element MUST
339 NOT contain any of the following SCA Java annotations:

340 @AllowsPassByReference, @Callback, @ComponentName, @Constructor, @Context, @Destroy,
341 @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service. [JCA30007]

342 **3.6 Compatibility of Java Interfaces**

343 The SCA Assembly Model specification [ASSEMBLY] defines a number of criteria that need to be
344 satisfied in order for two interfaces to be compatible or have a compatible superset or subset relationship.
345 If these interfaces are both Java interfaces, compatibility also means that every method that is present in
346 both interfaces is defined consistently in both interfaces with respect to the @OneWay annotation, that is,
347 the annotation is either present in both interfaces or absent in both interfaces. [JCA30009]

348 4 SCA Component Implementation Lifecycle

349 This section describes the lifecycle of an SCA component implementation.

350 4.1 Overview of SCA Component Implementation Lifecycle

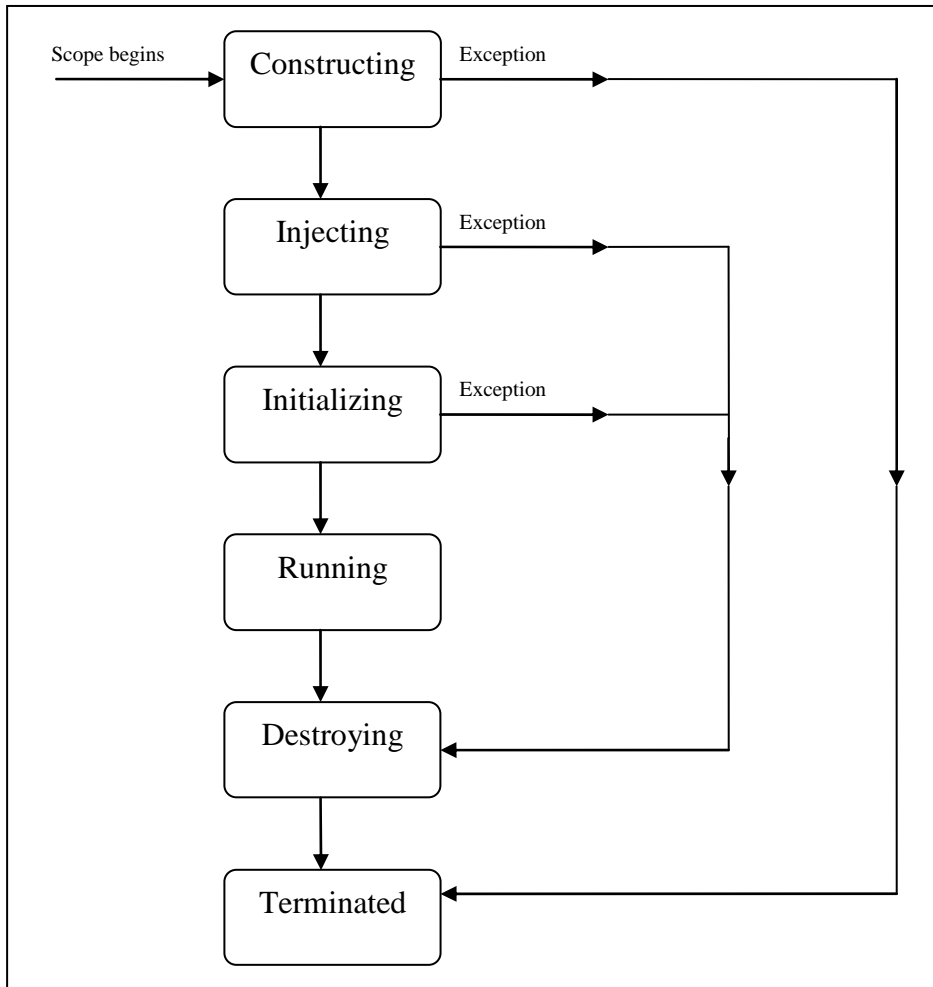
351 At a high level, there are 3 main phases through which an SCA component implementation will transition
352 when it is used by an SCA Runtime:

- 353 • **The Initialization phase.** This involves constructing an instance of the component implementation
354 class and injecting any properties and references. Once injection is complete, the method annotated
355 with `@Init` is called, if present, which provides the component implementation an opportunity to
356 perform any internal initialization it requires.
- 357 • **The Running phase.** This is where the component implementation has been initialized and the SCA
358 Runtime can dispatch service requests to it over its Service interfaces.
- 359 • **The Destroying phase.** This is where the component implementation's scope has ended and the
360 SCA Runtime destroys the component implementation instance. The SCA Runtime calls the method
361 annotated with `@Destroy`, if present, which provides the component implementation an opportunity to
362 perform any internal clean up that is required.

363 4.2 SCA Component Implementation Lifecycle State Diagram

364 The state diagram in Figure 4-1 shows the lifecycle of an SCA component implementation. The sections
365 that follow it describe each of the states that it contains.

366 It should be noted that some component implementation specifications might not implement all states of
367 the lifecycle. In this case, that state of the lifecycle is skipped over.



368
369 *Figure 4-1: SCA - Component Implementation Lifecycle*

370 **4.2.1 Constructing State**

371 The SCA Runtime **MUST** call a constructor of the component implementation at the start of the
 372 Constructing state. [JCA40001] The SCA Runtime **MUST** perform any constructor reference or property
 373 injection when it calls the constructor of a component implementation. [JCA40002]

374 The result of invoking operations on any injected references when the component implementation is in
 375 the Constructing state is undefined.

376 When the constructor completes successfully, the SCA Runtime **MUST** transition the component
 377 implementation to the Injecting state. [JCA40003] If an exception is thrown whilst in the Constructing
 378 state, the SCA Runtime **MUST** transition the component implementation to the Terminated state.
 379 [JCA40004]

380 **4.2.2 Injecting State**

381 When a component implementation instance is in the Injecting state, the SCA Runtime **MUST** first inject
 382 all field and setter properties that are present into the component implementation. [JCA40005] The order
 383 in which the properties are injected is unspecified.

384 When a component implementation instance is in the Injecting state, the SCA Runtime **MUST** inject all
 385 field and setter references that are present into the component implementation, after all the properties
 386 have been injected. [JCA40006] The order in which the references are injected is unspecified.

387 The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected
388 properties and references are made visible to the component implementation without requiring the
389 component implementation developer to do any specific synchronization. [JCA40007]
390 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
391 component implementation is in the Injecting state. [JCA40008]
392 The result of invoking operations on any injected references when the component implementation is in
393 the Injecting state is undefined.
394 When the injection of properties and references completes successfully, the SCA Runtime MUST
395 transition the component implementation to the Initializing state. [JCA40009] If an exception is thrown
396 whilst injecting properties or references, the SCA Runtime MUST transition the component
397 implementation to the Destroying state. [JCA40010] If a property or reference is unable to be injected, the
398 SCA Runtime MUST transition the component implementation to the Destroying state. [JCA40024]

399 4.2.3 Initializing State

400 When the component implementation enters the Initializing State, the SCA Runtime MUST call the
401 method annotated with @Init on the component implementation, if present. [JCA40011]
402 The component implementation can invoke operations on any injected references when it is in the
403 Initializing state. However, depending on the order in which the component implementations are
404 initialized, the target of the injected reference might not be available since it has not yet been initialized. If
405 a component implementation invokes an operation on an injected reference that refers to a target that has
406 not yet been initialized, the SCA Runtime MUST throw a ServiceUnavailableException. [JCA40012]
407 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
408 component implementation instance is in the Initializing state. [JCA40013]
409 Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition the
410 component implementation to the Running state. [JCA40014] If an exception is thrown whilst initializing,
411 the SCA Runtime MUST transition the component implementation to the Destroying state. [JCA40015]

412 4.2.4 Running State

413 The SCA Runtime MUST invoke Service methods on a component implementation instance when the
414 component implementation is in the Running state and a client invokes operations on a service offered by
415 the component. [JCA40016]
416 The component implementation can invoke operations on any injected references when the component
417 implementation instance is in the Running state.
418 When the component implementation scope ends, the SCA Runtime MUST transition the component
419 implementation to the Destroying state. [JCA40017]

420 4.2.5 Destroying State

421 When a component implementation enters the Destroying state, the SCA Runtime MUST call the method
422 annotated with @Destroy on the component implementation, if present. [JCA40018]
423 The component implementation can invoke operations on any injected references when it is in the
424 Destroying state. However, depending on the order in which the component implementations are
425 destroyed, the target of the injected reference might no longer be available since it has been destroyed. If
426 a component implementation invokes an operation on an injected reference that refers to a target that has
427 been destroyed, the SCA Runtime MUST throw an InvalidServiceException. [JCA40019]
428 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
429 component implementation instance is in the Destroying state. [JCA40020]
430 Once the method annotated with @Destroy completes successfully, the SCA Runtime MUST transition
431 the component implementation to the Terminated state. [JCA40021] If an exception is thrown whilst
432 destroying, the SCA Runtime MUST transition the component implementation to the Terminated state.
433 [JCA40022]

434 **4.2.6 Terminated State**

435 The lifecycle of the SCA Component has ended.

436 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
437 component implementation instance is in the Terminated state. [JCA40023]

438 5 Client API

439 This section describes how SCA services can be programmatically accessed from components and also
440 from non-managed code, that is, code not running as an SCA component.

441 5.1 Accessing Services from an SCA Component

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

446 5.1.1 Using the Component Context API

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

449 5.2 Accessing Services from non-SCA Component Implementations

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

452 5.2.1 SCAClientFactory Interface and Related Classes

453 Client code can use the **SCAClientFactory** class to obtain proxy reference objects for a service which is
454 in an SCA Domain. The URI of the domain, the relative URI of the service and the business interface of
455 the service must all be known in order to use the SCAClientFactory class.

456
457 Objects which implement the SCAClientFactory are obtained using the newInstance() methods of the
458 SCAClientFactory class.

459 Snippet 5-1 is a sample of the code that a client would use:

460

```
461 package org.oasisopen.sca.client.example;
462
463 import java.net.URI;
464
465 import org.oasisopen.sca.client.SCAClientFactory;
466 import org.oasisopen.sca.client.example.HelloService;
467
468 /**
469  * Example of use of Client API for a client application to obtain
470  * an SCA reference proxy for a service in an SCA Domain.
471  */
472 public class Client1 {
473
474     public void someMethod() {
475
476         try {
477
478             String serviceURI = "SomeHelloServiceURI";
479             URI domainURI = new URI("SomeDomainURI");
480
481             SCAClientFactory scaClient =
482                 SCAClientFactory.newInstance( domainURI );
483             HelloService helloService =
484                 scaClient.getService(HelloService.class,
485                                     serviceURI);
```

```
486         String reply = helloService.sayHello("Mark");
487
488     } catch (Exception e) {
489         System.out.println("Received exception");
490     }
491 }
492 }
```

493 *Snippet 5-1: Using the SCAClientFactory Interface*

494

495 For details about the SCAClientFactory interface and its related classes see the section
496 ["SCAClientFactory Class"](#).

497 **6 Error Handling**

498 Clients calling service methods can experience business exceptions and SCA runtime exceptions.

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

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

504 7 Asynchronous Programming

505 Asynchronous programming of a service is where a client invokes a service and carries on executing
506 without waiting for the service to execute. Typically, the invoked service executes at some later time.
507 Output from the invoked service, if any, is fed back to the client through a separate mechanism, since no
508 output is available at the point where the service is invoked. This is in contrast to the call-and-return style
509 of synchronous programming, where the invoked service executes and returns any output to the client
510 before the client continues. The SCA asynchronous programming model consists of:

- 511 • support for non-blocking method calls
- 512 • callbacks

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

514 7.1 @OneWay

515 **Non-blocking calls** represent the simplest form of asynchronous programming, where the client of the
516 service invokes the service and continues processing immediately, without waiting for the service to
517 execute.

518 A method with a void return type and which has no declared exceptions can be marked with a **@OneWay**
519 annotation. This means that the method is non-blocking and communication with the service provider can
520 use a binding that buffers the request and sends it at some later time.

521 For a Java client to make a non-blocking call to methods that either return values or throw exceptions, a
522 Java client can use the JAX-WS asynchronous client API model that is described in [the section "JAX-WS
523 Client Asynchronous API for a Synchronous Service"](#). It is considered to be a best practice that service
524 designers define one-way methods as often as possible, in order to give the greatest degree of binding
525 flexibility to deployers.

526 7.2 Callbacks

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

- 530 • an interface for the provided service
- 531 • a callback interface that is provided by the client

532 Callbacks can be used for both remotable and local services. Either both interfaces of a bidirectional
533 service are remotable, or both are local. It is illegal to mix the two, as defined in [the SCA Assembly
534 Model specification \[ASSEMBLY\]](#).

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

538 7.2.1 Using Callbacks

539 Bidirectional interfaces and callbacks are used when a simple request/response pattern isn't sufficient to
540 capture the business semantics of a service interaction. Callbacks are well suited for cases when a
541 service request can result in multiple responses or new requests from the service back to the client, or
542 where the service might respond to the client some time after the original request has completed.

543 Snippet 7-1 shows a scenario in which bidirectional interfaces and callbacks could be used. A client
544 requests a quotation from a supplier. To process the enquiry and return the quotation, some suppliers
545 might need additional information from the client. The client does not know which additional items of
546 information will be needed by different suppliers. This interaction can be modeled as a bidirectional
547 interface with callback requests to obtain the additional information.

548

```
549 package somepackage;
550 import org.oasisopen.sca.annotation.Callback;
551 import org.oasisopen.sca.annotation.Remotable;
552
553 @Remotable
554 @Callback(QuotationCallback.class)
555 public interface Quotation {h
556     double requestQuotation(String productCode, int quantity);
557 }
558
559 @Remotable
560 public interface QuotationCallback {
561     String getState();
562     String getZipCode();
563     String getCreditRating();
564 }
```

565 *Snippet 7-1: Using a Bidirectional Interface*

566

567 In Snippet 7-1, the `requestQuotation` operation requests a quotation to supply a given quantity of a
568 specified product. The `QuotationCallback` interface provides a number of operations that the supplier can
569 use to obtain additional information about the client making the request. For example, some suppliers
570 might quote different prices based on the state or the ZIP code to which the order will be shipped, and
571 some suppliers might quote a lower price if the ordering company has a good credit rating. Other
572 suppliers might quote a standard price without requesting any additional information from the client.

573 Snippet 7-2 illustrates a possible implementation of the example service, using the `@Callback` annotation
574 to request that a callback proxy be injected.

575

```
576 @Callback
577 protected QuotationCallback callback;
578
579 public double requestQuotation(String productCode, int quantity) {
580     double price = getPrice(productCode, quantity);
581     double discount = 0;
582     if (quantity > 1000 && callback.getState().equals("FL")) {
583         discount = 0.05;
584     }
585     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
586         discount += 0.05;
587     }
588     return price * (1-discount);
589 }
```

590 *Snippet 7-2: Example Implementation of a Service with a Bidirectional Interface*

591

592 Snippet 7-3 is taken from the client of this example service. The client's service implementation class
593 implements the methods of the `QuotationCallback` interface as well as those of its own service interface
594 `ClientService`.

595

```
596 public class ClientImpl implements ClientService, QuotationCallback {
597
598     private QuotationService myService;
599
600     @Reference
601     public void setMyService(QuotationService service) {
602         myService = service;
603     }
604 }
```

```

605     public void aClientMethod() {
606         ...
607         double quote = myService.requestQuotation("AB123", 2000);
608         ...
609     }
610
611     public String getState() {
612         return "TX";
613     }
614     public String getZipCode() {
615         return "78746";
616     }
617     public String getCreditRating() {
618         return "AA";
619     }
620 }

```

621 *Snippet 7-3: Example Client Using a Bidirectional Interface*

622

623 Snippet 7-3 the callback is **stateless**, i.e., the callback requests do not need any information relating to
624 the original service request. For a callback that needs information relating to the original service request
625 (a **stateful** callback), this information can be passed to the client by the service provider as parameters
626 on the callback request.

627 7.2.2 Callback Instance Management

628 Instance management for callback requests received by the client of the bidirectional service is handled in
629 the same way as instance management for regular service requests. If the client implementation has
630 STATELESS scope, the callback is dispatched using a newly initialized instance. If the client
631 implementation has COMPOSITE scope, the callback is dispatched using the same shared instance that
632 is used to dispatch regular service requests.

633 As described in [the section "Using Callbacks"](#), a stateful callback can obtain information relating to the
634 original service request from parameters on the callback request. Alternatively, a composite-scoped
635 client could store information relating to the original request as instance data and retrieve it when the
636 callback request is received. These approaches could be combined by using a key passed on the
637 callback request (e.g., an order ID) to retrieve information that was stored in a composite-scoped instance
638 by the client code that made the original request.

639 7.2.3 Callback Injection

640 When a bidirectional service is invoked, the SCA runtime MUST inject a callback reference for the
641 invoking service into all fields and setter methods of the service implementation class that are marked
642 with a @Callback annotation and typed by the callback interface of the bidirectional service, and the SCA
643 runtime MUST inject null into all other fields and setter methods of the service implementation class that
644 are marked with a @Callback annotation. [JCA60001] When a non-bidirectional service is invoked, the
645 SCA runtime MUST inject null into all fields and setter methods of the service implementation class that
646 are marked with a @Callback annotation. [JCA60002]

647 7.2.4 Implementing Multiple Bidirectional Interfaces

648 Since it is possible for a single implementation class to implement multiple services, it is also possible for
649 callbacks to be defined for each of the services that it implements. The service implementation can
650 include an injected field for each of its callbacks. The runtime injects the callback onto the appropriate
651 field based on the type of the callback. Snippet 7-4 shows the declaration of two fields, each of which
652 corresponds to a particular service offered by the implementation.

653

```

654     @Callback
655     protected MyService1Callback callback1;

```

```
656
657 @Callback
658 protected MyService2Callback callback2;
```

659 *Snippet 7-4: Multiple Bidirectional Interfaces in an Implementation*

660

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

663 7.2.5 Accessing Callbacks

664 In addition to injecting a reference to a callback service, it is also possible to obtain a reference to a
665 Callback instance by annotating a field or method of type **ServiceReference** with the **@Callback**
666 annotation.

667

668 A reference implementing the callback service interface can be obtained using
669 `ServiceReference.getService()`.

670 Snippet 7-5 comes from a service implementation that uses the callback API:

671

```
672 @Callback
673 protected ServiceReference<MyCallback> callback;
674
675 public void someMethod() {
676     MyCallback myCallback = callback.getService();    ...
677
678     myCallback.receiveResult(theResult);
679 }
680
```

681 *Snippet 7-5: Using the Callback API*

682

683 Because `ServiceReference` objects are serializable, they can be stored persistently and retrieved at a
684 later time to make a callback invocation after the associated service request has completed.
685 `ServiceReference` objects can also be passed as parameters on service invocations, enabling the
686 responsibility for making the callback to be delegated to another service.

687 Alternatively, a callback can be retrieved programmatically using the **RequestContext** API. Snippet 7-6
688 shows how to retrieve a callback in a method programmatically:

689

```
690 @Context
691 ComponentContext context;
692
693 public void someMethod() {
694     MyCallback myCallback = context.getRequestContext().getCallback();
695
696     ...
697
698     myCallback.receiveResult(theResult);
699 }
```

700 *Snippet 7-6: Using RequestContext to get a Callback*

701

702 This is necessary if the service implementation has **COMPOSITE** scope, because callback injection is not
703 performed for composite-scoped implementations.

7.3 Asynchronous handling of Long Running Service Operations

Long-running request-response operations are described in the SCA Assembly Specification [ASSEMBLY]. These operations are characterized by following the WSDL request-response message exchange pattern, but where the timing of the sending of the response message is arbitrarily later than the receipt of the request message, with an impact on the client component, on the service component and also on the transport binding used to communicate between them.

In SCA, such operations are marked with an intent "asyncInvocation" and is expected that the client component, the service component and the binding are all affected by the presence of this intent. This specification does not describe the effects of the intent on the binding, other than to note that in general, there is an implication that the sending of the response message is typically separate from the sending of the request message, typically requiring a separate response endpoint on the client to which the response can be sent.

For components that are clients of a long-running request-response operation, it is strongly recommended that the client makes use of the JAX-WS Client Asynchronous API, either using the polling interface or the callback mechanism described in the section "[JAX-WS Client Asynchronous API for a Synchronous Service](#)". The principle is that the client should not synchronously wait for a response from the long running operation since this could take a long time and it is preferable not to tie up resources while waiting.

For the service implementation component, the JAX-WS client asynchronous API is not suitable, so the SCA Java Common Annotations and APIs specification defines the SCA Asynchronous Service interface, which, like the JAX-WS client asynchronous API, is an alternative mapping of a WSDL request-response operation into a Java interface.

7.4 SCA Asynchronous Service Interface

The SCA Asynchronous Service interface follows some of the patterns defined by the JAX-WS client asynchronous API, but it is a simpler interface aligned with the needs of a service implementation class.

As an example, for a WSDL portType with a single operation "getPrice" with a String request parameter and a float response, the synchronous Java interface mapping appears in Snippet 7-7.

```
731
732 // synchronous mapping
733 public interface StockQuote {
734     float getPrice(String ticker);
735 }
```

736 *Snippet 7-7: Example Synchronous Java Interface Mapping*

737
738 The JAX-WS client asynchronous API for the same portType adds two asynchronous forms for each
739 synchronous method, as shown in Snippet 7-8.

```
740
741 // asynchronous mapping
742 public interface StockQuote {
743     float getPrice(String ticker);
744     Response<Float> getPriceAsync(String ticker);
745     Future<?> getPriceAsync(String ticker, AsyncHandler<Float> handler);
746 }
```

747 *Snippet 7-8: Example JAX-WS Client Asynchronous Java interface Mapping*

748
749 The SCA Asynchronous Service interface has a single method similar to the final one in the
750 asynchronous client interface, as shown in Snippet 7-8.

```
751
752 // asynchronous mapping
```

```

753 @Requires("sca:asyncInvocation")
754 public interface StockQuote {
755     void getPriceAsync(String ticker, ResponseDispatch<Float> dispatch);
756 }

```

757 *Snippet 7-9: Example SCA Asynchronous Service Java interface Mapping*

758

759 The main characteristics of the SCA asynchronous mapping are:

- 760 • there is a single method, with a name with the string "Async" appended to the operation name
- 761 • it has a void return type
- 762 • it has two input parameters, the first is the request message of the operation and the second is a
- 763 ResponseDispatch object typed by the response message of the operation (following the rules
- 764 expressed in the JAX-WS specification for the typing of the AsyncHandler object in the client
- 765 asynchronous API)
- 766 • it is annotated with the asyncInvocation intent
- 767 • if the synchronous method has any business faults/exceptions, it is annotated with @AsyncFault,
- 768 containing a list of the exception classes

769 Unlike the JAX-WS asynchronous client interface, there is only a single operation for the service
770 implementation to provide (it would be inconvenient for the service implementation to be required to
771 implement multiple methods for each operation in the WSDL interface).

772 The ResponseDispatch parameter is the mechanism by which the service implementation sends back the
773 response message resulting from the invocation of the service method. The ResponseDispatch is
774 serializable and it can be invoked once at any time after the invocation of the service method, either
775 before or after the service method returns. This enables the service implementation to store the
776 ResponseDispatch in serialized form and release resources while waiting for the completion of whatever
777 activities result from the processing of the initial invocation.

778 The ResponseDispatch object is allocated by the SCA runtime/binding implementation and it is expected
779 to contain whatever metadata is required to deliver the response message back to the client that invoked
780 the service operation.

781 **The SCA asynchronous service Java interface mapping of a WSDL request-response operation**
782 **MUST appear as follows:**

783 **The interface is annotated with the "asyncInvocation" intent.**

- 784 – **For each service operation in the WSDL, the Java interface contains an operation with**
- 785 – **a name which is the JAX-WS mapping of the WSDL operation name, with the suffix "Async"**
- 786 **added**
- 787 – **a void return type**
- 788 – **a set of input parameter(s) which match the JAX-WS mapping of the input parameter(s) of the**
- 789 **WSDL operation plus an additional last parameter which is a ResponseDispatch object typed by**
- 790 **the JAX-WS Response Bean mapping of the output parameter(s) of the WSDL operation, where**
- 791 **ResponseDispatch is the type defined in the SCA Java Common Annotations and APIs**
- 792 **specification. [JCA60003]**

793 **An SCA Runtime MUST support the use of the SCA asynchronous service interface for the interface of an**
794 **SCA service. [JCA60004]**

795 The ResponseDispatch object passed in as a parameter to a method of a service implementation using
796 the SCA asynchronous service Java interface can be invoked once only through either its sendResponse
797 method or through its sendFault method to return the response resulting from the service method
798 invocation. **If the SCA asynchronous service interface ResponseDispatch handleResponse method is**
799 **invoked more than once through either its sendResponse or its sendFault method, the SCA runtime**
800 **MUST throw an IllegalStateException. [JCA60005]**

801

802 For the purposes of matching interfaces (when wiring between a reference and a service, or when using
803 an implementation class by a component), an interface which has one or more methods which follow the
804 SCA asynchronous service pattern MUST be treated as if those methods are mapped as the equivalent
805 synchronous methods, as follows:

806 Asynchronous service methods are characterized by:

- 807 – void return type
- 808 – a method name with the suffix "Async"
- 809 – a last input parameter with a type of ResponseDispatch<X>
- 810 – annotation with the asyncInvocation intent
- 811 – possible annotation with the @AsyncFault annotation

812 The mapping of each such method is as if the method had the return type "X", the method name without
813 the suffix "Async" and all the input parameters except the last parameter of the type
814 ResponseDispatch<X>, plus the list of exceptions contained in the @AsyncFault annotation. [JCA60006]

8 Policy Annotations for Java

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

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

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

832 This specification has a series of annotations which provide the capability for the developer to attach
833 policy information to Java implementation code. The annotations concerned first provide general facilities
834 for attaching SCA Intents and Policy Sets to Java code. Secondly, there are further specific annotations
835 that deal with particular policy intents for certain policy domains such as Security and Transactions.

836 This specification supports using [the Common Annotations for the Java Platform specification \(JSR-250\)](#)
837 [\[JSR-250\]](#). An implication of adopting the common annotation for Java platform specification is that the
838 SCA Java specification supports consistent annotation and Java class inheritance relationships. SCA
839 policy annotation semantics follow the General Guidelines for Inheritance of Annotations in [the Common](#)
840 [Annotations for the Java Platform specification \[JSR-250\]](#), except that member-level annotations in a
841 class or interface do not have any effect on how class-level annotations are applied to other members of
842 the class or interface.

843

8.1 General Intent Annotations

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

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

848 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI
849 followed by the name of the Intent. The precise form used follows the string representation used by the
850 `javax.xml.namespace.QName` class, which is shown in Snippet 8-1.

851

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

853 *Snippet 8-1: Intent Format*

854

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

857 This representation is quite verbose, so we expect that reusable String constants will be defined for the
858 namespace part of this string, as well as for each intent that is used by Java code. SCA defines
859 constants for intents such as those in Snippet 8-2.

860

```

861 public static final String SCA_PREFIX =
862     "{http://docs.oasis-open.org/ns/opencsa/sca/200912}";
863 public static final String CONFIDENTIALITY =
864     SCA_PREFIX + "confidentiality";
865 public static final String CONFIDENTIALITY_MESSAGE =
866     CONFIDENTIALITY + ".message";

```

867 *Snippet 8-2: Example Intent Constants*

868

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

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

874 An example of the `@Requires` annotation with 2 qualified intents (from the Security domain) is shown in
875 Snippet 8-3:

876

```

877 @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})

```

878 *Snippet 8-3: Multiple Intents in One Annotation*

879

880 The annotation in Snippet 8-3 attaches the intents "confidentiality.message" and "integrity.message".

881 Snippet 8-4 is an example of a reference requiring support for confidentiality:

882

```

883 package com.foo;
884
885 import static org.oasisopen.sca.annotation.Confidentiality.*;
886 import static org.oasisopen.sca.annotation.Reference;
887 import static org.oasisopen.sca.annotation.Requires;
888
889 public class Foo {
890     @Requires(CONFIDENTIALITY)
891     @Reference
892     public void setBar(Bar bar) {
893         ...
894     }
895 }

```

896 *Snippet 8-4: Annotation a Reference*

897

898 Users can also choose to only use constants for the namespace part of the QName, so that they can add
899 new intents without having to define new constants. In that case, the definition of Snippet 8-4 would
900 instead look like Snippet 8-5.

901

```

902 package com.foo;
903
904 import static org.oasisopen.sca.Constants.*;
905 import static org.oasisopen.sca.annotation.Reference;
906 import static org.oasisopen.sca.annotation.Requires;
907
908 public class Foo {
909     @Requires(SCA_PREFIX+"confidentiality")
910     @Reference
911     public void setBar(Bar bar) {
912         ...
913     }
914 }

```


915 *Snippet 8-5: Using Intent Constants and strings*

916

917 The formal syntax [EBNF-Syntax] for the @Requires annotation follows:

```
918 '@Requires('' QualifiedIntent ''' ('','' QualifiedIntent ''')* ''')
```

919 where

```
920 QualifiedIntent ::= QName('.' Qualifier)*  
921 Qualifier ::= NCName
```

922

923 See [section @Requires](#) for the formal definition of the @Requires annotation.

924 **8.2 Specific Intent Annotations**

925 In addition to the general intent annotation supplied by the @Requires annotation described in section
926 8.2, it is also possible to have Java annotations that correspond to specific policy intents. SCA provides a
927 number of these specific intent annotations and it is also possible to create new specific intent
928 annotations for any intent.

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

932 For example, the SCA confidentiality intent described in [the section on General Intent Annotations](#) using
933 the @Requires(CONFIDENTIALITY) annotation can also be specified with the @Confidentiality specific
934 intent annotation. The specific intent annotation for the "integrity" security intent is shown in Snippet 8-6.

935

```
936 @Integrity
```

937 *Snippet 8-6: Example Specific Intent Annotation*

938

939 An example of a qualified specific intent for the "authentication" intent is shown in Snippet 8-7.

940

```
941 @Authentication( { "message", "transport" } )
```

942 *Snippet 8-7: Example Qualified Specific Intent Annotation*

943

944 This annotation attaches the pair of qualified intents: "authentication.message" and
945 "authentication.transport" (the sca: namespace is assumed in this both of these cases –
946 "http://docs.oasis-open.org/ns/opencsa/sca/200912").

947 The general form of specific intent annotations is shown in Snippet 8-8

948

```
949 '@' Intent ((' qualifiers '))?
```

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

```
951 Intent ::= NCName  
952 qualifiers ::= '' qualifier '' ('','' qualifier ''')*  
953 qualifier ::= NCName ('.' qualifier)?
```

954 *Snippet 8-8: Specific Intent Annotation Format*

955 8.2.1 How to Create Specific Intent Annotations

956 SCA identifies annotations that correspond to intents by providing an `@Intent` annotation which MUST be
957 used in the definition of a specific intent annotation. [JCA70001]

958 The `@Intent` annotation takes a single parameter, which (like the `@Requires` annotation) is the String
959 form of the QName of the intent. As part of the intent definition, it is good practice (although not required)
960 to also create String constants for the Namespace, for the Intent and for Qualified versions of the Intent (if
961 defined). These String constants are then available for use with the `@Requires` annotation and it is also
962 possible to use one or more of them as parameters to the specific intent annotation.

963 Alternatively, the QName of the intent can be specified using separate parameters for the
964 targetNamespace and the localPart, as shown in Snippet 8-9:

965

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

967 *Snippet 8-9: Defining a Specific Intent Annotation*

968

969 See [section @Intent](#) for the formal definition of the `@Intent` annotation.

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

972 In this case, the attribute's definition needs to be marked with the `@Qualifier` annotation. The `@Qualifier`
973 tells SCA that the value of the attribute is treated as a qualifier for the intent represented by the whole
974 annotation. If more than one qualifier value is specified in an annotation, it means that multiple qualified
975 forms exist. For example the annotation in Snippet 8-10

976

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

978 *Snippet 8-10: Multiple Qualifiers in an Annotation'*

979

980 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport" are set for
981 the element to which the `@Confidentiality` annotation is attached.

982 See [section @Qualifier](#) for the formal definition of the `@Qualifier` annotation.

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

985 8.3 Application of Intent Annotations

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

- 987 • Java class
- 988 • Java interface
- 989 • Method
- 990 • Field
- 991 • Constructor parameter

992 Intent annotations MUST NOT be applied to the following:

- 993 • A method of a service implementation class, except for a setter method that is either annotated with
994 `@Reference` or introspected as an SCA reference according to the rules in the appropriate
995 Component Implementation specification
- 996 • A service implementation class field that is not either annotated with `@Reference` or introspected as
997 an SCA reference according to the rules in the appropriate Component Implementation specification
- 998 • A service implementation class constructor parameter that is not annotated with `@Reference`

999 [JCA70002]

1000 Intent annotations can be applied to classes, interfaces, and interface methods. Applying an intent
1001 annotation to a field, setter method, or constructor parameter allows intents to be defined at references.
1002 Intent annotations can also be applied to reference interfaces and their methods.

1003 Where multiple intent annotations (general or specific) are applied to the same Java element, the SCA
1004 runtime MUST compute the combined intents for the Java element by merging the intents from all intent
1005 annotations on the Java element according to the SCA Policy Framework [POLICY] rules for merging
1006 intents at the same hierarchy level. [JCA70003]

1007 An example of multiple policy annotations being used together is shown in Snippet 8-11:

1008

```
1009 @Authentication  
1010 @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1011 *Snippet 8-11: Multiple Policy Annotations*

1012

1013 In this case, the effective intents are "authentication", "confidentiality.message" and "integrity.message".

1014 If intent annotations are specified on both an interface method and the method's declaring interface, the
1015 SCA runtime MUST compute the effective intents for the method by merging the combined intents from
1016 the method with the combined intents for the interface according to the SCA Policy Framework [POLICY]
1017 rules for merging intents within a structural hierarchy, with the method at the lower level and the interface
1018 at the higher level. [JCA70004] This merging process does not remove or change any intents that are
1019 applied to the interface.

1020 8.3.1 Intent Annotation Examples

1021 The following examples show how the rules defined in section 8.3 are applied.

1022 Snippet 8-12 shows how intents on references are merged. In this example, the intents for `myRef` are
1023 "authentication" and "confidentiality.message".

1024

```
1025 @Authentication  
1026 @Requires(CONFIDENTIALITY)  
1027 @Confidentiality("message")  
1028 @Reference  
1029 protected MyService myRef;
```

1030 *Snippet 8-12: Merging Intents on References*

1031

1032 Snippet 8-13 shows that mutually exclusive intents cannot be applied to the same Java element. In this
1033 example, the Java code is in error because of contradictory mutually exclusive intents
1034 "managedTransaction" and "noManagedTransaction".

1035

```
1036 @Requires({SCA_PREFIX+"managedTransaction",  
1037           SCA_PREFIX+"noManagedTransaction"})  
1038 @Reference  
1039 protected MyService myRef;
```

1040 *Snippet 8-13: Mutually Exclusive Intents*

1041

1042 Snippet 8-14 shows that intents can be applied to Java service interfaces and their methods. In this
1043 example, the effective intents for `MyService.mymethod()` are "authentication" and "confidentiality".

1044

```
1045 @Authentication  
1046 public interface MyService {
```

```

1047     @Confidentiality
1048     public void mymethod();
1049 }
1050 @Service(MyService.class)
1051 public class MyServiceImpl {
1052     public void mymethod() {...}
1053 }

```

1054 *Snippet 8-14: Intents on Java Interfaces, Interface Methods, and Java Classes*

1055
1056 Snippet 8-15 shows that intents can be applied to Java service implementation classes. In this example,
1057 the effective intents for `MyService.mymethod()` are "authentication", "confidentiality", and
1058 "managedTransaction".

```

1060 @Authentication
1061 public interface MyService {
1062     @Confidentiality
1063     public void mymethod();
1064 }
1065 @Service(MyService.class)
1066 @Requires(SCA_PREFIX+"managedTransaction")
1067 public class MyServiceImpl {
1068     public void mymethod() {...}
1069 }

```

1070 *Snippet 8-15: Intents on Java Service Implementation Classes*

1071
1072 Snippet 8-16 shows that intents can be applied to Java reference interfaces and their methods, and also
1073 to Java references. In this example, the effective intents for the method `mymethod()` of the reference
1074 `myRef` are "authentication", "integrity", and "confidentiality".

```

1076 @Authentication
1077 public interface MyRefInt {
1078     @Integrity
1079     public void mymethod();
1080 }
1081 @Service(MyService.class)
1082 public class MyServiceImpl {
1083     @Confidentiality
1084     @Reference
1085     protected MyRefInt myRef;
1086 }

```

1087 *Snippet 8-16: Intents on Java References and their Interfaces and Methods*

1088
1089 Snippet 8-17 shows that intents cannot be applied to methods of Java implementation classes. In this
1090 example, the Java code is in error because of the `@Authentication` intent annotation on the
1091 implementation method `MyServiceImpl.mymethod()`.

```

1093 public interface MyService {
1094     public void mymethod();
1095 }
1096 @Service(MyService.class)
1097 public class MyServiceImpl {
1098     @Authentication
1099     public void mymethod() {...}
1100 }

```

1101 *Snippet 8-17: Intent on Implementation Method*

1102 Snippet 8-18 shows one effect of applying the SCA Policy Framework rules for merging intents within a
1103 structural hierarchy to Java service interfaces and their methods. In this example a qualified intent
1104 overrides an unqualified intent, so the effective intent for `MyService.mymethod()` is
1105 "confidentiality.message".

1106

```
1107 @Confidentiality("message")  
1108 public interface MyService {  
1109     @Confidentiality  
1110     public void mymethod();  
1111 }
```

1112 *Snippet 8-18: Merging Qualified and Unqualified Intents on Java Interfaces and Methods*

1113

1114 Snippet 8-19 shows another effect of applying the SCA Policy Framework rules for merging intents within
1115 a structural hierarchy to Java service interfaces and their methods. In this example a lower-level intent
1116 causes a mutually exclusive higher-level intent to be ignored, so the effective intent for `mymethod1()` is
1117 "managedTransaction" and the effective intent for `mymethod2()` is "noManagedTransaction".

1118

```
1119 @Requires(SCA_PREFIX+"managedTransaction")  
1120 public interface MyService {  
1121     public void mymethod1();  
1122     @Requires(SCA_PREFIX+"noManagedTransaction")  
1123     public void mymethod2();  
1124 }
```

1125 *Snippet 8-19: Merging Mutually Exclusive Intents on Java Interfaces and Methods*

1126 **8.3.2 Inheritance and Annotation**

1127 Snippet 8-20 shows the inheritance relations of intents on classes, operations, and super classes.

1128

```
1129 package services.hello;  
1130 import org.oasisopen.sca.annotation.Authentication;  
1131 import org.oasisopen.sca.annotation.Integrity;  
1132  
1133 @Integrity("transport")  
1134 @Authentication  
1135 public class HelloService {  
1136     @Integrity  
1137     @Authentication("message")  
1138     public String hello(String message) {...}  
1139  
1140     @Integrity  
1141     @Authentication("transport")  
1142     public String helloThere() {...}  
1143 }  
1144  
1145 package services.hello;  
1146 import org.oasisopen.sca.annotation.Authentication;  
1147 import org.oasisopen.sca.annotation.Confidentiality;  
1148  
1149 @Confidentiality("message")  
1150 public class HelloChildService extends HelloService {  
1151     @Confidentiality("transport")  
1152     public String hello(String message) {...}  
1153     @Authentication  
1154     String helloWorld() {...}  
1155 }
```

1156 Snippet 8-20: Usage example of Annotated Policy and Inheritance

1157

1158 The effective intent annotation on the **helloWorld** method of **HelloChildService** is **@Authentication** and
1159 **@Confidentiality("message")**.

1160 The effective intent annotation on the **hello** method of **HelloChildService** is **@Confidentiality("transport")**,

1161 The effective intent annotation on the **helloThere** method of **HelloChildService** is **@Integrity** and
1162 **@Authentication("transport")**, the same as for this method in the **HelloService** class.

1163 The effective intent annotation on the **hello** method of **HelloService** is **@Integrity** and
1164 **@Authentication("message")**

1165

1166 Table 8-1 shows the equivalent declarative security interaction policy of the methods of the **HelloService**
1167 and **HelloChildService** implementations corresponding to the Java classes shown in Snippet 8-20.

1168

	Method		
Class	hello()	helloThere()	helloWorld()
HelloService	integrity authentication.message	integrity authentication.transport	N/A
HelloChildService	confidentiality.transport	integrity authentication.transport	authentication confidentiality.message

1169 Table 8-1: Declarative Intents Equivalent to Annotated Intents in Snippet 8-20

1170 8.4 Relationship of Declarative and Annotated Intents

1171 Annotated intents on a Java class cannot be overridden by declarative intents in a composite document
1172 which uses the class as an implementation. This rule follows the general rule for intents that they
1173 represent requirements of an implementation in the form of a restriction that cannot be relaxed.

1174 However, a restriction can be made more restrictive so that an unqualified version of an intent expressed
1175 through an annotation in the Java class can be qualified by a declarative intent in a using composite
1176 document.

1177 8.5 Policy Set Annotations

1178 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies. For example,
1179 a concrete policy is the specific encryption algorithm to use when encrypting messages when using a
1180 specific communication protocol to link a reference to a service.

1181 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation. The
1182 **@PolicySets** annotation either takes the QName of a single policy set as a string or the name of two or
1183 more policy sets as an array of strings:

1184

```
1185 '@PolicySets({' policySetQName (',' policySetQName )* '})'
```

1186 Snippet 8-21: PolicySet Annotation Format

1187

1188 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

1189 An example of the **@PolicySets** annotation is shown in Snippet 8-22:

1190

```
1191 @Reference(name="helloService", required=true)
```

```

1192 @PolicySets({ MY_NS + "WS_Encryption_Policy",
1193              MY_NS + "WS_Authentication_Policy" })
1194 public setHelloService(HelloService service) {
1195     . . .
1196 }

```

1197 *Snippet 8-22: Use of @PolicySets*

1198

1199 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
1200 using the namespace defined for the constant MY_NS.

1201 PolicySets need to satisfy intents expressed for the implementation when both are present, according to
1202 the rules defined in [the Policy Framework specification \[POLICY\]](#).

1203 The SCA Policy Set annotation can be applied to the following Java elements:

- 1204 • Java class
- 1205 • Java interface
- 1206 • Method
- 1207 • Field
- 1208 • Constructor parameter

1209 The @PolicySets annotation MUST NOT be applied to the following:

- 1210 • A method of a service implementation class, except for a setter method that is either annotated with
1211 @Reference or introspected as an SCA reference according to the rules in the appropriate
1212 Component Implementation specification
- 1213 • A service implementation class field that is not either annotated with @Reference or introspected as
1214 an SCA reference according to the rules in the appropriate Component Implementation specification
- 1215 • A service implementation class constructor parameter that is not annotated with @Reference

1216 [\[JCA70005\]](#)

1217 The @PolicySets annotation can be applied to classes, interfaces, and interface methods. Applying a
1218 @PolicySets annotation to a field, setter method, or constructor parameter allows policy sets to be
1219 defined at references. The @PolicySets annotation can also be applied to reference interfaces and their
1220 methods.

1221 If the @PolicySets annotation is specified on both an interface method and the method's declaring
1222 interface, the SCA runtime MUST compute the effective policy sets for the method by merging the policy
1223 sets from the method with the policy sets from the interface. [\[JCA70006\]](#) This merging process does not
1224 remove or change any policy sets that are applied to the interface.

1225 8.6 Security Policy Annotations

1226 This section introduces annotations for commonly used SCA security intents, as defined in [the SCA
1227 Policy Framework Specification \[POLICY\]](#). Also see the SCA Policy Framework Specification for
1228 additional security policy intents that can be used with the @Requires annotation. The following
1229 annotations for security policy intents and qualifiers are defined:

- 1230 • @Authentication
- 1231 • @Authorization
- 1232 • @Confidentiality
- 1233 • @Integrity
- 1234 • @MutualAuthentication

1235 The @Authentication, @Confidentiality, and @Integrity intents have the same pair of Qualifiers:

- 1236 • message
- 1237 • transport

1238 The formal definitions of the security intent annotations are found in the section "Java Annotations".
1239 Snippet 8-23 shows an example of applying security intents to the setter method used to inject a
1240 reference. Accessing the hello operation of the referenced HelloService requires both "integrity.message"
1241 and "authentication.message" intents to be honored.

1242

```
1243 package services.hello;  
1244 // Interface for HelloService  
1245 public interface HelloService {  
1246     String hello(String helloMsg);  
1247 }  
1248  
1249 package services.client;  
1250 // Interface for ClientService  
1251 public interface ClientService {  
1252     public void clientMethod();  
1253 }  
1254  
1255 // Implementation class for ClientService  
1256 package services.client;  
1257  
1258 import services.hello.HelloService;  
1259 import org.oasisopen.sca.annotation.*;  
1260  
1261 @Service(ClientService.class)  
1262 public class ClientServiceImpl implements ClientService {  
1263  
1264     private HelloService helloService;  
1265  
1266     @Reference(name="helloService", required=true)  
1267     @Integrity("message")  
1268     @Authentication("message")  
1269     public void setHelloService(HelloService service) {  
1270         helloService = service;  
1271     }  
1272  
1273     public void clientMethod() {  
1274         String result = helloService.hello("Hello World!");  
1275         ...  
1276     }  
1277 }
```

1278 *Snippet 8-23: Usage of Security Intents on a Reference*

1279 **8.7 Transaction Policy Annotations**

1280 This section introduces annotations for commonly used SCA transaction intents, as defined in [the SCA](#)
1281 [Policy Framework specification \[POLICY\]](#). Also see the SCA Policy Framework Specification for
1282 additional transaction policy intents that can be used with the @Requires annotation. The following
1283 annotations for transaction policy intents and qualifiers are defined:

- 1284 • @ManagedTransaction
- 1285 • @NoManagedTransaction
- 1286 • @SharedManagedTransaction

1287 The @ManagedTransaction intent has the following Qualifiers:

- 1288 • global
- 1289 • local

1290 The formal definitions of the transaction intent annotations are found in the section "Java Annotations".

1291 Snippet 8-24 shows an example of applying a transaction intent to a component implementation, where
1292 the component implementation requires a global transaction.

1293

```
1294 package services.hello;  
1295 // Interface for HelloService  
1296 public interface HelloService {  
1297     String hello(String helloMsg);  
1298 }  
1299  
1300 // Implementation class for HelloService  
1301 package services.hello.impl;  
1302  
1303 import services.hello.HelloService;  
1304 import org.oasisopen.sca.annotation.*;  
1305  
1306 @Service(HelloService.class)  
1307 @ManagedTransaction("global")  
1308 public class HelloServiceImpl implements HelloService {  
1309  
1310     public void someMethod() {  
1311         ...  
1312     }  
1313 }
```

1314 *Snippet 8-24: Usage of Transaction Intents in an Implementation*

1315 9 Java API

1316 This section provides a reference for the Java API offered by SCA.

1317 9.1 Component Context

1318 Figure 9-1 defines the **ComponentContext** interface:

1319

```
1320 package org.oasisopen.sca;
1321 import java.util.Collection;
1322 public interface ComponentContext {
1323
1324     String getURI();
1325
1326     <B> B getService(Class<B> businessInterface, String referenceName);
1327
1328     <B> ServiceReference<B> getServiceReference( Class<B> businessInterface,
1329                                               String referenceName);
1330     <B> Collection<B> getServices( Class<B> businessInterface,
1331                                 String referenceName);
1332
1333     <B> Collection<ServiceReference<B>> getServiceReferences(
1334                                     Class<B> businessInterface,
1335                                     String referenceName);
1336
1337     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface);
1338
1339     <B> ServiceReference<B> createSelfReference( Class<B> businessInterface,
1340                                               String serviceName);
1341
1342     <B> B getProperty(Class<B> type, String propertyName);
1343
1344     RequestContext getRequestContext();
1345
1346     <B> ServiceReference<B> cast(B target) throws IllegalArgumentException;
1347
1348 }
```

1349 *Figure 9-1: ComponentContext Interface*

1350

1351 **getURI () method:**

1352 Returns the absolute URI of the component within the SCA Domain.

1353 Returns:

- 1354 • **String** which contains the absolute URI of the component in the SCA Domain
1355 **The ComponentContext.getURI method MUST return the absolute URI of the component in the SCA**
1356 **Domain. [JCA80008]**

1357 Parameters:

- 1358 • **none**

1359 Exceptions:

- 1360 • **none**

1361

1362 **getService (Class businessInterface, String referenceName) method:**

1363 Returns a typed service proxy object for a reference defined by the current component, where the
1364 reference has multiplicity 0..1 or 1..1.

1365 Returns:

- 1366 • **B** which is a proxy object for the reference, which implements the interface B contained in the
1367 businessInterface parameter.

1368 The ComponentContext.getService method MUST return the proxy object implementing the interface
1369 provided by the businessInterface parameter, for the reference named by the referenceName
1370 parameter with the interface defined by the businessInterface parameter when that reference has a
1371 target service configured. [JCA80009]

1372 The ComponentContext.getService method MUST return null if the multiplicity of the reference
1373 named by the referenceName parameter is 0..1 and the reference has no target service configured.
1374 [JCA80010]

1375 Parameters:

- 1376 • **Class businessInterface** - the Java interface for the service reference
- 1377 • **String referenceName** - the name of the service reference

1378 Exceptions:

- 1379 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the
1380 reference identified by the referenceName parameter has multiplicity of 0..n or 1..n. [JCA80001]

- 1381 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the
1382 component does not have a reference with the name supplied in the referenceName parameter.
1383 [JCA80011]

- 1384 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the service
1385 reference with the name supplied in the referenceName does not have an interface compatible with
1386 the interface supplied in the businessInterface parameter. [JCA80012]

1387

1388 **getServiceReference (Class businessInterface, String referenceName) method:**

1389 Returns a ServiceReference object for a reference defined by the current component, where the
1390 reference has multiplicity 0..1 or 1..1.

1391 Returns:

- 1392 • **ServiceReference** which is a ServiceReference proxy object for the reference, which implements
1393 the interface contained in the businessInterface parameter.

1394 The ComponentContext.getServiceReference method MUST return a ServiceReference object typed
1395 by the interface provided by the businessInterface parameter, for the reference named by the
1396 referenceName parameter with the interface defined by the businessInterface parameter when that
1397 reference has a target service configured. [JCA80013]

1398 The ComponentContext.getServiceReference method MUST return null if the multiplicity of the
1399 reference named by the referenceName parameter is 0..1 and the reference has no target service
1400 configured. [JCA80007]

1401 Parameters:

- 1402 • **Class businessInterface** - the Java interface for the service reference
- 1403 • **String referenceName** - the name of the service reference

1404 Exceptions:

- 1405 • The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if
1406 the reference named by the referenceName parameter has multiplicity greater than one. [JCA80004]

- 1407 • The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if
1408 the reference named by the referenceName parameter does not have an interface of the type defined
1409 by the businessInterface parameter. [JCA80005]

- 1410 • The `ComponentContext.getServiceReference` method MUST throw an `IllegalArgumentException` if
1411 the component does not have a reference with the name provided in the `referenceName` parameter.
1412 [JCA80006]

1413

1414 ***getServices(Class businessInterface, String referenceName) method:***

1415 Returns a list of typed service proxies for a reference defined by the current component, where the
1416 reference has multiplicity 0..n or 1..n.

1417 Returns:

- 1418 • **Collection** which is a collection of proxy objects for the reference, one for each target service to
1419 which the reference is wired, where each proxy object implements the interface B contained in the
1420 `businessInterface` parameter.

1421 The `ComponentContext.getServices` method MUST return a collection containing one proxy object
1422 implementing the interface provided by the `businessInterface` parameter for each of the target
1423 services configured on the reference identified by the `referenceName` parameter. [JCA80014]

1424 The `ComponentContext.getServices` method MUST return an empty collection if the service reference
1425 with the name supplied in the `referenceName` parameter is not wired to any target services.
1426 [JCA80015]

1427 Parameters:

- 1428 • **Class businessInterface** - the Java interface for the service reference
1429 • **String referenceName** - the name of the service reference

1430 Exceptions:

1431 • The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the
1432 reference identified by the `referenceName` parameter has multiplicity of 0..1 or 1..1. [JCA80016]

1433 • The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the
1434 component does not have a reference with the name supplied in the `referenceName` parameter.
1435 [JCA80017]

1436 • The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the service
1437 reference with the name supplied in the `referenceName` does not have an interface compatible with
1438 the interface supplied in the `businessInterface` parameter. [JCA80018]

1439

1440 ***getServiceReferences(Class businessInterface, String referenceName) method:***

1441 Returns a list of typed `ServiceReference` objects for a reference defined by the current component, where
1442 the reference has multiplicity 0..n or 1..n.

1443 Returns:

- 1444 • **Collection<ServiceReference>** which is a collection of `ServiceReference` objects for the
1445 reference, one for each target service to which the reference is wired, where each proxy object
1446 implements the interface B contained in the `businessInterface` parameter. The collection is empty if
1447 the reference is not wired to any target services.

1448 The `ComponentContext.getServiceReferences` method MUST return a collection containing one
1449 `ServiceReference` object typed by the interface provided by the `businessInterface` parameter for each
1450 of the target services configured on the reference identified by the `referenceName` parameter.
1451 [JCA80019]

1452 The `ComponentContext.getServiceReferences` method MUST return an empty collection if the
1453 service reference with the name supplied in the `referenceName` parameter is not wired to any target
1454 services. [JCA80020]

1455 Parameters:

- 1456 • **Class businessInterface** - the Java interface for the service reference

1457 • **String referenceName** - the name of the service reference

1458 Exceptions:

1459 • The `ComponentContext.getServiceReferences` method MUST throw an `IllegalArgumentException` if
1460 the reference identified by the `referenceName` parameter has multiplicity of 0..1 or 1..1. [JCA80021]

1461 • The `ComponentContext.getServiceReferences` method MUST throw an `IllegalArgumentException` if
1462 the component does not have a reference with the name supplied in the `referenceName` parameter.
1463 [JCA80022]

1464 • The `ComponentContext.getServiceReferences` method MUST throw an `IllegalArgumentException` if
1465 the service reference with the name supplied in the `referenceName` does not have an interface
1466 compatible with the interface supplied in the `businessInterface` parameter. [JCA80023]

1467

1468 **`createSelfReference(Class businessInterface)` method:**

1469 Returns a `ServiceReference` object that can be used to invoke this component over the designated
1470 service.

1471 Returns:

1472 • **`ServiceReference`** which is a `ServiceReference` object for the service of this component which
1473 has the supplied business interface. If the component has multiple services with the same business
1474 interface the SCA runtime can return a `ServiceReference` for any one of them.

1475 The `ComponentContext.createSelfReference` method MUST return a `ServiceReference` object typed
1476 by the interface defined by the `businessInterface` parameter for one of the services of the invoking
1477 component which has the interface defined by the `businessInterface` parameter. [JCA80024]

1478 Parameters:

1479 • **`Class businessInterface`** - the Java interface for the service

1480 Exceptions:

1481 • The `ComponentContext.getServiceReferences` method MUST throw an `IllegalArgumentException` if
1482 the component does not have a service which implements the interface identified by the
1483 `businessInterface` parameter. [JCA80025]

1484

1485 **`createSelfReference(Class businessInterface, String serviceName)` method:**

1486 Returns a `ServiceReference` that can be used to invoke this component over the designated service. The
1487 `serviceName` parameter explicitly declares the service name to invoke

1488 Returns:

1489 • **`ServiceReference`** which is a `ServiceReference` proxy object for the reference, which implements
1490 the interface contained in the `businessInterface` parameter.

1491 The `ComponentContext.createSelfReference` method MUST return a `ServiceReference` object typed
1492 by the interface defined by the `businessInterface` parameter for the service identified by the
1493 `serviceName` of the invoking component and which has the interface defined by the `businessInterface`
1494 parameter. [JCA80026]

1495 Parameters:

1496 • **`Class businessInterface`** - the Java interface for the service reference

1497 • **`String serviceName`** - the name of the service reference

1498 Exceptions:

1499 • The `ComponentContext.createSelfReference` method MUST throw an `IllegalArgumentException` if the
1500 component does not have a service with the name identified by the `serviceName` parameter.
1501 [JCA80027]

- 1502 • The `ComponentContext.createSelfReference` method MUST throw an `IllegalArgumentException` if the
1503 component service with the name identified by the `serviceName` parameter does not implement a
1504 business interface which is compatible with the supplied `businessInterface` parameter. [JCA80028]

1505

1506 ***getProperty (Class type, String propertyName) method:***

1507 Returns the value of an SCA property defined by this component.

1508 Returns:

- 1509 • **** which is an object of the type identified by the type parameter containing the value specified for
1510 the property in the SCA configuration of the component. **null** if the SCA configuration of the
1511 component does not specify any value for the property.

1512 The `ComponentContext.getProperty` method MUST return an object of the type identified by the type
1513 parameter containing the value specified in the component configuration for the property named by
1514 the `propertyName` parameter or null if no value is specified in the configuration. [JCA80029]

1515 Parameters:

- 1516 • **Class type** - the Java class of the property (Object mapped type for primitive Java types - e.g.
1517 Integer if the type is int)
- 1518 • **String propertyName** - the name of the property

1519 Exceptions:

- 1520 • The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the
1521 component does not have a property with the name identified by the `propertyName` parameter.
1522 [JCA80030]
- 1523 • The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the
1524 component property with the name identified by the `propertyName` parameter does not have a type
1525 which is compatible with the supplied type parameter. [JCA80031]

1526

1527 ***getRequestContext() method:***

1528 Returns the `RequestContext` for the current SCA service request.

1529 Returns:

- 1530 • **RequestContext** which is the `RequestContext` object for the current SCA service invocation. **null** if
1531 there is no current request or if the context is unavailable.

1532 The `ComponentContext.getRequestContext` method MUST return non-null when invoked during the
1533 execution of a Java business method for a service operation or a callback operation, on the same
1534 thread that the SCA runtime provided, and MUST return null in all other cases. [JCA80002]

1535 Parameters:

- 1536 • **none**

1537 Exceptions:

- 1538 • **none**

1539

1540 ***cast(B target) method:***

1541 Casts a type-safe reference to a `ServiceReference`

1542 Returns:

- 1543 • **ServiceReference** which is a `ServiceReference` object which implements the same business
1544 interface B as a reference proxy object

1545 The `ComponentContext.cast` method MUST return a `ServiceReference` object which is typed by the
1546 same business interface as specified by the reference proxy object supplied in the `target` parameter.
1547 [JCA80032]

1548 Parameters:

1549 • **B target** - a type safe reference proxy object which implements the business interface B

1550 Exceptions:

1551 • **The ComponentContext.cast method MUST throw an IllegalArgumentException if the supplied target**
1552 **parameter is not an SCA reference proxy object. [JCA80033]**

1553 A component can access its component context by defining a field or setter method typed by
1554 **org.oasisopen.sca.ComponentContext** and annotated with **@Context**. To access a target service, the
1555 component uses **ComponentContext.getService(..)**.

1556 Snippet 9-1 shows an example of component context usage in a Java class using the @Context
1557 annotation.

```
1558     private ComponentContext componentContext;  
1559  
1560     @Context  
1561     public void setContext(ComponentContext context) {  
1562         componentContext = context;  
1563     }  
1564  
1565     public void doSomething() {  
1566         HelloWorld service =  
1567             componentContext.getService(HelloWorld.class, "HelloWorldComponent");  
1568         service.hello("hello");  
1569     }
```

1570 *Snippet 9-1: ComponentContext Injection Example*

1571 Similarly, non-SCA client code can use the ComponentContext API to perform operations against a
1572 component in an SCA domain. How the non-SCA client code obtains a reference to a ComponentContext
1573 is runtime specific.

1574 9.2 Request Context

1575 Figure 9-2 shows the **RequestContext** interface:

1576

```
1577     package org.oasisopen.sca;  
1578  
1579     import javax.security.auth.Subject;  
1580  
1581     public interface RequestContext {  
1582  
1583         Subject getSecuritySubject();  
1584  
1585         String getServiceName();  
1586         <CB> ServiceReference<CB> getCallbackReference();  
1587         <CB> CB getCallback();  
1588         <B> ServiceReference<B> getServiceReference();  
1589     }
```

1590 *Figure 9-2: RequestContext Interface*

1591

1592 **getSecuritySubject () method:**

1593 Returns the JAAS Subject of the current request (see [the JAAS Reference Guide \[JAAS\]](#) for details of
1594 JAAS).

1595 Returns:

1596 • **javax.security.auth.Subject** object which is the JAAS subject for the request.

1597 **null** if there is no subject for the request.

1598 The RequestContext.getSecuritySubject method MUST return the JAAS subject of the current
1599 request, or null if there is no subject or null if the method is invoked from code not processing a
1600 service request or callback request. [JCA80034]

1601 Parameters:

- 1602 • **none**

1603 Exceptions:

- 1604 • **none**

1605

1606 **getServiceName () method:**

1607 Returns the name of the service on the Java implementation the request came in on.

1608 Returns:

- 1609 • **String** containing the name of the service. **null** if the method is invoked from a thread that is not
1610 processing a service operation or a callback operation.

1611 The RequestContext.getServiceName method MUST return the name of the service for which an
1612 operation is being processed, or null if invoked from a thread that is not processing a service
1613 operation or a callback operation. [JCA80035]

1614 Parameters:

- 1615 • **none**

1616 Exceptions:

- 1617 • **none**

1618

1619 **getCallbackReference () method:**

1620 Returns a service reference proxy for the callback for the invoked service operation, as specified by the
1621 service client.

1622 Returns:

- 1623 • **ServiceReference<CB>** which is a service reference for the callback for the invoked service, as
1624 supplied by the service client. It is typed with the callback interface.
1625 **null** if the invoked service has an interface which is not bidirectional or if the getCallbackReference()
1626 method is called during the processing of a callback operation.
1627 **null** if the method is invoked from a thread that is not processing a service operation.

1628 The RequestContext.getCallbackReference method MUST return a ServiceReference object typed by
1629 the interface of the callback supplied by the client of the invoked service, or null if either the invoked
1630 service is not bidirectional or if the method is invoked from a thread that is not processing a service
1631 operation. [JCA80036]

1632 Parameters:

- 1633 • **none**

1634 Exceptions:

- 1635 • **none**

1636

1637 **getCallback () method:**

1638 Returns a proxy for the callback for the invoked service as specified by the service client.

1639 Returns:

- 1640 • **CB** proxy object for the callback for the invoked service as supplied by the service client. It is typed
1641 with the callback interface.

1642 **null** if the invoked service has an interface which is not bidirectional or if the `getCallback()` method is
1643 called during the processing of a callback operation.

1644 **null** if the method is invoked from a thread that is not processing a service operation.

1645 The `RequestContext.getCallback` method **MUST** return a reference proxy object typed by the
1646 interface of the callback supplied by the client of the invoked service, or **null** if either the invoked
1647 service is not bidirectional or if the method is invoked from a thread that is not processing a service
1648 operation. [JCA80037]

1649 Parameters:

- 1650 • **none**

1651 Exceptions:

- 1652 • **none**

1653

1654 ***getServiceReference () method:***

1655 Returns a `ServiceReference` object for the service that was invoked.

1656 Returns:

- 1657 • ***ServiceReference*** which is a service reference for the invoked service. It is typed with the
1658 interface of the service.

1659 **null** if the method is invoked from a thread that is not processing a service operation or a callback
1660 operation.

1661 When invoked during the execution of a service operation, the `RequestContext.getServiceReference`
1662 method **MUST** return a `ServiceReference` that represents the service that was invoked. [JCA80003]

1663 When invoked during the execution of a callback operation, the `RequestContext.getServiceReference`
1664 method **MUST** return a `ServiceReference` that represents the callback that was invoked. [JCA80038]

1665 When invoked from a thread not involved in the execution of either a service operation or of a
1666 callback operation, the `RequestContext.getServiceReference` method **MUST** return **null**. [JCA80039]

1667 Parameters:

- 1668 • **none**

1669 Exceptions:

- 1670 • **none**

1671 `ServiceReferences` can be injected using the `@Reference` annotation on a field, a setter method, or
1672 constructor parameter taking the type `ServiceReference`. The detailed description of the usage of these
1673 methods is described in the section on Asynchronous Programming in this document.

1674 9.3 ServiceReference Interface

1675 `ServiceReferences` can be injected using the `@Reference` annotation on a field, a setter method, or
1676 constructor parameter taking the type `ServiceReference`. The detailed description of the usage of these
1677 methods is described in the section on Asynchronous Programming in this document.

1678 Figure 9-3 defines the ***ServiceReference*** interface:

1679

```
1680 package org.oasisopen.sca;  
1681  
1682 public interface ServiceReference<B> extends java.io.Serializable {  
1683  
1684     B getService();  
1685     Class<B> getBusinessInterface();  
1686 }  
1687
```

1688 *Figure 9-3: ServiceReference Interface*

1689

1690 **getService () method:**

1691 Returns a type-safe reference to the target of this reference. The instance returned is guaranteed to
1692 implement the business interface for this reference. The value returned is a proxy to the target that
1693 implements the business interface associated with this reference.

1694 Returns:

- 1695 • **** which is type-safe reference proxy object to the target of this reference. It is typed with the
1696 interface of the target service.

1697 **The ServiceReference.getService method MUST return a reference proxy object which can be used**
1698 **to invoke operations on the target service of the reference and which is typed with the business**
1699 **interface of the reference. [JCA80040]**

1700 Parameters:

- 1701 • **none**

1702 Exceptions:

- 1703 • **none**

1704

1705 **getBusinessInterface () method:**

1706 Returns the Java class for the business interface associated with this ServiceReference.

1707 Returns:

- 1708 • **Class** which is a Class object of the business interface associated with the reference.

1709 **The ServiceReference.getBusinessInterface method MUST return a Class object representing the**
1710 **business interface of the reference. [JCA80041]**

1711 Parameters:

- 1712 • **none**

1713 Exceptions:

- 1714 • **none**

1715 **9.4 ResponseDispatch interface**

1716 The **ResponseDispatch** interface is shown in Figure 9-4:

1717

```
1718 package org.oasisopen.sca;  
1719  
1720 public interface ResponseDispatch<T> {  
1721     void sendResponse(T res);  
1722     void sendFault(Throwable e);  
1723     Map<String, Object> getContext();  
1724 }
```

1725 *Figure 9-4: ResponseDispatch Interface*

1726

1727 **sendResponse (T response) method:**

1728 Sends the response message from an asynchronous service method. This method can only be invoked
1729 once for a given ResponseDispatch object and cannot be invoked if sendFault has previously been
1730 invoked for the same ResponseDispatch object.

1731 Returns:

1732 • **void**
1733 The `ResponseDispatch.sendResponse()` method MUST send the response message to the client of
1734 an asynchronous service. [JCA50057]

1735 Parameters:

1736 • **T** - an instance of the response message returned by the service operation

1737 Exceptions:

1738 • The `ResponseDispatch.sendResponse()` method MUST throw an `InvalidStateException` if either the
1739 `sendResponse` method or the `sendFault` method has already been called once. [JCA80058]

1740

1741 ***sendFault (Throwable e) method:***

1742 Sends an exception as a fault from an asynchronous service method. This method can only be invoked
1743 once for a given `ResponseDispatch` object and cannot be invoked if `sendResponse` has previously been
1744 invoked for the same `ResponseDispatch` object.

1745 Returns:

1746 • **void**

1747 The `ResponseDispatch.sendFault()` method MUST send the supplied fault to the client of an
1748 asynchronous service. [JCA80059]

1749 Parameters:

1750 • **e** - an instance of an exception returned by the service operation

1751 Exceptions:

1752 • The `ResponseDispatch.sendFault()` method MUST throw an `InvalidStateException` if either the
1753 `sendResponse` method or the `sendFault` method has already been called once. [JCA80060]

1754

1755 ***getContext () method:***

1756 Obtains the context object for the `ResponseDispatch` method

1757 Returns:

1758 • **Map<String, object>** which is the context object for the `ResponseDispatch` object.
1759 The invoker can update the context object with appropriate context information, prior to invoking
1760 either the `sendResponse` method or the `sendFault` method

1761 Parameters:

1762 • **none**

1763 Exceptions:

1764 • **none**

1765 9.5 ServiceRuntimeException

1766 Figure 9-5 shows the ***ServiceRuntimeException***.

1767

```
1768 package org.oasisopen.sca;  
1769  
1770 public class ServiceRuntimeException extends RuntimeException {  
1771     ...  
1772 }
```

1773 *Figure 9-5: ServiceRuntimeException*

1774

1775 This exception signals problems in the management of SCA component execution.

1776 9.6 ServiceUnavailableException

1777 Figure 9-6 shows the *ServiceUnavailableException*.

1778

```
1779 package org.oasisopen.sca;
1780
1781 public class ServiceUnavailableException extends ServiceRuntimeException {
1782     ...
1783 }
```

1784 *Figure 9-6: ServiceUnavailableException*

1785

1786 This exception signals problems in the interaction with remote services. These are exceptions that can
1787 be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException that is *not* a
1788 ServiceUnavailableException is unlikely to be resolved by retrying the operation, since it most likely
1789 requires human intervention

1790 9.7 InvalidServiceException

1791 Figure 9-7 shows the *InvalidServiceException*.

1792

```
1793 package org.oasisopen.sca;
1794
1795 public class InvalidServiceException extends ServiceRuntimeException {
1796     ...
1797 }
```

1798 *Figure 9-7: InvalidServiceException*

1799

1800 This exception signals that the ServiceReference is no longer valid. This can happen when the target of
1801 the reference is undeployed. This exception is not transient and therefore is unlikely to be resolved by
1802 retrying the operation and will most likely require human intervention.

1803 9.8 Constants

1804 The SCA *Constants* interface defines a number of constant values that are used in the SCA Java APIs
1805 and Annotations. Figure 9-8 shows the Constants interface:

```
1806 package org.oasisopen.sca;
1807
1808 public interface Constants {
1809
1810     String SCA_NS = "http://docs.oasis-open.org/ns/opencsa/sca/200912";
1811
1812     String SCA_PREFIX = "{"+SCA_NS+"}";
1813
1814     String SERVERAUTHENTICATION = SCA_PREFIX + "serverAuthentication";
1815     String CLIENTAUTHENTICATION = SCA_PREFIX + "clientAuthentication";
1816     String ATLEASTONCE = SCA_PREFIX + "atLeastOnce";
1817     String ATMOSTONCE = SCA_PREFIX + "atMostOnce";
1818     String EXACTLYONCE = SCA_PREFIX + "exactlyOnce";
1819     String ORDERED = SCA_PREFIX + "ordered";
1820     String TRANSACTEDONEWAY = SCA_PREFIX + "transactedOneWay";
1821     String IMMEDIATEONEWAY = SCA_PREFIX + "immediateOneWay";
1822     String PROPAGATESTransaction = SCA_PREFIX + "propagatesTransaction";
1823     String SUSPENDSTransaction = SCA_PREFIX + "suspendsTransaction";
1824     String ASYNCINVOcation = SCA_PREFIX + "asyncInvocation";
1825     String SOAP = SCA_PREFIX + "SOAP";

```

```

1826 String JMS = SCA_PREFIX + "JMS";
1827 String NOLISTENER = SCA_PREFIX + "noListener";
1828 String EJB = SCA_PREFIX + "EJB";
1829
1830 }

```

1831 *Figure 9-8: Constants Interface*

1832 9.9 SCAClientFactory Class

1833 The SCAClientFactory class provides the means for client code to obtain a proxy reference object for a
 1834 service within an SCA Domain, through which the client code can invoke operations of that service. This
 1835 is particularly useful for client code that is running outside the SCA Domain containing the target service,
 1836 for example where the code is "unmanaged" and is not running under an SCA runtime.

1837 The SCAClientFactory is an abstract class which provides a set of static newInstance(...) methods which
 1838 the client can invoke in order to obtain a concrete object implementing the SCAClientFactory interface for
 1839 a particular SCA Domain. The returned SCAClientFactory object provides a getService() method which
 1840 provides the client with the means to obtain a reference proxy object for a service running in the SCA
 1841 Domain.

1842 The SCAClientFactory class is shown in Figure 9-9:

```

1843
1844 package org.oasisopen.sca.client;
1845
1846 import java.net.URI;
1847 import java.util.Properties;
1848
1849 import org.oasisopen.sca.NoSuchDomainException;
1850 import org.oasisopen.sca.NoSuchServiceException;
1851 import org.oasisopen.sca.client.SCAClientFactoryFinder;
1852 import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;
1853
1854 public abstract class SCAClientFactory {
1855
1856     protected static SCAClientFactoryFinder factoryFinder;
1857
1858     private URI domainURI;
1859
1860     private SCAClientFactory() {
1861     }
1862
1863     protected SCAClientFactory(URI domainURI)
1864         throws NoSuchDomainException {
1865         this.domainURI = domainURI;
1866     }
1867
1868     protected URI getDomainURI() {
1869         return domainURI;
1870     }
1871
1872     public static SCAClientFactory newInstance( URI domainURI )
1873         throws NoSuchDomainException {
1874         return newInstance( null, null, domainURI);
1875     }
1876
1877     public static SCAClientFactory newInstance(Properties properties,
1878                                             URI domainURI)
1879         throws NoSuchDomainException {
1880         return newInstance(properties, null, domainURI);
1881     }
1882
1883     public static SCAClientFactory newInstance(ClassLoader classLoader,

```

```

1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933

```

```

    throws NoSuchDomainException {
    return newInstance(null, classLoader, domainURI);
}

public static SCAClientFactory newInstance(Properties properties,
                                           ClassLoader classLoader,
                                           URI domainURI)
    throws NoSuchDomainException {
    final SCAClientFactoryFinder finder =
        factoryFinder != null ? factoryFinder :
        new SCAClientFactoryFinderImpl();
    final SCAClientFactory factory
        = finder.find(properties, classLoader, domainURI);
    return factory;
}

public abstract <T> T getService(Class<T> interfaze, String serviceURI)
    throws NoSuchServiceException, NoSuchDomainException;
}

```

Figure 9-9: SCAClientFactory Class

newInstance (URI domainURI) method:

Obtains a object implementing the SCAClientFactory class.

Returns:

- **object** which implements the SCAClientFactory class

The SCAClientFactory.newInstance(URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter. [JCA80042]

Parameters:

- **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

Exceptions:

- The SCAClientFactory.newInstance(URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain. [JCA80043]

newInstance(Properties properties, URI domainURI) method:

Obtains a object implementing the SCAClientFactory class, using a specified set of properties.

Returns:

- **object** which implements the SCAClientFactory class

The SCAClientFactory.newInstance(Properties, URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter. [JCA80044]

Parameters:

- **properties** - a set of Properties that can be used when creating the object which implements the SCAClientFactory class.
- **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

Exceptions:

- The SCAClientFactory.newInstance(Properties, URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain. [JCA80045]

1934 ***newInstance(Classloader classLoader, URI domainURI) method:***

1935 Obtains a object implementing the SCAClientFactory class using a specified classloader.

1936 Returns:

- 1937 • **object** which implements the SCAClientFactory class

1938 The SCAClientFactory.newInstance(Classloader, URI) method MUST return an object which
1939 implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
1940 [JCA80046]

1941 Parameters:

- 1942 • **classLoader** - a ClassLoader to use when creating the object which implements the
1943 SCAClientFactory class.
- 1944 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

1945 Exceptions:

- 1946 • The SCAClientFactory.newInstance(Classloader, URI) method MUST throw a
1947 NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
1948 [JCA80047]

1949

1950 ***newInstance(Properties properties, Classloader classLoader, URI domainURI) method:***

1951 Obtains a object implementing the SCAClientFactory class using a specified set of properties and a
1952 specified classloader.

1953 Returns:

- 1954 • **object** which implements the SCAClientFactory class

1955 The SCAClientFactory.newInstance(Properties, Classloader, URI) method MUST return an object
1956 which implements the SCAClientFactory class for the SCA Domain identified by the domainURI
1957 parameter. [JCA80048]

1958 Parameters:

- 1959 • **properties** - a set of Properties that can be used when creating the object which implements the
1960 SCAClientFactory class.
- 1961 • **classLoader** - a ClassLoader to use when creating the object which implements the
1962 SCAClientFactory class.
- 1963 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

1964 Exceptions:

- 1965 • The SCAClientFactory.newInstance(Properties, Classloader, URI) MUST throw a
1966 NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
1967 [JCA80049]

1968

1969 ***getService(Class<T> interfaze, String serviceURI) method:***

1970 Obtains a proxy reference object for a specified target service in a specified SCA Domain.

1971 Returns:

- 1972 • **<T>** a proxy object which implements the business interface T
1973 Invocations of a business method of the proxy causes the invocation of the corresponding operation
1974 of the target service.

1975 The SCAClientFactory.getService method MUST return a proxy object which implements the
1976 business interface defined by the interfaze parameter and which can be used to invoke operations on
1977 the service identified by the serviceURI parameter. [JCA80050]

1978 Parameters:

- 1979 • **interfaze** - a Java interface class which is the business interface of the target service

- 1980 • **serviceURI** - a String containing the relative URI of the target service within its SCA Domain.
1981 Takes the form componentName/serviceName or can also take the extended form
1982 componentName/serviceName/bindingName to use a specific binding of the target service

1983 Exceptions:

- 1984 • The `SCAClientFactory.getService` method MUST throw a `NoSuchServiceException` if a service with
1985 the relative URI `serviceURI` and a business interface which matches `interfaze` cannot be found in the
1986 SCA Domain targeted by the `SCAClient` object. [JCA80051]
1987 • The `SCAClientFactory.getService` method MUST throw a `NoSuchServiceException` if the `domainURI`
1988 of the `SCAClientFactory` does not identify a valid SCA Domain. [JCA80052]

1989
1990 **SCAClientFactory (URI) method:** a single argument constructor that must be available on all concrete
1991 subclasses of `SCAClientFactory`. The URI required is the URI of the Domain targeted by the
1992 `SCAClientFactory`

1993

1994 **getDomainURI() method:**

1995 Obtains the Domain URI value for this `SCAClientFactory`

1996 Returns:

- 1997 • **URI** of the target SCA Domain for this `SCAClientFactory`
1998 The `SCAClientFactory.getDomainURI` method MUST return the SCA Domain URI of the Domain
1999 associated with the `SCAClientFactory` object. [JCA80053]

2000 Parameters:

- 2001 • **none**

2002 Exceptions:

- 2003 • The `SCAClientFactory.getDomainURI` method MUST throw a **NoSuchServiceException** if the
2004 `domainURI` of the `SCAClientFactory` does not identify a valid SCA Domain. [JCA80054]

2005

2006 **private SCAClientFactory() method:**

2007 This private no-argument constructor prevents instantiation of an `SCAClientFactory` instance without the
2008 use of the constructor with an argument, even by subclasses of the abstract `SCAClientFactory` class.

2009

2010 **factoryFinder protected field:**

2011 Provides a means by which a provider of an `SCAClientFactory` implementation can inject a factory finder
2012 implementation into the abstract `SCAClientFactory` class - once this is done, future invocations of the
2013 `SCAClientFactory` use the injected factory finder to locate and return an instance of a subclass of
2014 `SCAClientFactory`.

2015 9.10 SCAClientFactoryFinder Interface

2016 The `SCAClientFactoryFinder` interface is a Service Provider Interface representing a `SCAClientFactory`
2017 finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can
2018 create alternative implementations of this interface that use different class loading or lookup mechanisms:

2019

```
2020 package org.oasisopen.sca.client;  
2021  
2022 public interface SCAClientFactoryFinder {  
2023     SCAClientFactory find(Properties properties,  
2024                          ClassLoader classLoader,  
2025
```



```

2026         URI domainURI )
2027     throws NoSuchDomainException ;
2028 }

```

Figure 9-10: SCAClientFactoryFinder Interface

find (Properties properties, ClassLoader classloader, URI domainURI) method:

Obtains an implementation of the SCAClientFactory interface.

Returns:

- **SCAClientFactory** implementation object
- The implementation of the SCAClientFactoryFinder.find method MUST return an object which is an implementation of the SCAClientFactory interface, for the SCA Domain represented by the doaminURI parameter, using the supplied properties and classloader. [JCA80055]

Parameters:

- **properties** - a set of Properties that can be used when creating the object which implements the SCAClientFactory interface.
- **classLoader** - a ClassLoader to use when creating the object which implements the SCAClientFactory interface.
- **domainURI** - a URI for the SCA Domain targeted by the SCAClientFactory

Exceptions:

- The implementation of the SCAClientFactoryFinder.find method MUST throw a ServiceRuntimeException if the SCAClientFactory implementation could not be found. [JCA80056]

9.11 SCAClientFactoryFinderImpl Class

This class is a default implementation of an SCAClientFactoryFinder, which is used to find an implementation of an SCAClientFactory subclass, as used to obtain an SCAClient object for use by a client. SCA runtime providers can replace this implementation with their own version.

```

package org.oasisopen.sca.client.impl;

public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {
    ...
    public SCAClientFactoryFinderImpl() {...}

    public SCAClientFactory find(Properties properties,
                               ClassLoader classLoader
                               URI domainURI)
    throws NoSuchDomainException, ServiceRuntimeException {...}
    ...
}

```

Snippet 9-2: SCAClientFactoryFinderImpl Class

SCAClientFactoryFinderImpl () method:

Public constructor for the SCAClientFactoryFinderImpl.

Returns:

- **SCAClientFactoryFinderImpl** which implements the SCAClientFactoryFinder interface

Parameters:

- **none**

2072 Exceptions:

- 2073 • **none**

2074

2075 **find (Properties, ClassLoader, URI) method:**

2076 Obtains an implementation of the SCAClientFactory interface. It discovers a provider's SCAClientFactory
2077 implementation by referring to the following information in this order:

- 2078 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the
2079 newInstance() method call if specified
- 2080 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 2081 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

2082 Returns:

- 2083 • **SCAClientFactory** implementation object

2084 Parameters:

- 2085 • **properties** - a set of Properties that can be used when creating the object which implements the
2086 SCAClientFactory interface.
- 2087 • **classLoader** - a ClassLoader to use when creating the object which implements the
2088 SCAClientFactory interface.
- 2089 • **domainURI** - a URI for the SCA Domain targeted by the SCAClientFactory

2090 Exceptions:

- 2091 • **ServiceRuntimeException** - if the SCAClientFactory implementation could not be found

2092 9.12 NoSuchDomainException

2093 Figure 9-11 shows the **NoSuchDomainException**:

2094

```
2095 package org.oasisopen.sca;  
2096  
2097 public class NoSuchDomainException extends Exception {  
2098     ...  
2099 }
```

2100 *Figure 9-11: NoSuchDomainException Class*

2101

2102 This exception indicates that the Domain specified could not be found.

2103 9.13 NoSuchServiceException

2104 Figure 9-12 shows the **NoSuchServiceException**:

2105

```
2106 package org.oasisopen.sca;  
2107  
2108 public class NoSuchServiceException extends Exception {  
2109     ...  
2110 }
```

2111 *Figure 9-12: NoSuchServiceException Class*

2112

2113 This exception indicates that the service specified could not be found.

2114 10 Java Annotations

2115 This section provides definitions of all the Java annotations which apply to SCA.

2116 This specification places constraints on some annotations that are not detectable by a Java compiler. For
2117 example, the definition of the `@Property` and `@Reference` annotations indicate that they are allowed on
2118 parameters, but the sections "`@Property`" and "`@Reference`" constrain those definitions to constructor
2119 parameters. An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is
2120 improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation
2121 code. [JCA90001]

2122 SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an SCA
2123 annotation on a static method or a static field of an implementation class and the SCA runtime MUST
2124 NOT instantiate such an implementation class. [JCA90002]

2125 10.1 @AllowsPassByReference

2126 Figure 10-1 defines the `@AllowsPassByReference` annotation:

2127

```
2128 package org.oasisopen.sca.annotation;  
2129  
2130 import static java.lang.annotation.ElementType.FIELD;  
2131 import static java.lang.annotation.ElementType.METHOD;  
2132 import static java.lang.annotation.ElementType.PARAMETER;  
2133 import static java.lang.annotation.ElementType.TYPE;  
2134 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2135 import java.lang.annotation.Retention;  
2136 import java.lang.annotation.Target;  
2137  
2138 @Target({TYPE, METHOD, FIELD, PARAMETER})  
2139 @Retention(RUNTIME)  
2140 public @interface AllowsPassByReference {  
2141  
2142     boolean value() default true;  
2143 }  
2144
```

2144 Figure 10-1: `AllowsPassByReference` Annotation

2145

2146 The `@AllowsPassByReference` annotation allows service method implementations and client references
2147 to be marked as “allows pass by reference” to indicate that they use input parameters, return values and
2148 exceptions in a manner that allows the SCA runtime to avoid the cost of copying mutable objects when a
2149 remotable service is called locally within the same JVM.

2150 The `@AllowsPassByReference` annotation has the attribute:

- 2151 • **value** – specifies whether the “allows pass by reference” marker applies to the service
2152 implementation class, service implementation method, or client reference to which this annotation
2153 applies; if not specified, defaults to true.

2154 The `@AllowsPassByReference` annotation MUST only annotate the following locations:

- 2155 • a service implementation class
- 2156 • an individual method of a remotable service implementation
- 2157 • an individual reference which uses a remotable interface, where the reference is a field, a setter
2158 method, or a constructor parameter [JCA90052]

2159 The “allows pass by reference” marking of a method implementation of a remotable service is determined
2160 as follows:

- 2161 1. If the method has an `@AllowsPassByReference` annotation, the method is marked “allows pass by
2162 reference” if and only if the value of the method’s annotation is true.
- 2163 2. Otherwise, if the class has an `@AllowsPassByReference` annotation, the method is marked “allows
2164 pass by reference” if and only if the value of the class’s annotation is true.
- 2165 3. Otherwise, the method is not marked “allows pass by reference”.
- 2166 The “allows pass by reference” marking of a reference for a remotable service is determined as follows:
- 2167 1. If the reference has an `@AllowsPassByReference` annotation, the reference is marked “allows pass
2168 by reference” if and only if the value of the reference’s annotation is true.
- 2169 2. Otherwise, if the service implementation class containing the reference has an
2170 `@AllowsPassByReference` annotation, the reference is marked “allows pass by reference” if and only
2171 if the value of the class’s annotation is true.
- 2172 3. Otherwise, the reference is not marked “allows pass by reference”.
- 2173 Snippet 10-1 shows a sample where `@AllowsPassByReference` is defined for the implementation of a
2174 service method on the Java component implementation class.

2175

```
2176 @AllowsPassByReference  
2177 public String hello(String message) {  
2178     ...  
2179 }
```

2180 *Snippet 10-1: Use of `@AllowsPassByReference` on a Method*

2181

2182 Snippet 10-2 shows a sample where `@AllowsPassByReference` is defined for a client reference of a Java
2183 component implementation class.

2184

```
2185 @AllowsPassByReference  
2186 @Reference  
2187 private StockQuoteService stockQuote;
```

2188 *Snippet 10-2: Use of `@AllowsPassByReference` on a Reference*

2189 **10.2 @AsyncFault**

2190 Figure 10-2 defines the `@AsyncFault` annotation:

2191

```
2192 package org.oasisopen.sca.annotation;  
2193  
2194 import static java.lang.annotation.ElementType.METHOD;  
2195 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2196  
2197 import java.lang.annotation.Inherited;  
2198 import java.lang.annotation.Retention;  
2199 import java.lang.annotation.Target;  
2200  
2201 @Inherited  
2202 @Target({METHOD})  
2203 @Retention(RUNTIME)  
2204 public @interface AsyncFault {  
2205  
2206     Class<?>[] value() default {};  
2207  
2208 }
```

2209 *Figure 10-2: AsyncFault Annotation*

2210

2211 The **@AsyncFault** annotation is used to indicate the faults/exceptions which are returned by the
2212 asynchronous service method which it annotates.

2213 **10.3 @AsyncInvocation**

2214 Figure 10-3 defines the **@AsyncInvocation** annotation, which is used to attach the "asyncInvocation"
2215 policy intent to an interface or to a method:

2216

```
2217 package org.oasisopen.sca.annotation;  
2218  
2219 import static java.lang.annotation.ElementType.METHOD;  
2220 import static java.lang.annotation.ElementType.TYPE;  
2221 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2222 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2223  
2224 import java.lang.annotation.Inherited;  
2225 import java.lang.annotation.Retention;  
2226 import java.lang.annotation.Target;  
2227  
2228 @Inherited  
2229 @Target({TYPE, METHOD})  
2230 @Retention(RUNTIME)  
2231 @Intent(AsyncInvocation.ASYNCINVOCATION)  
2232 public @interface AsyncInvocation {  
2233     String ASYNCINVOCATION = SCA_PREFIX + "asyncInvocation";  
2234  
2235     boolean value() default true;  
2236 }
```

2237 *Figure 10-3: AsyncInvocation Annotation*

2238

2239 The **@AsyncInvocation** annotation is used to indicate that the operations of a Java interface uses the
2240 long-running request-response pattern as described in the SCA Assembly specification.

2241 **10.4 @Authentication**

2242 The following Java code defines the **@Authentication** annotation:

2243

```
2244 package org.oasisopen.sca.annotation;  
2245  
2246 import static java.lang.annotation.ElementType.FIELD;  
2247 import static java.lang.annotation.ElementType.METHOD;  
2248 import static java.lang.annotation.ElementType.PARAMETER;  
2249 import static java.lang.annotation.ElementType.TYPE;  
2250 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2251 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2252  
2253 import java.lang.annotation.Inherited;  
2254 import java.lang.annotation.Retention;  
2255 import java.lang.annotation.Target;  
2256  
2257 @Inherited  
2258 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2259 @Retention(RUNTIME)  
2260 @Intent(Authentication.AUTHENTICATION)  
2261 public @interface Authentication {  
2262     String AUTHENTICATION = SCA_PREFIX + "authentication";  
2263     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";  
2264     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";  
2265 }
```

```

2265
2266     /**
2267     * List of authentication qualifiers (such as "message"
2268     * or "transport").
2269     *
2270     * @return authentication qualifiers
2271     */
2272     @Qualifier
2273     String[] value() default "";
2274 }

```

2275 *Figure 10-4: Authentication Annotation*

2276
2277 The **@Authentication** annotation is used to indicate the need for authentication. See the SCA Policy
2278 Framework Specification [POLICY] for details on the meaning of the intent. See the [section on](#)
2279 [Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

2280 **10.5 @Authorization**

2281 Figure 10-5 defines the **@Authorization** annotation:

```

2282
2283 package org.oasisopen.sca.annotation;
2284
2285 import static java.lang.annotation.ElementType.FIELD;
2286 import static java.lang.annotation.ElementType.METHOD;
2287 import static java.lang.annotation.ElementType.PARAMETER;
2288 import static java.lang.annotation.ElementType.TYPE;
2289 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2290 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2291
2292 import java.lang.annotation.Inherited;
2293 import java.lang.annotation.Retention;
2294 import java.lang.annotation.Target;
2295
2296 /**
2297 * The @Authorization annotation is used to indicate that
2298 * an authorization policy is required.
2299 */
2300 @Inherited
2301 @Target({TYPE, FIELD, METHOD, PARAMETER})
2302 @Retention(RUNTIME)
2303 @Intent(Authorization.AUTHORIZATION)
2304 public @interface Authorization {
2305     String AUTHORIZATION = SCA_PREFIX + "authorization";
2306 }

```

2307 *Figure 10-5: Authorization Annotation*

2308
2309 The **@Authorization** annotation is used to indicate the need for an authorization policy. See the SCA
2310 Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on](#)
2311 [Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

2312 **10.6 @Callback**

2313 Figure 10-6 defines the **@Callback** annotation:

```

2314
2315 package org.oasisopen.sca.annotation;

```

```

2316
2317 import static java.lang.annotation.ElementType.FIELD;
2318
2319 import static java.lang.annotation.ElementType.METHOD;
2319 import static java.lang.annotation.ElementType.TYPE;
2320 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2321 import java.lang.annotation.Retention;
2322 import java.lang.annotation.Target;
2323
2324 @Target({TYPE, METHOD, FIELD})
2325 @Retention(RUNTIME)
2326 public @interface Callback {
2327
2328     Class<?> value() default Void.class;
2329 }

```

Figure 10-6: Callback Annotation

The @Callback annotation is used to annotate a service interface or to annotate a Java class (used to define an interface) with a callback interface by specifying the Java class object of the callback interface as an attribute.

The @Callback annotation has the attribute:

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

The @Callback annotation can also be used to annotate a method or a field of an SCA implementation class, in order to have a callback object injected. When used to annotate a method or a field of an implementation class for injection of a callback object, the @Callback annotation MUST NOT specify any attributes. [JCA90046] When used to annotate a method or a field of an implementation class for injection of a callback object, the type of the method or field MUST be the callback interface of at least one bidirectional service offered by the implementation class. [JCA90054] When used to annotate a setter method or a field of an implementation class for injection of a callback object, the SCA runtime MUST inject a callback reference proxy into that method or field when the Java class is initialized, if the component is invoked via a service which has a callback interface and where the type of the setter method or field corresponds to the type of the callback interface. [JCA90058]

The @Callback annotation MUST NOT appear on a setter method or a field of a Java implementation class that has COMPOSITE scope. [JCA90057]

Snippet 10-3 shows an example use of the @Callback annotation to declare a callback interface.

```

2350
2351 package somepackage;
2352 import org.oasisopen.sca.annotation.Callback;
2353 import org.oasisopen.sca.annotation.Remotable;
2354 @Remotable
2355 @Callback(MyServiceCallback.class)
2356 public interface MyService {
2357
2358     void someMethod(String arg);
2359 }
2360
2361 @Remotable
2362 public interface MyServiceCallback {
2363
2364     void receiveResult(String result);
2365 }

```

Snippet 10-3: Use of @Callback

The implied component type is for Snippet 10-3 is shown in Snippet 10-4.

2369
2370
2371
2372
2373
2374
2375
2376

```
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" >  
  <service name="MyService">  
    <interface.java interface="somepackage.MyService"  
      callbackInterface="somepackage.MyServiceCallback" />  
  </service>  
</componentType>
```

2377 *Snippet 10-4: Implied componentType for Snippet 10-3*

2378 **10.7 @ComponentName**

2379 Figure 10-7 defines the **@ComponentName** annotation:

2380

2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393

```
package org.oasisopen.sca.annotation;  
  
import static java.lang.annotation.ElementType.FIELD;  
import static java.lang.annotation.ElementType.METHOD;  
import static java.lang.annotation.RetentionPolicy.RUNTIME;  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
  
@Target({METHOD, FIELD})  
@Retention(RUNTIME)  
public @interface ComponentName {  
}
```

2394 *Figure 10-7: ComponentName Annotation*

2395

2396 The **@ComponentName** annotation is used to denote a Java class field or setter method that is used to
2397 inject the component name.

2398 Snippet 10-5 shows a component name field definition sample.

2399

2400
2401

```
@ComponentName  
private String componentName;
```

2402 *Snippet 10-5: Use of @ComponentName on a Field*

2403

2404 Snippet 10-6 shows a component name setter method sample.

2405

2406
2407
2408
2409

```
@ComponentName  
public void setComponentName(String name) {  
  //...  
}
```

2410 *Snippet 10-6: Use of @ComponentName on a Setter*

2411 **10.8 @Confidentiality**

2412 Figure 10-8 defines the **@Confidentiality** annotation:

2413

2414
2415
2416

```
package org.oasisopen.sca.annotation;  
  
import static java.lang.annotation.ElementType.FIELD;
```



```

2417 import static java.lang.annotation.ElementType.METHOD;
2418 import static java.lang.annotation.ElementType.PARAMETER;
2419 import static java.lang.annotation.ElementType.TYPE;
2420 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2421 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2422
2423 import java.lang.annotation.Inherited;
2424 import java.lang.annotation.Retention;
2425 import java.lang.annotation.Target;
2426
2427 @Inherited
2428 @Target({TYPE, FIELD, METHOD, PARAMETER})
2429 @Retention(RUNTIME)
2430 @Intent(Confidentiality.CONFIDENTIALITY)
2431 public @interface Confidentiality {
2432     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
2433     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
2434     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
2435
2436     /**
2437      * List of confidentiality qualifiers such as "message" or
2438      * "transport".
2439      *
2440      * @return confidentiality qualifiers
2441      */
2442     @Qualifier
2443     String[] value() default "";
2444 }

```

Figure 10-8: Confidentiality Annotation

The **@Confidentiality** annotation is used to indicate the need for confidentiality. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

10.9 @Constructor

Figure 10-9 defines the **@Constructor** annotation:

```

2453 package org.oasisopen.sca.annotation;
2454
2455 import static java.lang.annotation.ElementType.CONSTRUCTOR;
2456 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2457 import java.lang.annotation.Retention;
2458 import java.lang.annotation.Target;
2459
2460 @Target(CONSTRUCTOR)
2461 @Retention(RUNTIME)
2462 public @interface Constructor { }

```

Figure 10-9: Constructor Annotation

The **@Constructor** annotation is used to mark a particular constructor to use when instantiating a Java component implementation. **If a constructor of an implementation class is annotated with @Constructor and the constructor has parameters, each of these parameters MUST have either a @Property annotation or a @Reference annotation. [JCA90003]**

Snippet 10-7 shows a sample for the **@Constructor** annotation.

```

2470
2471 public class HelloServiceImpl implements HelloService {
2472
2473     public HelloServiceImpl(){
2474         ...
2475     }
2476
2477     @Constructor
2478     public HelloServiceImpl(@Property(name="someProperty")
2479                             String someProperty ){
2480         ...
2481     }
2482
2483     public String hello(String message) {
2484         ...
2485     }
2486 }

```

2487 *Snippet 10-7: Use of @Constructor*

2488 10.10 @Context

2489 Figure 10-10 defines the **@Context** annotation:

2490

```

2491 package org.oasisopen.sca.annotation;
2492
2493 import static java.lang.annotation.ElementType.FIELD;
2494 import static java.lang.annotation.ElementType.METHOD;
2495 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2496 import java.lang.annotation.Retention;
2497 import java.lang.annotation.Target;
2498
2499 @Target({METHOD, FIELD})
2500 @Retention(RUNTIME)
2501 public @interface Context {
2502
2503 }

```

2504 *Figure 10-10: Context Annotation*

2505

2506 The @Context annotation is used to denote a Java class field or a setter method that is used to inject a
2507 composite context for the component. The type of context to be injected is defined by the type of the Java
2508 class field or type of the setter method input argument; the type is either **ComponentContext** or
2509 **RequestContext**.

2510 The @Context annotation has no attributes.

2511 Snippet 10-8 shows a ComponentContext field definition sample.

2512

```

2513 @Context
2514 protected ComponentContext context;

```

2515 *Snippet 10-8: Use of @Context for a ComponentContext*

2516

2517 Snippet 10-9 shows a RequestContext field definition sample.

2518

```

2519 @Context
2520 protected RequestContext context;

```

2521 Snippet 10-9: Use of @Context for a RequestContext

2522 10.11 @Destroy

2523 Figure 10-11 defines the **@Destroy** annotation:

2524

```
2525 package org.oasisopen.sca.annotation;
2526
2527 import static java.lang.annotation.ElementType.METHOD;
2528 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2529 import java.lang.annotation.Retention;
2530 import java.lang.annotation.Target;
2531
2532 @Target(METHOD)
2533 @Retention(RUNTIME)
2534 public @interface Destroy {
2535
2536 }
```

2537 *Figure 10-11: Destroy Annotation*

2538

2539 The @Destroy annotation is used to denote a single Java class method that will be called when the scope
2540 defined for the implementation class ends. A method annotated with @Destroy can have any access
2541 modifier and MUST have a void return type and no arguments. [JCA90004]

2542 If there is a method annotated with @Destroy that matches the criteria for the annotation, the SCA
2543 runtime MUST call the annotated method when the scope defined for the implementation class ends.
2544 [JCA90005]

2545 Snippet 10-10 shows a sample for a destroy method definition.

2546

```
2547 @Destroy
2548 public void myDestroyMethod() {
2549     ...
2550 }
```

2551 *Snippet 10-10: Use of @Destroy*

2552 10.12 @EagerInit

2553 Figure 10-12: EagerInit Annotation defines the **@EagerInit** annotation:

2554

```
2555 package org.oasisopen.sca.annotation;
2556
2557 import static java.lang.annotation.ElementType.TYPE;
2558 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2559 import java.lang.annotation.Retention;
2560 import java.lang.annotation.Target;
2561
2562 @Target(TYPE)
2563 @Retention(RUNTIME)
2564 public @interface EagerInit {
2565
2566 }
```

2567 *Figure 10-12: EagerInit Annotation*

2568

2569 The **@EagerInit** annotation is used to mark the Java class of a COMPOSITE scoped implementation for
2570 eager initialization. When marked for eager initialization with an @EagerInit annotation, the composite
2571 scoped instance MUST be created when its containing component is started. [JCA90007]

2572 10.13 @Init

2573 Figure 10-13: Init Annotation defines the **@Init** annotation:
2574

```
2575 package org.oasisopen.sca.annotation;  
2576  
2577 import static java.lang.annotation.ElementType.METHOD;  
2578 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2579 import java.lang.annotation.Retention;  
2580 import java.lang.annotation.Target;  
2581  
2582 @Target(METHOD)  
2583 @Retention(RUNTIME)  
2584 public @interface Init {  
2585  
2586  
2587 }
```

2588 *Figure 10-13: Init Annotation*

2589
2590 The @Init annotation is used to denote a single Java class method that is called when the scope defined
2591 for the implementation class starts. A method marked with the @Init annotation can have any access
2592 modifier and MUST have a void return type and no arguments. [JCA90008]

2593 If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime
2594 MUST call the annotated method after all property and reference injection is complete. [JCA90009]

2595 Snippet 10-11 shows an example of an init method definition.

```
2596  
2597 @Init  
2598 public void myInitMethod() {  
2599     ...  
2600 }
```

2601 *Snippet 10-11: Use of @Init*

2602 10.14 @Integrity

2603 Figure 10-14 defines the **@Integrity** annotation:
2604

```
2605 package org.oasisopen.sca.annotation;  
2606  
2607 import static java.lang.annotation.ElementType.FIELD;  
2608 import static java.lang.annotation.ElementType.METHOD;  
2609 import static java.lang.annotation.ElementType.PARAMETER;  
2610 import static java.lang.annotation.ElementType.TYPE;  
2611 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2612 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2613  
2614 import java.lang.annotation.Inherited;  
2615 import java.lang.annotation.Retention;  
2616 import java.lang.annotation.Target;  
2617  
2618 @Inherited
```

```

2619 @Target({TYPE, FIELD, METHOD, PARAMETER})
2620 @Retention(RUNTIME)
2621 @Intent(Integrity.INTEGRITY)
2622 public @interface Integrity {
2623     String INTEGRITY = SCA_PREFIX + "integrity";
2624     String INTEGRITY_MESSAGE = INTEGRITY + ".message";
2625     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
2626
2627     /**
2628      * List of integrity qualifiers (such as "message" or "transport").
2629      *
2630      * @return integrity qualifiers
2631      */
2632     @Qualifier
2633     String[] value() default "";
2634 }

```

Figure 10-14: Integrity Annotation

The **@Integrity** annotation is used to indicate that the invocation requires integrity (i.e. no tampering of the messages between client and service). See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

10.15 @Intent

Figure 10-15 defines the **@Intent** annotation:

```

2644 package org.oasisopen.sca.annotation;
2645
2646 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
2647 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2648 import java.lang.annotation.Retention;
2649 import java.lang.annotation.Target;
2650
2651 @Target({ANNOTATION_TYPE})
2652 @Retention(RUNTIME)
2653 public @interface Intent {
2654     /**
2655      * The qualified name of the intent, in the form defined by
2656      * {@link javax.xml.namespace.QName#toString}.
2657      * @return the qualified name of the intent
2658      */
2659     String value() default "";
2660
2661     /**
2662      * The XML namespace for the intent.
2663      * @return the XML namespace for the intent
2664      */
2665     String targetNamespace() default "";
2666
2667     /**
2668      * The name of the intent within its namespace.
2669      * @return name of the intent within its namespace
2670      */
2671     String localPart() default "";
2672 }

```

Figure 10-15: Intent Annotation

2675 The @Intent annotation is used for the creation of new annotations for specific intents. It is not expected
2676 that the @Intent annotation will be used in application code.
2677 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to define new
2678 intent annotations.

2679 10.16 @ManagedSharedTransaction

2680 Figure 10-16 defines the @ManagedSharedTransaction annotation:

2681

```
2682 package org.oasisopen.sca.annotation;  
2683  
2684 import static java.lang.annotation.ElementType.FIELD;  
2685 import static java.lang.annotation.ElementType.METHOD;  
2686 import static java.lang.annotation.ElementType.PARAMETER;  
2687 import static java.lang.annotation.ElementType.TYPE;  
2688 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2689 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2690  
2691 import java.lang.annotation.Inherited;  
2692 import java.lang.annotation.Retention;  
2693 import java.lang.annotation.Target;  
2694  
2695 /**  
2696  * The @ManagedSharedTransaction annotation is used to indicate that  
2697  * a distributed ACID transaction is required.  
2698  */  
2699 @Inherited  
2700 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2701 @Retention(RUNTIME)  
2702 @Intent(ManagedSharedTransaction.MANAGEDSHAREDTRANSACTION)  
2703 public @interface ManagedSharedTransaction {  
2704     String MANAGEDSHAREDTRANSACTION = SCA_PREFIX + "managedSharedTransaction";  
2705 }
```

2706 *Figure 10-16: ManagedSharedTransaction Annotation*

2707

2708 The **@ManagedSharedTransaction** annotation is used to indicate the need for a distributed and globally
2709 coordinated ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the
2710 meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent
2711 annotations are used in Java.

2712 10.17 @ManagedTransaction

2713 Figure 10-17 defines the @ManagedTransaction annotation:

2714

```
2715 import static java.lang.annotation.ElementType.FIELD;  
2716 import static java.lang.annotation.ElementType.METHOD;  
2717 import static java.lang.annotation.ElementType.PARAMETER;  
2718 import static java.lang.annotation.ElementType.TYPE;  
2719 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2720 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2721  
2722 import java.lang.annotation.Inherited;  
2723 import java.lang.annotation.Retention;  
2724 import java.lang.annotation.Target;  
2725  
2726 /**  
2727  * The @ManagedTransaction annotation is used to indicate the
```

```

2728 * need for an ACID transaction environment.
2729 */
2730 @Inherited
2731 @Target({TYPE, FIELD, METHOD, PARAMETER})
2732 @Retention(RUNTIME)
2733 @Intent(ManagedTransaction.MANAGEDTRANSACTION)
2734 public @interface ManagedTransaction {
2735     String MANAGEDTRANSACTION = SCA_PREFIX + "managedTransaction";
2736     String MANAGEDTRANSACTION_MESSAGE = MANAGEDTRANSACTION + ".local";
2737     String MANAGEDTRANSACTION_TRANSPORT = MANAGEDTRANSACTION + ".global";
2738
2739     /**
2740      * List of managedTransaction qualifiers (such as "global" or "local").
2741      *
2742      * @return managedTransaction qualifiers
2743      */
2744     @Qualifier
2745     String[] value() default "";
2746 }

```

Figure 10-17: ManagedTransaction Annotation

The **@ManagedTransaction** annotation is used to indicate the need for an ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

10.18 @MutualAuthentication

Figure 10-18 defines the @MutualAuthentication annotation:

```

2755 package org.oasisopen.sca.annotation;
2756
2757 import static java.lang.annotation.ElementType.FIELD;
2758 import static java.lang.annotation.ElementType.METHOD;
2759 import static java.lang.annotation.ElementType.PARAMETER;
2760 import static java.lang.annotation.ElementType.TYPE;
2761 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2762 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2763
2764 import java.lang.annotation.Inherited;
2765 import java.lang.annotation.Retention;
2766 import java.lang.annotation.Target;
2767
2768 /**
2769  * The @MutualAuthentication annotation is used to indicate that
2770  * a mutual authentication policy is needed.
2771  */
2772 @Inherited
2773 @Target({TYPE, FIELD, METHOD, PARAMETER})
2774 @Retention(RUNTIME)
2775 @Intent(MutualAuthentication.MUTUALAUTHENTICATION)
2776 public @interface MutualAuthentication {
2777     String MUTUALAUTHENTICATION = SCA_PREFIX + "mutualAuthentication";
2778 }

```

Figure 10-18: MutualAuthentication Annotation

The **@MutualAuthentication** annotation is used to indicate the need for mutual authentication between a service consumer and a service provider. See the SCA Policy Framework Specification [POLICY] for

2783 details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of
2784 how intent annotations are used in Java.

2785 10.19 @NoManagedTransaction

2786 Figure 10-19 defines the @NoManagedTransaction annotation:
2787

```
2788 package org.oasisopen.sca.annotation;  
2789  
2790 import static java.lang.annotation.ElementType.FIELD;  
2791 import static java.lang.annotation.ElementType.METHOD;  
2792 import static java.lang.annotation.ElementType.PARAMETER;  
2793 import static java.lang.annotation.ElementType.TYPE;  
2794 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2795 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2796  
2797 import java.lang.annotation.Inherited;  
2798 import java.lang.annotation.Retention;  
2799 import java.lang.annotation.Target;  
2800  
2801 /**  
2802  * The @NoManagedTransaction annotation is used to indicate that  
2803  * a non-transactional environment is needed.  
2804  */  
2805 @Inherited  
2806 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2807 @Retention(RUNTIME)  
2808 @Intent(NoManagedTransaction.NOMANAGEDTRANSACTION)  
2809 public @interface NoManagedTransaction {  
2810     String NOMANAGEDTRANSACTION = SCA_PREFIX + "noManagedTransaction";  
2811 }
```

2812 *Figure 10-19: NoManagedTransaction Annotation*

2813

2814 The **@NoManagedTransaction** annotation is used to indicate that the component does not want to run in
2815 an ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the meaning
2816 of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations
2817 are used in Java.

2818 10.20 @OneWay

2819 Figure 10-20 defines the @OneWay annotation:
2820

```
2821 package org.oasisopen.sca.annotation;  
2822  
2823 import static java.lang.annotation.ElementType.METHOD;  
2824 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2825 import java.lang.annotation.Retention;  
2826 import java.lang.annotation.Target;  
2827  
2828 @Target(METHOD)  
2829 @Retention(RUNTIME)  
2830 public @interface OneWay {  
2831  
2832  
2833 }
```

2834 *Figure 10-20: OneWay Annotation*

2835

2836 A method annotated with `@OneWay` MUST have a void return type and MUST NOT have declared
2837 checked exceptions. [JCA90055]

2838 When a method of a Java interface is annotated with `@OneWay`, the SCA runtime MUST ensure that all
2839 invocations of that method are executed in a non-blocking fashion, as described in the section on
2840 Asynchronous Programming. [JCA90056]

2841 The `@OneWay` annotation has no attributes.

2842 Snippet 10-12 shows the use of the `@OneWay` annotation on an interface.

2843

```
2844 package services.hello;
2845
2846 import org.oasisopen.sca.annotation.OneWay;
2847
2848 public interface HelloService {
2849     @OneWay
2850     void hello(String name);
2851 }
```

2852 *Snippet 10-12: Use of `@OneWay`*

2853 **10.21 @PolicySets**

2854 Figure 10-21 defines the `@PolicySets` annotation:

2855

```
2856 package org.oasisopen.sca.annotation;
2857
2858 import static java.lang.annotation.ElementType.FIELD;
2859 import static java.lang.annotation.ElementType.METHOD;
2860 import static java.lang.annotation.ElementType.PARAMETER;
2861 import static java.lang.annotation.ElementType.TYPE;
2862 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2863
2864 import java.lang.annotation.Retention;
2865 import java.lang.annotation.Target;
2866
2867 @Target({TYPE, FIELD, METHOD, PARAMETER})
2868 @Retention(RUNTIME)
2869 public @interface PolicySets {
2870     /**
2871      * Returns the policy sets to be applied.
2872      *
2873      * @return the policy sets to be applied
2874      */
2875     String[] value() default "";
2876 }
```

2877 *Figure 10-21: PolicySets Annotation*

2878

2879 The `@PolicySets` annotation is used to attach one or more SCA Policy Sets to a Java implementation
2880 class or to one of its subelements.

2881 See the [section "Policy Set Annotations"](#) for details and samples.

2882 **10.22 @Property**

2883 Figure 10-22 defines the `@Property` annotation:

2884

```

2885 package org.oasisopen.sca.annotation;
2886
2887 import static java.lang.annotation.ElementType.FIELD;
2888 import static java.lang.annotation.ElementType.METHOD;
2889 import static java.lang.annotation.ElementType.PARAMETER;
2890 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2891 import java.lang.annotation.Retention;
2892 import java.lang.annotation.Target;
2893
2894 @Target({METHOD, FIELD, PARAMETER})
2895 @Retention(RUNTIME)
2896 public @interface Property {
2897
2898     String name() default "";
2899     boolean required() default true;
2900 }

```

Figure 10-22: Property Annotation

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

When the Java type of a field, setter method or constructor parameter with the @Property annotation is a primitive type or a JAXB annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition into an instance of the Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled. [JCA90061]

When the Java type of a field, setter method or constructor parameter with the @Property annotation is not a JAXB annotated class, the SCA runtime can use any XML to Java mapping when converting property values into instances of the Java type.

The @Property annotation MUST NOT be used on a class field that is declared as final. [JCA90011]

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

The @Property annotation has the attributes:

- **name (0..1)** – the name of the property. For a field annotation, the default is the name of the field of the Java class. For a setter method annotation, the default is the JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a @Property annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present. [JCA90013]
- **required (0..1)** – a boolean value which specifies whether injection of the property value is required or not, where true means injection is required and false means injection is not required. Defaults to true. For a @Property annotation applied to a constructor parameter, the required attribute MUST NOT have the value false. [JCA90014]

Snippet 10-13 shows a property field definition sample.

```

2928 @Property(name="currency", required=true)
2929 protected String currency;
2930
2931 The following snippet shows a property setter sample
2932
2933 @Property(name="currency", required=true)
2934 public void setCurrency( String theCurrency ) {
2935     ....
2936 }

```

Snippet 10-13: Use of @Property on a Field

2938

2939 For a `@Property` annotation, if the type of the Java class field or the type of the input parameter of the
2940 setter method or constructor is defined as an array or as any type that extends or implements
2941 `java.util.Collection`, then the SCA runtime MUST introspect the component type of the implementation
2942 with a `<property/>` element with a `@many` attribute set to true, otherwise `@many` MUST be set to false.
2943 [JCA90047]

2944 Snippet 10-14 shows the definition of a configuration property using the `@Property` annotation for a
2945 collection.

```
2946 ...  
2947 private List<String> helloConfigurationProperty;  
2948  
2949 @Property(required=true)  
2950 public void setHelloConfigurationProperty(List<String> property) {  
2951     helloConfigurationProperty = property;  
2952 }  
2953 ...
```

2954 *Snippet 10-14: Use of @Property with a Collection*

2955 10.23 @Qualifier

2956 Figure 10-23 defines the `@Qualifier` annotation:

2957

```
2958 package org.oasisopen.sca.annotation;  
2959  
2960 import static java.lang.annotation.ElementType.METHOD;  
2961 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2962  
2963 import java.lang.annotation.Retention;  
2964 import java.lang.annotation.Target;  
2965  
2966 @Target(METHOD)  
2967 @Retention(RUNTIME)  
2968 public @interface Qualifier {  
2969 }
```

2970 *Figure 10-23: Qualifier Annotation*

2971

2972 The `@Qualifier` annotation is applied to an attribute of a specific intent annotation definition, defined using
2973 the `@Intent` annotation, to indicate that the attribute provides qualifiers for the intent. The `@Qualifier`
2974 annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.
2975 [JCA90015]

2976 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to define new
2977 intent annotations.

2978 10.24 @Reference

2979 Figure 10-24 defines the `@Reference` annotation:

2980

```
2981 package org.oasisopen.sca.annotation;  
2982  
2983 import static java.lang.annotation.ElementType.FIELD;  
2984 import static java.lang.annotation.ElementType.METHOD;  
2985 import static java.lang.annotation.ElementType.PARAMETER;  
2986 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2987 import java.lang.annotation.Retention;
```

```

2988 import java.lang.annotation.Target;
2989 @Target({METHOD, FIELD, PARAMETER})
2990 @Retention(RUNTIME)
2991 public @interface Reference {
2992
2993     String name() default "";
2994     boolean required() default true;
2995 }

```

Figure 10-24: Reference Annotation

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

The @Reference annotation MUST NOT be used on a class field that is declared as final. [JCA90016]

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

The @Reference annotation has the attributes:

- name : String (0..1)** – the name of the reference. For a field annotation, the default is the name of the field of the Java class. For a setter method annotation, the default is the JavaBeans property name corresponding to the setter method name. For a @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present. [JCA90018]
- required (0..1)** – a boolean value which specifies whether injection of the service reference is required or not, where true means injection is required and false means injection is not required. Defaults to true. For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true. [JCA90019]

Snippet 10-15 shows a reference field definition sample.

```

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

```

Snippet 10-15: Use of @Reference on a Field

Snippet 10-16 shows a reference setter sample

```

@Reference(name="stockQuote", required=true)
public void setStockQuote( StockQuoteService theSQService ) {
    ...
}

```

Snippet 10-16: Use of @Reference on a Setter

Snippet 10-17 shows a sample of a service reference using the @Reference annotation. The name of the reference is "helloService" and its type is HelloService. The clientMethod() calls the "hello" operation of the service referenced by the helloService reference.

```

package services.hello;

private HelloService helloService;

@Reference(name="helloService", required=true)
public setHelloService(HelloService service) {

```

```

3038     helloService = service;
3039 }
3040
3041 public void clientMethod() {
3042     String result = helloService.hello("Hello World!");
3043     ...
3044 }

```

3045 *Snippet 10-17: Use of @Reference and a ServiceReference*

3046

3047 The presence of a @Reference annotation is reflected in the componentType information that the runtime
3048 generates through reflection on the implementation class. Snippet 10-18 shows the component type for
3049 the component implementation fragment in Snippet 10-17.

3050

```

3051 <?xml version="1.0" encoding="ASCII"?>
3052 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
3053
3054     <!-- Any services offered by the component would be listed here -->
3055     <reference name="helloService" multiplicity="1..1">
3056         <interface.java interface="services.hello.HelloService"/>
3057     </reference>
3058
3059 </componentType>

```

3060 *Snippet 10-18: Implied componentType for Implementation in Snippet 10-17*

3061

3062 If the type of a reference is not an array or any type that extends or implements java.util.Collection, then
3063 the SCA runtime MUST introspect the component type of the implementation with a <reference/> element
3064 with @multiplicity= 0..1 if the @Reference annotation required attribute is false and with
3065 @multiplicity=1..1 if the @Reference annotation required attribute is true. [JCA90020]

3066 If the type of a reference is defined as an array or as any type that extends or implements
3067 java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation
3068 with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is
3069 false and with @multiplicity=1..n if the @Reference annotation required attribute is true. [JCA90021]

3070 Snippet 10-19 shows a sample of a service reference definition using the @Reference annotation on a
3071 java.util.List. The name of the reference is "helloServices" and its type is HelloService. The clientMethod()
3072 calls the "hello" operation of all the services referenced by the helloServices reference. In this case, at
3073 least one HelloService needs to be present, so **required** is true.

3074

```

3075 @Reference(name="helloServices", required=true)
3076 protected List<HelloService> helloServices;
3077
3078 public void clientMethod() {
3079
3080     ...
3081     for (int index = 0; index < helloServices.size(); index++) {
3082         HelloService helloService =
3083         (HelloService)helloServices.get(index);
3084         String result = helloService.hello("Hello World!");
3085     }
3086     ...
3087 }

```

3088 *Snippet 10-19: Use of @Reference with a List of ServiceReferences*

3089

3090 Snippet 10-20 shows the XML representation of the component type reflected from for the former
3091 component implementation fragment. There is no need to author this component type in this case since it
3092 can be reflected from the Java class.

3093

```
3094 <?xml version="1.0" encoding="ASCII"?>  
3095 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">  
3096  
3097     <!-- Any services offered by the component would be listed here -->  
3098     <reference name="helloServices" multiplicity="1..n">  
3099         <interface.java interface="services.hello.HelloService"/>  
3100     </reference>  
3101  
3102 </componentType>
```

3103 *Snippet 10-20: Implied componentType for Implementation in Snippet 10-19*

3104

3105 An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the
3106 SCA runtime as null [JCA90022] An unwired reference with a multiplicity of 0..n MUST be presented to
3107 the implementation code by the SCA runtime as an empty array or empty collection [JCA90023]

3108 10.24.1 Reinjection

3109 References MAY be reinjected by an SCA runtime after the initial creation of a component if the reference
3110 target changes due to a change in wiring that has occurred since the component was initialized.
3111 [JCA90024]

3112 In order for reinjection to occur, the following MUST be true:

- 3113 1. The component MUST NOT be STATELESS scoped.
- 3114 2. The reference MUST use either field-based injection or setter injection. References that are
3115 injected through constructor injection MUST NOT be changed.

3116 [JCA90025]

3117 Setter injection allows for code in the setter method to perform processing in reaction to a change.

3118 If a reference target changes and the reference is not reinjected, the reference MUST continue to work as
3119 if the reference target was not changed. [JCA90026]

3120 If an operation is called on a reference where the target of that reference has been undeployed, the SCA
3121 runtime SHOULD throw an InvalidServiceException. [JCA90027] If an operation is called on a reference
3122 where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD
3123 throw a ServiceUnavailableException. [JCA90028] If the target service of the reference is changed, the
3124 reference MUST either continue to work or throw an InvalidServiceException when it is invoked.
3125 [JCA90029] If it doesn't work, the exception thrown will depend on the runtime and the cause of the
3126 failure.

3127 A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds
3128 to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the
3129 ServiceReference obtained from the original reference MUST continue to work as if the reference target
3130 was not changed. [JCA90030] If the target of a ServiceReference has been undeployed, the SCA runtime
3131 SHOULD throw a InvalidServiceException when an operation is invoked on the ServiceReference.
3132 [JCA90031] If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD
3133 throw a ServiceUnavailableException when an operation is invoked on the ServiceReference.
3134 [JCA90032] If the target service of a ServiceReference is changed, the reference MUST either continue
3135 to work or throw an InvalidServiceException when it is invoked. [JCA90033] If it doesn't work, the
3136 exception thrown will depend on the runtime and the cause of the failure.

3137 A reference or ServiceReference accessed through the component context by calling getService() or
3138 getServiceReference() MUST correspond to the current configuration of the domain. This applies whether
3139 or not reinjection has taken place. [JCA90034] If the target of a reference or ServiceReference accessed
3140 through the component context by calling getService() or getServiceReference() has been undeployed or

3141 has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
 3142 and attempts to call business methods SHOULD throw an InvalidServiceException or a
 3143 ServiceUnavailableException. [JCA90035] If the target service of a reference or ServiceReference
 3144 accessed through the component context by calling getService() or getServiceReference() has changed,
 3145 the returned value SHOULD be a reference to the changed service. [JCA90036]

3146 The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This means
 3147 that in the cases where reference reinjection is not allowed, the array or Collection for a reference of
 3148 multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference
 3149 wiring or to the targets of the wiring. [JCA90037] In cases where the contents of a reference array or
 3150 collection change when the wiring changes or the targets change, then for references that use setter
 3151 injection, the setter method MUST be called by the SCA runtime for any change to the contents.
 3152 [JCA90038] A reinjected array or Collection for a reference MUST NOT be the same array or Collection
 3153 object previously injected to the component. [JCA90039]

3154

<u>Change event</u>	<u>Effect on</u>		
	Injected Reference or ServiceReference	Existing ServiceReference Object**	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	can be reinjected (if other conditions* apply). If not reinjected, then it continues to work as if the reference target was not changed.	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 throw InvalidServiceException.	Business methods throw InvalidServiceException.	Result is a reference to the undeployed service. Business methods throw InvalidServiceException.
Target service becomes unavailable	Business methods throw ServiceUnavailableException	Business methods throw ServiceUnavailableException	Result is be a reference to the unavailable service. Business methods throw ServiceUnavailableException.
Target service changed	might 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 failure.	might 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 failure.	Result is a reference to the changed service.
<p>* Other conditions: The component cannot be STATELESS scoped. The reference has to use either field-based injection or setter injection. References that are injected through constructor injection cannot be changed.</p> <p>** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().</p>			

3155 Table 10-1Reinjection Effects

3156 10.25 @Remotable

3157 Figure 10-25 defines the **@Remotable** annotation:

3158

```
3159 package org.oasisopen.sca.annotation;
3160
3161 import static java.lang.annotation.ElementType.TYPE;
3162 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3163 import java.lang.annotation.Retention;
3164 import java.lang.annotation.Target;
3165
3166 @Target(TYPE)
3167 @Retention(RUNTIME)
3168 public @interface Remotable {
3169 }
3170
3171 }
```

3172 *Figure 10-25: Remotable Annotation*

3173

3174 The @Remotable annotation is used to indicate that an SCA service interface is remotable. The
3175 @Remotable annotation is valid only on a Java interface, a Java class, a field, a setter method, or a
3176 constructor parameter. It MUST NOT appear anywhere else. [JCA90053] A remotable service can be
3177 published externally as a service and MUST be translatable into a WSDL portType. [JCA90040]

3178 The @Remotable annotation has no attributes. When placed on a Java service interface, it indicates that
3179 the interface is remotable. When placed on a Java service implementation class, it indicates that all SCA
3180 service interfaces provided by the class (including the class itself, if the class defines an SCA service
3181 interface) are remotable. When placed on a service reference, it indicates that the interface for the
3182 reference is remotable.

3183 Snippet 10-21 shows the Java interface for a remotable service with its @Remotable annotation.

3184

```
3185 package services.hello;
3186
3187 import org.oasisopen.sca.annotation.*;
3188
3189 @Remotable
3190 public interface HelloService {
3191     String hello(String message);
3192 }
3193 }
```

3194 *Snippet 10-21: Use of @Remotable on an Interface*

3195

3196 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
3197 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

3198 Complex data types exchanged via remotable service interfaces need to be compatible with the
3199 marshalling technology used by the service binding. For example, if the service is going to be exposed
3200 using the standard Web Service binding, then the parameters can be JAXB [JAX-B] types or they can be
3201 Service Data Objects (SDOs) [SDO].

3202 Independent of whether the remotable service is called from outside of the composite that contains it or
3203 from another component in the same composite, the data exchange semantics are **by-value**.

3204 Implementations of remotable services can modify input data during or after an invocation and can modify
3205 return data after the invocation. If a remotable service is called locally or remotely, the SCA container is
3206 responsible for making sure that no modification of input data or post-invocation modifications to return
3207 data are seen by the caller.

3208 Snippet 10-22 shows how a Java service implementation class can use the `@Remotable` annotation to
3209 define a remotable SCA service interface using a Java service interface that is not marked as remotable.

3210

```
3211 package services.hello;  
3212  
3213 import org.oasisopen.sca.annotation.*;  
3214  
3215 public interface HelloService {  
3216     String hello(String message);  
3217 }  
3218  
3219  
3220 package services.hello;  
3221  
3222 import org.oasisopen.sca.annotation.*;  
3223  
3224 @Remotable  
3225 @Service(HelloService.class)  
3226 public class HelloServiceImpl implements HelloService {  
3227  
3228     public String hello(String message) {  
3229         ...  
3230     }  
3231 }
```

3232 *Snippet 10-22: Use of @Remotable on a Class*

3233

3234 Snippet 10-23 shows how a reference can use the `@Remotable` annotation to define a remotable SCA
3235 service interface using a Java service interface that is not marked as remotable.

3236

```
3237 package services.hello;  
3238  
3239 import org.oasisopen.sca.annotation.*;  
3240  
3241 public interface HelloService {  
3242     String hello(String message);  
3243 }  
3244  
3245  
3246 package services.hello;  
3247  
3248 import org.oasisopen.sca.annotation.*;  
3249  
3250 public class HelloClient {  
3251  
3252     @Remotable  
3253     @Reference  
3254     protected HelloService myHello;  
3255  
3256     public String greeting(String message) {  
3257         return myHello.hello(message);  
3258     }  
3259 }
```

3260 *Snippet 10-23: Use of @Remotable on a Reference*

3261 **10.26 @Requires**

3262 Figure 10-26 defines the **@Requires** annotation:

3263

```

3264 package org.oasisopen.sca.annotation;
3265
3266 import static java.lang.annotation.ElementType.FIELD;
3267 import static java.lang.annotation.ElementType.METHOD;
3268 import static java.lang.annotation.ElementType.PARAMETER;
3269 import static java.lang.annotation.ElementType.TYPE;
3270 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3271
3272 import java.lang.annotation.Inherited;
3273 import java.lang.annotation.Retention;
3274 import java.lang.annotation.Target;
3275
3276 @Inherited
3277 @Retention(RUNTIME)
3278 @Target({TYPE, METHOD, FIELD, PARAMETER})
3279 public @interface Requires {
3280     /**
3281      * Returns the attached intents.
3282      *
3283      * @return the attached intents
3284      */
3285     String[] value() default "";
3286 }

```

Figure 10-26: Requires Annotation

The **@Requires** annotation supports general purpose intents specified as strings. Users can also define specific intent annotations using the @Intent annotation.

See the [section "General Intent Annotations"](#) for details and samples.

10.27 @Scope

Figure 10-27 defines the **@Scope** annotation:

```

3295 package org.oasisopen.sca.annotation;
3296
3297 import static java.lang.annotation.ElementType.TYPE;
3298 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3299 import java.lang.annotation.Retention;
3300 import java.lang.annotation.Target;
3301
3302 @Target(TYPE)
3303 @Retention(RUNTIME)
3304 public @interface Scope {
3305
3306     String value() default "STATELESS";
3307 }

```

Figure 10-27: Scope Annotation

The **@Scope** annotation **MUST** only be used on a service's implementation class. It is an error to use this annotation on an interface. [JCA90041]

The **@Scope** annotation has the attribute:

- **value** – the name of the scope.
- SCA defines the following scope names, but others can be defined by particular Java-based implementation types

3316 STATELESS
3317 COMPOSITE

3318 The default value is STATELESS.

3319 Snippet 10-24 shows a sample for a COMPOSITE scoped service implementation:

3320

```
3321 package services.hello;  
3322  
3323 import org.oasisopen.sca.annotation.*;  
3324  
3325 @Service(HelloService.class)  
3326 @Scope("COMPOSITE")  
3327 public class HelloServiceImpl implements HelloService {  
3328  
3329     public String hello(String message) {  
3330         ...  
3331     }  
3332 }
```

3333 *Snippet 10-24: Use of @Scope*

3334 10.28 @Service

3335 Figure 10-28 defines the @Service annotation:

3336

```
3337 package org.oasisopen.sca.annotation;  
3338  
3339 import static java.lang.annotation.ElementType.TYPE;  
3340 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
3341 import java.lang.annotation.Retention;  
3342 import java.lang.annotation.Target;  
3343  
3344 @Target(TYPE)  
3345 @Retention(RUNTIME)  
3346 public @interface Service {  
3347  
3348     Class<?>[] value();  
3349     String[] names() default {};  
3350 }
```

3351 *Figure 10-28: Service Annotation*

3352

3353 The @Service annotation is used on a component implementation class to specify the SCA services
3354 offered by the implementation. An implementation class need not be declared as implementing all of the
3355 interfaces implied by the services declared in its @Service annotation, but all methods of all the declared
3356 service interfaces MUST be present. [JCA90042] A class used as the implementation of a service is not
3357 required to have a @Service annotation. If a class has no @Service annotation, then the rules
3358 determining which services are offered and what interfaces those services have are determined by the
3359 specific implementation type.

3360 The @Service annotation has the attributes:

- 3361
- 3362 • **value (1..1)** – An array of interface or class objects that are exposed as services by this implementation. If the array is empty, no services are exposed.
 - 3363 • **names (0..1)** - An array of Strings which are used as the service names for each of the interfaces
3364 declared in the **value** array. The number of Strings in the names attribute array of the @Service
3365 annotation MUST match the number of elements in the value attribute array. [JCA90050] The value of

3366 each element in the @Service names array MUST be unique amongst all the other element values in
3367 the array. [JCA90060]

3368 The **service name** of an exposed service defaults to the name of its interface or class, without the
3369 package name. If the names attribute is specified, the service name for each interface or class in the
3370 value attribute array is the String declared in the corresponding position in the names attribute array.

3371 If a component implementation has two services with the same Java simple name, the names attribute of
3372 the @Service annotation MUST be specified. [JCA90045] If a Java implementation needs to realize two
3373 services with the same Java simple name then this can be achieved through subclassing of the interface.

3374 Snippet 10-25 shows an implementation of the HelloService marked with the @Service annotation.

3375

```
3376 package services.hello;  
3377  
3378 import org.oasisopen.sca.annotation.Service;  
3379  
3380 @Service(HelloService.class)  
3381 public class HelloServiceImpl implements HelloService {  
3382  
3383     public void hello(String name) {  
3384         System.out.println("Hello " + name);  
3385     }  
3386 }
```

3387 *Snippet 10-25: Use of @Service*

3388 11 WSDL to Java and Java to WSDL

3389 This specification applies the WSDL to Java and Java to WSDL mapping rules as defined by [the JAX-WS](#)
 3390 [2.1 specification \[JAX-WS\]](#) for generating remotable Java interfaces from WSDL portTypes and vice
 3391 versa.

3392 **SCA runtimes MUST support the JAX-WS 2.1 mappings from WSDL to Java and from Java to WSDL.**
 3393 **[JCA100022]** For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime **MUST** treat a
 3394 Java interface as if it had a `@WebService` annotation on the class, even if it doesn't. **[JCA100001]** The
 3395 SCA runtime **MUST** treat an `@org.oasisopen.sca.annotation.OneWay` annotation as a synonym for the
 3396 `@javax.jws.OneWay` annotation. **[JCA100002]** For the WSDL-to-Java mapping, the SCA runtime **MUST**
 3397 take the generated `@WebService` annotation to imply that the Java interface is `@Remotable`.
 3398 **[JCA100003]**

3399 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B] mapping
 3400 and the SDO 2.1 [SDO] mapping. **SCA runtimes MUST** support the JAXB 2.1 mapping from XML Schema
 3401 to Java and from Java to XML Schema. **[JCA100004]** SCA runtimes **MAY** support the SDO 2.1 mapping
 3402 from XML schema types to Java and from Java to XML Schema. **[JCA100005]** Having a choice of binding
 3403 technologies is allowed, as noted in the first paragraph of section 5 of the JSR 181 (version 2)
 3404 specification, which is referenced by the JAX-WS specification.

3405 11.1 JAX-WS Annotations and SCA Interfaces

3406 A Java class or interface used to define an SCA interface can contain JAX-WS annotations. In addition to
 3407 affecting the Java to WSDL mapping defined by [the JAX-WS specification \[JAX-WS\]](#) these annotations
 3408 can impact the SCA interface. **An SCA runtime MUST** apply the JAX-WS annotations as described in
 3409 [Table 11-1 and Table 11-2](#) when introspecting a Java class or interface class. **[JCA100011]** This could
 3410 mean that the interface of a Java implementation is defined by a WSDL interface declaration.

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
<code>@WebService</code>		A Java interface or class annotated with <code>@WebService</code> MUST be treated as if annotated with the SCA <code>@Remotable</code> annotation [JCA100012]
	name	If used to define a service, sets service name
	targetNamespace	None
	serviceName	None
	wsdlLocation	A Java class annotated with the <code>@WebService</code> annotation with its <code>wsdlLocation</code> attribute set MUST have its interface defined by the referenced WSDL definition instead of the annotated Java class.

[JCA100013]

endpointInterface A Java class annotated with the `@WebService` annotation with its `endpointInterface` attribute set **MUST** have its interface defined by the referenced interface instead of annotated Java class. [JCA100014]

`portName` None

@WebMethod

operationName Sets operation name

`action` None

exclude Method is excluded from the interface.

@OneWay

The SCA runtime **MUST** treat an `@org.oasisopen.sca.annotation.OneWay` annotation as a synonym for the `@javax.jws.OneWay` annotation.

[JCA100002]

@WebParam

name Sets parameter name

`targetNamespace` None

mode Sets directionality of parameter

header A Java class or interface containing an `@WebParam` annotation with its `header` attribute set to "true" **MUST** be treated as if the SOAP intent is applied to the Java class or interface. [JCA100015]

partName Overrides name

@WebResult

name	Sets parameter name
targetNamespace	None
header	A Java class or interface containing an @WebResult annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100016]
partName	Overrides name

@SOAPBinding

	A Java class or interface containing an @SOAPBinding annotation MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100021]
style	
use	
parameterStyle	

@HandlerChain

	None
file	
name	

3411 Table 11-1: JSR 181 Annotations and SCA Interfaces

3412

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
-------------------	-----------------	--------------------------------

@ServiceMode		A Java class containing an @ServiceMode annotation MUST be treated as if the SOAP intent is applied to the Java class. [JCA100017]
--------------	--	--

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
	value	
@WebFault	name	Sets fault name
	targetNamespace	None
	faultBean	None
@RequestWrapper		None
	localName	
	targetNamespace	
	className	
@ResponseWrapper		None
	localName	
	targetNamespace	
	className	
@WebServiceClient		An interface or class annotated with @WebServiceClient MUST NOT be used to define an SCA interface. [JCA100018]
	name	
	targetNamespace	
	wSDLLocation	
@WebEndpoint		None

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
	name	
@WebServiceProvider		A class annotated with @WebServiceProvider MUST be treated as if annotated with the SCA @Remotable annotation. [JCA100019]
	wSDLLocation	A Java class annotated with the @WebServiceProvider annotation with its wSDLLocation attribute set MUST have its interface defined by the referenced WSDL definition is used instead of the annotated Java class. [JCA100020]
	serviceName	None
	portName	None
	targetNamespace	None
@BindingType		None
	value	
@WebServiceRef		See JEE specification
	name	
	wSDLLocation	
	type	
	value	
	mappedName	
@WebServiceRefs		See JEE specification

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
	value	
@Action		None
	fault	
	input	
	output	
@FaultAction		None
	value	
	output	

3413 *Table 11-2: JSR 224 Annotations and SCA Interfaces*

3414 **11.2 JAX-WS Client Asynchronous API for a Synchronous Service**

3415 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client
3416 application with a means of invoking that service asynchronously, so that the client can invoke a service
3417 operation and proceed to do other work without waiting for the service operation to complete its
3418 processing. The client application can retrieve the results of the service either through a polling
3419 mechanism or via a callback method which is invoked when the operation completes.

3420 For SCA service interfaces defined using `interface.java`, the Java interface MUST NOT contain the
3421 additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100006] For
3422 SCA reference interfaces defined using `interface.java`, the SCA runtime MUST support a Java interface
3423 which contains the additional client-side asynchronous polling and callback methods defined by JAX-WS.
3424 [JCA100007] If the additional client-side asynchronous polling and callback methods defined by JAX-WS
3425 are present in the interface which declares the type of a reference in the implementation, SCA Runtimes
3426 MUST NOT include these methods in the SCA reference interface in the component type of the
3427 implementation. [JCA100008]

3428 The additional client-side asynchronous polling and callback methods defined by JAX-WS are recognized
3429 in a Java interface according to the steps:

3430 For each method M in the interface, if another method P in the interface has

- 3431 a. a method name that is M's method name with the characters "Async" appended, and
- 3432 b. the same parameter signature as M, and
- 3433 c. a return type of `Response<R>` where R is the return type of M

3434 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

3435 For each method M in the interface, if another method C in the interface has

- 3436 a. a method name that is M's method name with the characters "Async" appended, and
- 3437 b. a parameter signature that is M's parameter signature with an additional final parameter of
3438 type `AsyncHandler<R>` where R is the return type of M, and
- 3439 c. a return type of `Future<?>`

3440 then C is a JAX-WS callback method that isn't part of the SCA interface contract.
3441 As an example, an interface can be defined in WSDL as shown in Snippet 11-1:

3442

```
3443 <!-- WSDL extract -->
3444 <message name="getPrice">
3445   <part name="ticker" type="xsd:string"/>
3446 </message>
3447
3448 <message name="getPriceResponse">
3449   <part name="price" type="xsd:float"/>
3450 </message>
3451
3452 <portType name="StockQuote">
3453   <operation name="getPrice">
3454     <input message="tns:getPrice"/>
3455     <output message="tns:getPriceResponse"/>
3456   </operation>
3457 </portType>
```

3458 *Snippet 11-1: Example WSDL Interface*

3459

3460 The JAX-WS asynchronous mapping will produce the Java interface in Snippet 11-2:

3461

```
3462 // asynchronous mapping
3463 @WebService
3464 public interface StockQuote {
3465   float getPrice(String ticker);
3466   Response<Float> getPriceAsync(String ticker);
3467   Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
3468 }
```

3469 *Snippet 11-2: JAX-WS Asynchronous Interface for WSDL Interface in Snippet 11-1*

3470

3471 For SCA interface definition purposes, this is treated as equivalent to the interface in Snippet 11-3:

3472

```
3473 // synchronous mapping
3474 @WebService
3475 public interface StockQuote {
3476   float getPrice(String ticker);
3477 }
```

3478 *Snippet 11-3: Equivalent SCA Interface Corresponding to Java Interface in Snippet 11-2*

3479

3480 **SCA runtimes MUST support the use of the JAX-WS client asynchronous model.** [JCA100009] If the
3481 client implementation uses the asynchronous form of the interface, the two additional getPriceAsync()
3482 methods can be used for polling and callbacks as defined by the JAX-WS specification.

3483 **11.3 Treatment of SCA Asynchronous Service API**

3484 **For SCA service interfaces defined using interface.java, the SCA runtime MUST support a Java interface**
3485 **which contains the server-side asynchronous methods defined by SCA.** [JCA100010]

3486 Asynchronous service methods are identified as described in the section "Asynchronous handling of Long
3487 Running Service Operations" and are mapped to WSDL in the same way as the equivalent synchronous
3488 method described in that section.

3489 Generating an asynchronous service method from a WSDL request/response operation follows the
3490 algorithm described in the same section.

3491 12 Conformance

3492 The XML schema pointed to by the RDDDL document at the namespace URI, defined by this specification,
3493 are considered to be authoritative and take precedence over the XML schema defined in the appendix of
3494 this document.

3495 Normative code artifacts related to this specification are considered to be authoritative and take
3496 precedence over specification text.

3497 There are three categories of artifacts for which this specification defines conformance:

- 3498 a) SCA Java XML Document,
- 3499 b) SCA Java Class
- 3500 c) SCA Runtime.

3501 12.1 SCA Java XML Document

3502 An SCA Java XML document is an SCA Composite Document, or an SCA ComponentType Document,
3503 as defined by the [SCA Assembly Model specification \[ASSEMBLY\]](#), that uses the <interface.java>
3504 element. Such an SCA Java XML document MUST be a conformant SCA Composite Document or SCA
3505 ComponentType Document, as defined by the [SCA Assembly Model specification \[ASSEMBLY\]](#), and
3506 MUST comply with the requirements specified in [the Interface section](#) of this specification.

3507 12.2 SCA Java Class

3508 An SCA Java Class is a Java class or interface that complies with Java Standard Edition version 5.0 and
3509 MAY include annotations and APIs defined in this specification. An SCA Java Class that uses annotations
3510 and APIs defined in this specification MUST comply with the requirements specified in this specification
3511 for those annotations and APIs.

3512 12.3 SCA Runtime

3513 The APIs and annotations defined in this specification are meant to be used by Java-based component
3514 implementation models in either partial or complete fashion. A Java-based component implementation
3515 specification that uses this specification specifies which of the APIs and annotations defined here are
3516 used. The APIs and annotations an SCA Runtime has to support depends on which Java-based
3517 component implementation specification the runtime supports. For example, see the [SCA POJO
3518 Component Implementation Specification \[JAVA_CI\]](#).

3519 An implementation that claims to conform to this specification MUST meet the following conditions:

- 3520 1. The implementation MUST meet all the conformance requirements defined by the SCA Assembly
3521 Model Specification [ASSEMBLY].
- 3522 2. The implementation MUST support <interface.java> and MUST comply with all the normative
3523 statements in Section 3.
- 3524 3. The implementation MUST reject an SCA Java XML Document that does not conform to the sca-
3525 interface-java.xsd schema.
- 3526 4. The implementation MUST support and comply with all the normative statements in Section 10.

A. XML Schema: sca-interface-java-1.1.xsd

```
3528 <?xml version="1.0" encoding="UTF-8"?>
3529 <!-- Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
3530 OASIS trademark, IPR and other policies apply. -->
3531 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3532 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3533 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3534 elementFormDefault="qualified">
3535
3536 <include schemaLocation="sca-core-1.1-cd05.xsd"/>
3537
3538 <!-- Java Interface -->
3539 <element name="interface.java" type="sca:JavaInterface"
3540 substitutionGroup="sca:interface"/>
3541 <complexType name="JavaInterface">
3542 <complexContent>
3543 <extension base="sca:Interface">
3544 <sequence>
3545 <any namespace="##other" processContents="lax" minOccurs="0"
3546 maxOccurs="unbounded"/>
3547 </sequence>
3548 <attribute name="interface" type="NCName" use="required"/>
3549 <attribute name="callbackInterface" type="NCName"
3550 use="optional"/>
3551 </extension>
3552 </complexContent>
3553 </complexType>
3554
3555 </schema>
```

3556 B. Java Classes and Interfaces

3557 B.1 SCAClient Classes and Interfaces

3558 B.1.1 SCAClientFactory Class

3559 SCA provides an abstract base class SCAClientFactory. Vendors can provide subclasses of this class
3560 which create objects that implement the SCAClientFactory class suitable for linking to services in their
3561 SCA runtime.

3562

```
3563 /*  
3564  * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.  
3565  * OASIS trademark, IPR and other policies apply.  
3566  */  
3567 package org.oasisopen.sca.client;  
3568  
3569 import java.net.URI;  
3570 import java.util.Properties;  
3571  
3572 import org.oasisopen.sca.NoSuchDomainException;  
3573 import org.oasisopen.sca.NoSuchServiceException;  
3574 import org.oasisopen.sca.client.SCAClientFactoryFinder;  
3575 import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;  
3576  
3577 /**  
3578  * The SCAClientFactory can be used by non-SCA managed code to  
3579  * lookup services that exist in a SCADomain.  
3580  *  
3581  * @see SCAClientFactoryFinderImpl  
3582  * @see SCAClient  
3583  *  
3584  * @author OASIS Open  
3585  */  
3586  
3587 public abstract class SCAClientFactory {  
3588  
3589     /**  
3590      * The SCAClientFactoryFinder.  
3591      * Provides a means by which a provider of an SCAClientFactory  
3592      * implementation can inject a factory finder implementation into  
3593      * the abstract SCAClientFactory class - once this is done, future  
3594      * invocations of the SCAClientFactory use the injected factory  
3595      * finder to locate and return an instance of a subclass of  
3596      * SCAClientFactory.  
3597      */  
3598     protected static SCAClientFactoryFinder factoryFinder;  
3599     /**  
3600      * The Domain URI of the SCA Domain which is accessed by this  
3601      * SCAClientFactory  
3602      */  
3603     private URI domainURI;  
3604  
3605     /**  
3606      * Prevent concrete subclasses from using the no-arg constructor  
3607      */  
3608     private SCAClientFactory() {  
3609     }  
3610  
3611     /**
```

```

3612     * Constructor used by concrete subclasses
3613     * @param domainURI - The Domain URI of the Domain accessed via this
3614     * SCAClientFactory
3615     */
3616     protected SCAClientFactory(URI domainURI) {
3617     throws NoSuchDomainException {
3618         this.domainURI = domainURI;
3619     }
3620
3621     /**
3622     * Gets the Domain URI of the Domain accessed via this SCAClientFactory
3623     * @return - the URI for the Domain
3624     */
3625     protected URI getDomainURI() {
3626         return domainURI;
3627     }
3628
3629
3630     /**
3631     * Creates a new instance of the SCAClient that can be
3632     * used to lookup SCA Services.
3633     *
3634     * @param domainURI      URI of the target domain for the SCAClient
3635     * @return A new SCAClient
3636     */
3637     public static SCAClientFactory newInstance( URI domainURI )
3638         throws NoSuchDomainException {
3639         return newInstance(null, null, domainURI);
3640     }
3641
3642     /**
3643     * Creates a new instance of the SCAClient that can be
3644     * used to lookup SCA Services.
3645     *
3646     * @param properties    Properties that may be used when
3647     * creating a new instance of the SCAClient
3648     * @param domainURI      URI of the target domain for the SCAClient
3649     * @return A new SCAClient instance
3650     */
3651     public static SCAClientFactory newInstance(Properties properties,
3652                                               URI domainURI)
3653         throws NoSuchDomainException {
3654         return newInstance(properties, null, domainURI);
3655     }
3656
3657     /**
3658     * Creates a new instance of the SCAClient that can be
3659     * used to lookup SCA Services.
3660     *
3661     * @param classLoader    ClassLoader that may be used when
3662     * creating a new instance of the SCAClient
3663     * @param domainURI      URI of the target domain for the SCAClient
3664     * @return A new SCAClient instance
3665     */
3666     public static SCAClientFactory newInstance(ClassLoader classLoader,
3667                                               URI domainURI)
3668         throws NoSuchDomainException {
3669         return newInstance(null, classLoader, domainURI);
3670     }
3671
3672     /**
3673     * Creates a new instance of the SCAClient that can be
3674     * used to lookup SCA Services.
3675     *

```

```

3676 * @param properties      Properties that may be used when
3677 * creating a new instance of the SCAClient
3678 * @param classLoader    ClassLoader that may be used when
3679 * creating a new instance of the SCAClient
3680 * @param domainURI      URI of the target domain for the SCAClient
3681 * @return A new SCAClient instance
3682 */
3683 public static SCAClientFactory newInstance(Properties properties,
3684                                           ClassLoader classLoader,
3685                                           URI domainURI)
3686     throws NoSuchDomainException {
3687     final SCAClientFactoryFinder finder =
3688         factoryFinder != null ? factoryFinder :
3689         new SCAClientFactoryFinderImpl();
3690     final SCAClientFactory factory
3691         = finder.find(properties, classLoader, domainURI);
3692     return factory;
3693 }
3694
3695 /**
3696 * Returns a reference proxy that implements the business interface <T>
3697 * of a service in the SCA Domain handled by this SCAClientFactory
3698 *
3699 * @param serviceURI the relative URI of the target service. Takes the
3700 * form componentName/serviceName.
3701 * Can also take the extended form componentName/serviceName/bindingName
3702 * to use a specific binding of the target service
3703 *
3704 * @param interfaze The business interface class of the service in the
3705 * domain
3706 * @param <T> The business interface class of the service in the domain
3707 *
3708 * @return a proxy to the target service, in the specified SCA Domain
3709 * that implements the business interface <B>.
3710 * @throws NoSuchServiceException Service requested was not found
3711 * @throws NoSuchDomainException Domain requested was not found
3712 */
3713 public abstract <T> T getService(Class<T> interfaze, String serviceURI)
3714     throws NoSuchServiceException, NoSuchDomainException;
3715 }

```

3716 B.1.2 SCAClientFactoryFinder interface

3717 The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory
3718 finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can
3719 create alternative implementations of this interface that use different class loading or lookup mechanisms.

```

3720
3721 /*
3722 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
3723 * OASIS trademark, IPR and other policies apply.
3724 */
3725
3726 package org.oasisopen.sca.client;
3727
3728 import java.net.URI;
3729 import java.util.Properties;
3730
3731 import org.oasisopen.sca.NoSuchDomainException;
3732
3733 /* A Service Provider Interface representing a SCAClientFactory finder.
3734 * SCA provides a default reference implementation of this interface.
3735 * SCA runtime vendors can create alternative implementations of this
3736 * interface that use different class loading or lookup mechanisms.

```



```

3737 */
3738 public interface SCAClientFactoryFinder {
3739
3740     /**
3741     * Method for finding the SCAClientFactory for a given Domain URI using
3742     * a specified set of properties and a a specified ClassLoader
3743     * @param properties - properties to use - may be null
3744     * @param classLoader - ClassLoader to use - may be null
3745     * @param domainURI - the Domain URI - must be a valid SCA Domain URI
3746     * @return - the SCAClientFactory or null if the factory could not be
3747     * @throws - NoSuchDomainException if the domainURI does not reference
3748     * a valid SCA Domain
3749     * found
3750     */
3751     SCAClientFactory find(Properties properties,
3752                           ClassLoader classLoader,
3753                           URI domainURI )
3754     throws NoSuchDomainException ;
3755 }

```

3756 **B.1.3 SCAClientFactoryFinderImpl class**

3757 This class provides a default implementation for finding a provider's SCAClientFactory implementation
3758 class. It is used if the provider does not inject its SCAClientFactoryFinder implementation class into the
3759 base SCAClientFactory class.

3760 It discovers a provider's SCAClientFactory implementation by referring to the following information in this
3761 order:

- 3762 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the
3763 newInstance() method call if specified
- 3764 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 3765 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

```

3766
3767 /*
3768 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
3769 * OASIS trademark, IPR and other policies apply.
3770 */
3771 package org.oasisopen.sca.client.impl;
3772
3773 import org.oasisopen.sca.client.SCAClientFactoryFinder;
3774
3775 import java.io.BufferedReader;
3776 import java.io.Closeable;
3777 import java.io.IOException;
3778 import java.io.InputStream;
3779 import java.io.InputStreamReader;
3780 import java.lang.reflect.Constructor;
3781 import java.net.URI;
3782 import java.net.URL;
3783 import java.util.Properties;
3784
3785 import org.oasisopen.sca.NoSuchDomainException;
3786 import org.oasisopen.sca.ServiceRuntimeException;
3787 import org.oasisopen.sca.client.SCAClientFactory;
3788
3789 /**
3790 * This is a default implementation of an SCAClientFactoryFinder which is
3791 * used to find an implementation of the SCAClientFactory interface.
3792 *
3793 * @see SCAClientFactoryFinder
3794 * @see SCAClientFactory
3795 *

```

```

3796 * @author OASIS Open
3797 */
3798 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {
3799
3800     /**
3801      * The name of the System Property used to determine the SPI
3802      * implementation to use for the SCAClientFactory.
3803      */
3804     private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
3805         SCAClientFactory.class.getName();
3806
3807     /**
3808      * The name of the file loaded from the ClassPath to determine
3809      * the SPI implementation to use for the SCAClientFactory.
3810      */
3811     private static final String SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
3812         = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
3813
3814     /**
3815      * Public Constructor
3816      */
3817     public SCAClientFactoryFinderImpl() {
3818     }
3819
3820     /**
3821      * Creates an instance of the SCAClientFactorySPI implementation.
3822      * This discovers the SCAClientFactorySPI Implementation and instantiates
3823      * the provider's implementation.
3824      *
3825      * @param properties Properties that may be used when creating a new
3826      * instance of the SCAClient
3827      * @param classLoader ClassLoader that may be used when creating a new
3828      * instance of the SCAClient
3829      * @return new instance of the SCAClientFactory
3830      * @throws ServiceRuntimeException Failed to create SCAClientFactory
3831      * Implementation.
3832      */
3833     public SCAClientFactory find(Properties properties,
3834                                 ClassLoader classLoader,
3835                                 URI domainURI )
3836         throws NoSuchDomainException, ServiceRuntimeException {
3837         if (classLoader == null) {
3838             classLoader = getThreadContextClassLoader ();
3839         }
3840         final String factoryImplClassName =
3841             discoverProviderFactoryImplClass(properties, classLoader);
3842         final Class<? extends SCAClientFactory> factoryImplClass
3843             = loadProviderFactoryClass(factoryImplClassName,
3844                                     classLoader);
3845         final SCAClientFactory factory =
3846             instantiateSCAClientFactoryClass(factoryImplClass,
3847                                             domainURI );
3848         return factory;
3849     }
3850
3851     /**
3852      * Gets the Context ClassLoader for the current Thread.
3853      *
3854      * @return The Context ClassLoader for the current Thread.
3855      */
3856     private static ClassLoader getThreadContextClassLoader () {
3857         final ClassLoader threadClassLoader =
3858             Thread.currentThread().getContextClassLoader();
3859         return threadClassLoader;

```

```

3860     }
3861
3862     /**
3863     * Attempts to discover the class name for the SCAClientFactorySPI
3864     * implementation from the specified Properties, the System Properties
3865     * or the specified ClassLoader.
3866     *
3867     * @return The class name of the SCAClientFactorySPI implementation
3868     * @throw ServiceRuntimeException Failed to find implementation for
3869     * SCAClientFactorySPI.
3870     */
3871     private static String
3872     discoverProviderFactoryImplClass(Properties properties,
3873                                     ClassLoader classLoader)
3874     throws ServiceRuntimeException {
3875     String providerClassName =
3876         checkPropertiesForSPIClassName(properties);
3877     if (providerClassName != null) {
3878         return providerClassName;
3879     }
3880
3881     providerClassName =
3882         checkPropertiesForSPIClassName(System.getProperties());
3883     if (providerClassName != null) {
3884         return providerClassName;
3885     }
3886
3887     providerClassName = checkMETA-INFServicesForSIPClassName(classLoader);
3888     if (providerClassName == null) {
3889         throw new ServiceRuntimeException(
3890             "Failed to find implementation for SCAClientFactory");
3891     }
3892
3893     return providerClassName;
3894 }
3895
3896 /**
3897 * Attempts to find the class name for the SCAClientFactorySPI
3898 * implementation from the specified Properties.
3899 *
3900 * @return The class name for the SCAClientFactorySPI implementation
3901 * or <code>null</code> if not found.
3902 */
3903 private static String
3904 checkPropertiesForSPIClassName(Properties properties) {
3905     if (properties == null) {
3906         return null;
3907     }
3908
3909     final String providerClassName =
3910         properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
3911     if (providerClassName != null && providerClassName.length() > 0) {
3912         return providerClassName;
3913     }
3914
3915     return null;
3916 }
3917
3918 /**
3919 * Attempts to find the class name for the SCAClientFactorySPI
3920 * implementation from the META-INF/services directory
3921 *
3922 * @return The class name for the SCAClientFactorySPI implementation or
3923 * <code>null</code> if not found.

```

```

3924     */
3925 private static String checkMETAINFServicesForSIPClassName(ClassLoader cl)
3926 {
3927     final URL url =
3928         cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
3929     if (url == null) {
3930         return null;
3931     }
3932
3933     InputStream in = null;
3934     try {
3935         in = url.openStream();
3936         BufferedReader reader = null;
3937         try {
3938             reader =
3939                 new BufferedReader(new InputStreamReader(in, "UTF-8"));
3940
3941             String line;
3942             while ((line = readNextLine(reader)) != null) {
3943                 if (!line.startsWith("#") && line.length() > 0) {
3944                     return line;
3945                 }
3946             }
3947
3948             return null;
3949         } finally {
3950             closeStream(reader);
3951         }
3952     } catch (IOException ex) {
3953         throw new ServiceRuntimeException(
3954             "Failed to discover SCAClientFactory provider", ex);
3955     } finally {
3956         closeStream(in);
3957     }
3958 }
3959
3960 /**
3961  * Reads the next line from the reader and returns the trimmed version
3962  * of that line
3963  *
3964  * @param reader The reader from which to read the next line
3965  * @return The trimmed next line or <code>null</code> if the end of the
3966  * stream has been reached
3967  * @throws IOException I/O error occurred while reading from Reader
3968  */
3969 private static String readNextLine(BufferedReader reader)
3970     throws IOException {
3971
3972     String line = reader.readLine();
3973     if (line != null) {
3974         line = line.trim();
3975     }
3976     return line;
3977 }
3978
3979 /**
3980  * Loads the specified SCAClientFactory Implementation class.
3981  *
3982  * @param factoryImplClassName The name of the SCAClientFactory
3983  * Implementation class to load
3984  * @return The specified SCAClientFactory Implementation class
3985  * @throws ServiceRuntimeException Failed to load the SCAClientFactory
3986  * Implementation class
3987  */

```

```

3988 private static Class<? extends SCAClientFactory>
3989     loadProviderFactoryClass(String factoryImplClassName,
3990                             ClassLoader classLoader)
3991     throws ServiceRuntimeException {
3992
3993     try {
3994         final Class<?> providerClass =
3995             classLoader.loadClass(factoryImplClassName);
3996         final Class<? extends SCAClientFactory> providerFactoryClass =
3997             providerClass.asSubclass(SCAClientFactory.class);
3998         return providerFactoryClass;
3999     } catch (ClassNotFoundException ex) {
4000         throw new ServiceRuntimeException(
4001             "Failed to load SCAClientFactory implementation class "
4002             + factoryImplClassName, ex);
4003     } catch (ClassCastException ex) {
4004         throw new ServiceRuntimeException(
4005             "Loaded SCAClientFactory implementation class "
4006             + factoryImplClassName
4007             + " is not a subclass of "
4008             + SCAClientFactory.class.getName() , ex);
4009     }
4010 }
4011
4012 /**
4013  * Instantiate an instance of the specified SCAClientFactorySPI
4014  * Implementation class.
4015  *
4016  * @param factoryImplClass The SCAClientFactorySPI Implementation
4017  * class to instantiate.
4018  * @return An instance of the SCAClientFactorySPI Implementation class
4019  * @throws ServiceRuntimeException Failed to instantiate the specified
4020  * specified SCAClientFactorySPI Implementation class
4021  */
4022 private static SCAClientFactory instantiateSCAClientFactoryClass(
4023     Class<? extends SCAClientFactory> factoryImplClass,
4024     URI domainURI)
4025     throws NoSuchDomainException, ServiceRuntimeException {
4026
4027     try {
4028         Constructor<? extends SCAClientFactory> URIConstructor =
4029             factoryImplClass.getConstructor(domainURI.getClass());
4030         SCAClientFactory provider =
4031             URIConstructor.newInstance( domainURI );
4032         return provider;
4033     } catch (Throwable ex) {
4034         throw new ServiceRuntimeException(
4035             "Failed to instantiate SCAClientFactory implementation class "
4036             + factoryImplClass, ex);
4037     }
4038 }
4039
4040 /**
4041  * Utility method for closing Closeable Object.
4042  *
4043  * @param closeable The Object to close.
4044  */
4045 private static void closeStream(Closeable closeable) {
4046     if (closeable != null) {
4047         try{
4048             closeable.close();
4049         } catch (IOException ex) {
4050             throw new ServiceRuntimeException("Failed to close stream",
4051                 ex);

```

4052
4053
4054
4055

```
    }  
  }  
}
```

4056

B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?

4057
4058
4059

The SCAClient classes and interfaces are designed so that vendors can provide their own implementation suited to the needs of their SCA runtime. This section describes the tasks that a vendor needs to consider in relation to the SCAClient classes and interfaces.

4060

- Implement their SCAClientFactory implementation class

4061
4062
4063
4064

Vendors need to provide a subclass of SCAClientFactory that is capable of looking up Services in their SCA Runtime. Vendors need to subclass SCAClientFactory and implement the getService() method so that it creates reference proxies to services in SCA Domains handled by their SCA runtime(s).

4065

- Configure the Vendor SCAClientFactory implementation class so that it gets used

4066

Vendors have several options:

4067

Option 1: Set System Property to point to the Vendor's implementation

4068
4069

Vendors set the org.oasisopen.sca.client.SCAClientFactory System Property to point to their implementation class and use the reference implementation of SCAClientFactoryFinder

4070

Option 2: Provide a META-INF/services file

4071
4072

Vendors provide a META-INF/services/org.oasisopen.sca.client.SCAClientFactory file that points to their implementation class and use the reference implementation of SCAClientFactoryFinder

4073

Option 3: Inject a vendor implementation of the SCAClientFactoryFinder interface into

4074

SCAClientFactory

4075
4076
4077
4078
4079

Vendors inject an instance of the vendor implementation of SCAClientFactoryFinder into the factoryFinder field of the SCAClientFactory abstract class. The reference implementation of SCAClientFactoryFinder is not used in this scenario. The vendor implementation of SCAClientFactoryFinder can find the vendor implementation(s) of SCAClientFactory by any means.

4080

C. Conformance Items

4081 This section contains a list of conformance items for the SCA-J Common Annotations and APIs
4082 specification.

4083

Conformance ID	Description
[JCA20001]	Remotable Services MUST NOT make use of <i>method overloading</i> .
[JCA20002]	the SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time.
[JCA20003]	within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method.
[JCA20004]	Where an implementation is used by a "domain level component", and the implementation is marked "Composite" scope, the SCA runtime MUST ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation.
[JCA20005]	When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.
[JCA20006]	If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.
[JCA20007]	the SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT perform any synchronization.
[JCA20008]	Where an implementation is marked "Composite" scope and it is used by a component that is nested inside a composite that is used as the implementation of a higher level component, the SCA runtime MUST ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation. There can be multiple instances of the higher level component, each running on different nodes in a distributed SCA runtime.
[JCA20009]	The SCA runtime MAY use by-reference semantics when passing input parameters, return values or exceptions on calls to remotable services within the same JVM if both the service method implementation and the service proxy used by the client are marked "allows pass by reference".
[JCA20010]	The SCA runtime MUST use by-value semantics when passing input parameters, return values and exceptions on calls to remotable services within the same JVM if the service method implementation is not marked "allows pass by reference" or the service proxy used by the client is not marked "allows pass by reference".
[JCA30001]	The value of the @interface attribute MUST be the fully qualified name of the Java interface class
[JCA30002]	The value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used for callbacks
[JCA30003]	if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class.

- [JCA30004] The interface.java element MUST conform to the schema defined in the sca-interface-java.xsd schema.
- [JCA30005] The value of the @remotable attribute on the <interface.java/> element does not override the presence of a @Remotable annotation on the interface class and so if the interface class contains a @Remotable annotation and the @remotable attribute has a value of "false", then the SCA Runtime MUST raise an error and MUST NOT run the component concerned.
- [JCA30006] A Java interface referenced by the @interface attribute of an <interface.java/> element MUST NOT contain any of the following SCA Java annotations:
@AllowsPassByReference, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service.
- [JCA30007] A Java interface referenced by the @callbackInterface attribute of an <interface.java/> element MUST NOT contain any of the following SCA Java annotations:
@AllowsPassByReference, @Callback, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service.
- [JCA30009] The SCA Assembly Model specification [ASSEMBLY] defines a number of criteria that need to be satisfied in order for two interfaces to be compatible or have a compatible superset or subset relationship. If these interfaces are both Java interfaces, compatibility also means that every method that is present in both interfaces is defined consistently in both interfaces with respect to the @OneWay annotation, that is, the annotation is either present in both interfaces or absent in both interfaces.
- [JCA30010] If the identified class is annotated with either the JAX-WS @WebService or @WebServiceProvider annotations and the annotation has a non-empty **wsdlLocation** property, then the SCA Runtime MUST act as if an <interface.wsdl/> element is present instead of the <interface.java/> element, with an @interface attribute identifying the portType mapped from the Java interface class and containing @requires and @policySets attribute values equal to the @requires and @policySets attribute values of the <interface.java/> element.
- [JCA40001] The SCA Runtime MUST call a constructor of the component implementation at the start of the Constructing state.
- [JCA40002] The SCA Runtime MUST perform any constructor reference or property injection when it calls the constructor of a component implementation.
- [JCA40003] When the constructor completes successfully, the SCA Runtime MUST transition the component implementation to the Injecting state.
- [JCA40004] If an exception is thrown whilst in the Constructing state, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40005] When a component implementation instance is in the Injecting state, the SCA Runtime MUST first inject all field and setter properties that are present into the component implementation.
- [JCA40006] When a component implementation instance is in the Injecting state, the SCA Runtime MUST inject all field and setter references that are present into the component implementation, after all the properties have been injected.
- [JCA40007] The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected properties and references are made visible to the component

implementation without requiring the component implementation developer to do any specific synchronization.

- [JCA40008] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation is in the Injecting state.
- [JCA40009] When the injection of properties and references completes successfully, the SCA Runtime MUST transition the component implementation to the Initializing state.
- [JCA40010] If an exception is thrown whilst injecting properties or references, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40011] When the component implementation enters the Initializing State, the SCA Runtime MUST call the method annotated with @Init on the component implementation, if present.
- [JCA40012] If a component implementation invokes an operation on an injected reference that refers to a target that has not yet been initialized, the SCA Runtime MUST throw a ServiceUnavailableException.
- [JCA40013] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Initializing state.
- [JCA40014] Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition the component implementation to the Running state.
- [JCA40015] If an exception is thrown whilst initializing, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40016] The SCA Runtime MUST invoke Service methods on a component implementation instance when the component implementation is in the Running state and a client invokes operations on a service offered by the component.
- [JCA40017] When the component implementation scope ends, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40018] When a component implementation enters the Destroying state, the SCA Runtime MUST call the method annotated with @Destroy on the component implementation, if present.
- [JCA40019] If a component implementation invokes an operation on an injected reference that refers to a target that has been destroyed, the SCA Runtime MUST throw an InvalidServiceException.
- [JCA40020] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Destroying state.
- [JCA40021] Once the method annotated with @Destroy completes successfully, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40022] If an exception is thrown whilst destroying, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40023] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Terminated state.
- [JCA40024] If a property or reference is unable to be injected, the SCA Runtime MUST transition the component implementation to the Destroying state.

- [JCA60001] When a bidirectional service is invoked, the SCA runtime MUST inject a callback reference for the invoking service into all fields and setter methods of the service implementation class that are marked with a @Callback annotation and typed by the callback interface of the bidirectional service, and the SCA runtime MUST inject null into all other fields and setter methods of the service implementation class that are marked with a @Callback annotation.
- [JCA60002] When a non-bidirectional service is invoked, the SCA runtime MUST inject null into all fields and setter methods of the service implementation class that are marked with a @Callback annotation.
- [JCA60003] The SCA asynchronous service Java interface mapping of a WSDL request-response operation MUST appear as follows:
- The interface is annotated with the "asyncInvocation" intent.
 - For each service operation in the WSDL, the Java interface contains an operation with
 - a name which is the JAX-WS mapping of the WSDL operation name, with the suffix "Async" added
 - a void return type
 - a set of input parameter(s) which match the JAX-WS mapping of the input parameter(s) of the WSDL operation plus an additional last parameter which is a ResponseDispatch object typed by the JAX-WS Response Bean mapping of the output parameter(s) of the WSDL operation, where ResponseDispatch is the type defined in the SCA Java Common Annotations and APIs specification.
- [JCA60004] An SCA Runtime MUST support the use of the SCA asynchronous service interface for the interface of an SCA service.
- [JCA60005] If the SCA asynchronous service interface ResponseDispatch handleResponse method is invoked more than once through either its sendResponse or its sendFault method, the SCA runtime MUST throw an IllegalStateException.
- [JCA60006] For the purposes of matching interfaces (when wiring between a reference and a service, or when using an implementation class by a component), an interface which has one or more methods which follow the SCA asynchronous service pattern MUST be treated as if those methods are mapped as the equivalent synchronous methods, as follows:
- Asynchronous service methods are characterized by:
 - void return type
 - a method name with the suffix "Async"
 - a last input parameter with a type of ResponseDispatch<X>
 - annotation with the asyncInvocation intent
 - possible annotation with the @AsyncFault annotation
- The mapping of each such method is as if the method had the return type "X", the method name without the suffix "Async" and all the input parameters except the last parameter of the type ResponseDispatch<X>, plus the list of exceptions contained in the @AsyncFault annotation.
- [JCA70001] SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific intent annotation.
- [JCA70002] Intent annotations MUST NOT be applied to the following:
- A method of a service implementation class, except for a setter method that is

either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification

- A service implementation class field that is not either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class constructor parameter that is not annotated with `@Reference`

[JCA70003] Where multiple intent annotations (general or specific) are applied to the same Java element, the SCA runtime MUST compute the combined intents for the Java element by merging the intents from all intent annotations on the Java element according to the SCA Policy Framework [POLICY] rules for merging intents at the same hierarchy level.

[JCA70004] If intent annotations are specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective intents for the method by merging the combined intents from the method with the combined intents for the interface according to the SCA Policy Framework [POLICY] rules for merging intents within a structural hierarchy, with the method at the lower level and the interface at the higher level.

[JCA70005] The `@PolicySets` annotation MUST NOT be applied to the following:

- A method of a service implementation class, except for a setter method that is either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class field that is not either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class constructor parameter that is not annotated with `@Reference`

[JCA70006] If the `@PolicySets` annotation is specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective policy sets for the method by merging the policy sets from the method with the policy sets from the interface.

[JCA80001] The `ComponentContext.getService` method MUST throw an `IllegalArgumentException` if the reference identified by the `referenceName` parameter has multiplicity of `0..n` or `1..n`.

[JCA80002] The `ComponentContext.getRequestContext` method MUST return non-null when invoked during the execution of a Java business method for a service operation or a callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases.

[JCA80003] When invoked during the execution of a service operation, the `RequestContext.getServiceReference` method MUST return a `ServiceReference` that represents the service that was invoked.

[JCA80004] The `ComponentContext.getServiceReference` method MUST throw an `IllegalArgumentException` if the reference named by the `referenceName` parameter has multiplicity greater than one.

[JCA80005] The `ComponentContext.getServiceReference` method MUST throw an `IllegalArgumentException` if the reference named by the `referenceName` parameter

- does not have an interface of the type defined by the businessInterface parameter.
- [JCA80006] The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if the component does not have a reference with the name provided in the referenceName parameter.
- [JCA80007] The ComponentContext.getServiceReference method MUST return null if the multiplicity of the reference named by the referenceName parameter is 0..1 and the reference has no target service configured.
- [JCA80008] The ComponentContext.getURI method MUST return the absolute URI of the component in the SCA Domain.
- [JCA80009] The ComponentContext.getService method MUST return the proxy object implementing the interface provided by the businessInterface parameter, for the reference named by the referenceName parameter with the interface defined by the businessInterface parameter when that reference has a target service configured.
- [JCA80010] The ComponentContext.getService method MUST return null if the multiplicity of the reference named by the referenceName parameter is 0..1 and the reference has no target service configured.
- [JCA80011] The ComponentContext.getService method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
- [JCA80012] The ComponentContext.getService method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.
- [JCA80013] The ComponentContext.getServiceReference method MUST return a ServiceReference object typed by the interface provided by the businessInterface parameter, for the reference named by the referenceName parameter with the interface defined by the businessInterface parameter when that reference has a target service configured.
- [JCA80014] The ComponentContext.getServices method MUST return a collection containing one proxy object implementing the interface provided by the businessInterface parameter for each of the target services configured on the reference identified by the referenceName parameter.
- [JCA80015] The ComponentContext.getServices method MUST return an empty collection if the service reference with the name supplied in the referenceName parameter is not wired to any target services.
- [JCA80016] The ComponentContext.getServices method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
- [JCA80017] The ComponentContext.getServices method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
- [JCA80018] The ComponentContext.getServices method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.
- [JCA80019] The ComponentContext.getServiceReferences method MUST return a collection containing one ServiceReference object typed by the interface provided by the

businessInterface parameter for each of the target services configured on the reference identified by the referenceName parameter.

- [JCA80020] The ComponentContext.getServiceReferences method MUST return an empty collection if the service reference with the name supplied in the referenceName parameter is not wired to any target services.
- [JCA80021] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
- [JCA80022] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
- [JCA80023] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.
- [JCA80024] The ComponentContext.createSelfReference method MUST return a ServiceReference object typed by the interface defined by the businessInterface parameter for one of the services of the invoking component which has the interface defined by the businessInterface parameter.
- [JCA80025] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the component does not have a service which implements the interface identified by the businessInterface parameter.
- [JCA80026] The ComponentContext.createSelfReference method MUST return a ServiceReference object typed by the interface defined by the businessInterface parameter for the service identified by the serviceName of the invoking component and which has the interface defined by the businessInterface parameter.
- [JCA80027] The ComponentContext.createSelfReference method MUST throw an IllegalArgumentException if the component does not have a service with the name identified by the serviceName parameter.
- [JCA80028] The ComponentContext.createSelfReference method MUST throw an IllegalArgumentException if the component service with the name identified by the serviceName parameter does not implement a business interface which is compatible with the supplied businessInterface parameter.
- [JCA80029] The ComponentContext.getProperty method MUST return an object of the type identified by the type parameter containing the value specified in the component configuration for the property named by the propertyName parameter or null if no value is specified in the configuration.
- [JCA80030] The ComponentContext.getProperty method MUST throw an IllegalArgumentException if the component does not have a property with the name identified by the propertyName parameter.
- [JCA80031] The ComponentContext.getProperty method MUST throw an IllegalArgumentException if the component property with the name identified by the propertyName parameter does not have a type which is compatible with the supplied type parameter.
- [JCA80032] The ComponentContext.cast method MUST return a ServiceReference object which is typed by the same business interface as specified by the reference proxy object supplied in the target parameter.

- [JCA80033] The ComponentContext.cast method MUST throw an IllegalArgumentException if the supplied target parameter is not an SCA reference proxy object.
- [JCA80034] The RequestContext.getSecuritySubject method MUST return the JAAS subject of the current request, or null if there is no subject or null if the method is invoked from code not processing a service request or callback request.
- [JCA80035] The RequestContext.getServiceName method MUST return the name of the service for which an operation is being processed, or null if invoked from a thread that is not processing a service operation or a callback operation.
- [JCA80036] The RequestContext.getCallbackReference method MUST return a ServiceReference object typed by the interface of the callback supplied by the client of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation.
- [JCA80037] The RequestContext.getCallback method MUST return a reference proxy object typed by the interface of the callback supplied by the client of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation.
- [JCA80038] When invoked during the execution of a callback operation, the RequestContext.getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.
- [JCA80039] When invoked from a thread not involved in the execution of either a service operation or of a callback operation, the RequestContext.getServiceReference method MUST return null.
- [JCA80040] The ServiceReference.getService method MUST return a reference proxy object which can be used to invoke operations on the target service of the reference and which is typed with the business interface of the reference.
- [JCA80041] The ServiceReference.getBusinessInterface method MUST return a Class object representing the business interface of the reference.
- [JCA80042] The SCAClientFactory.newInstance(URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80043] The SCAClientFactory.newInstance(URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80044] The SCAClientFactory.newInstance(Properties, URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80045] The SCAClientFactory.newInstance(Properties, URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80046] The SCAClientFactory.newInstance(Classloader, URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80047] The SCAClientFactory.newInstance(Classloader, URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80048] The SCAClientFactory.newInstance(Properties, Classloader, URI) method MUST

- return an object which implements the `SCAClientFactory` class for the SCA Domain identified by the `domainURI` parameter.
- [JCA80049] The `SCAClientFactory.newInstance(Properties, Classloader, URI)` MUST throw a `NoSuchDomainException` if the `domainURI` parameter does not identify a valid SCA Domain.
- [JCA80050] The `SCAClientFactory.getService` method MUST return a proxy object which implements the business interface defined by the `interfaze` parameter and which can be used to invoke operations on the service identified by the `serviceURI` parameter.
- [JCA80051] The `SCAClientFactory.getService` method MUST throw a `NoSuchServiceException` if a service with the relative URI `serviceURI` and a business interface which matches `interfaze` cannot be found in the SCA Domain targeted by the `SCAClient` object.
- [JCA80052] The `SCAClientFactory.getService` method MUST throw a `NoSuchServiceException` if the `domainURI` of the `SCAClientFactory` does not identify a valid SCA Domain.
- [JCA80053] The `SCAClientFactory.getDomainURI` method MUST return the SCA Domain URI of the Domain associated with the `SCAClientFactory` object.
- [JCA80054] The `SCAClientFactory.getDomainURI` method MUST throw a **`NoSuchServiceException`** if the `domainURI` of the `SCAClientFactory` does not identify a valid SCA Domain.
- [JCA80055] The implementation of the `SCAClientFactoryFinder.find` method MUST return an object which is an implementation of the `SCAClientFactory` interface, for the SCA Domain represented by the `doaminURI` parameter, using the supplied properties and classloader.
- [JCA80056] The implementation of the `SCAClientFactoryFinder.find` method MUST throw a `ServiceRuntimeException` if the `SCAClientFactory` implementation could not be found.
- [JCA50057] The `ResponseDispatch.sendResponse()` method MUST send the response message to the client of an asynchronous service.
- [JCA80058] The `ResponseDispatch.sendResponse()` method MUST throw an `InvalidStateException` if either the `sendResponse` method or the `sendFault` method has already been called once.
- [JCA80059] The `ResponseDispatch.sendFault()` method MUST send the supplied fault to the client of an asynchronous service.
- [JCA80060] The `ResponseDispatch.sendFault()` method MUST throw an `InvalidStateException` if either the `sendResponse` method or the `sendFault` method has already been called once.
- [JCA90001] An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code.
- [JCA90001] SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an SCA annotation on a static method or a static field of an implementation class and the SCA runtime MUST NOT instantiate such an implementation class.
- [JCA90003] If a constructor of an implementation class is annotated with `@Constructor` and the constructor has parameters, each of these parameters MUST have either a `@Property` annotation or a `@Reference` annotation.

- [JCA90004] A method annotated with `@Destroy` can have any access modifier and MUST have a void return type and no arguments.
- [JCA90005] If there is a method annotated with `@Destroy` that matches the criteria for the annotation, the SCA runtime MUST call the annotated method when the scope defined for the implementation class ends.
- [JCA90007] When marked for eager initialization with an `@EagerInit` annotation, the composite scoped instance MUST be created when its containing component is started.
- [JCA90008] A method marked with the `@Init` annotation can have any access modifier and MUST have a void return type and no arguments.
- [JCA90009] If there is a method annotated with `@Init` that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.
- [JCA90011] The `@Property` annotation MUST NOT be used on a class field that is declared as `final`.
- [JCA90013] For a `@Property` annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present.
- [JCA90014] For a `@Property` annotation applied to a constructor parameter, the required attribute MUST NOT have the value `false`.
- [JCA90015] The `@Qualifier` annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.
- [JCA90016] The `@Reference` annotation MUST NOT be used on a class field that is declared as `final`.
- [JCA90018] For a `@Reference` annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present.
- [JCA90019] For a `@Reference` annotation applied to a constructor parameter, the required attribute MUST have the value `true`.
- [JCA90020] If the type of a reference is not an array or any type that extends or implements `java.util.Collection`, then the SCA runtime MUST introspect the component type of the implementation with a `<reference/>` element with `@multiplicity=0..1` if the `@Reference` annotation required attribute is `false` and with `@multiplicity=1..1` if the `@Reference` annotation required attribute is `true`.
- [JCA90021] If the type of a reference is defined as an array or as any type that extends or implements `java.util.Collection`, then the SCA runtime MUST introspect the component type of the implementation with a `<reference/>` element with `@multiplicity=0..n` if the `@Reference` annotation required attribute is `false` and with `@multiplicity=1..n` if the `@Reference` annotation required attribute is `true`.
- [JCA90022] An unwired reference with a multiplicity of `0..1` MUST be presented to the implementation code by the SCA runtime as `null` (either via injection or via API call).
- [JCA90023] An unwired reference with a multiplicity of `0..n` MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection (either via injection or via API call).
- [JCA90024] References MAY be reinjected by an SCA runtime after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized.
- [JCA90025] In order for reinjection to occur, the following MUST be true:

1. The component MUST NOT be STATELESS scoped.
2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed.

- [JCA90026] If a reference target changes and the reference is not reinjected, the reference MUST continue to work as if the reference target was not changed.
- [JCA90027] If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException.
- [JCA90028] If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw a ServiceUnavailableException.
- [JCA90029] If the target service of the reference is changed, the reference MUST either continue to work or throw an InvalidServiceException when it is invoked.
- [JCA90030] A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the ServiceReference obtained from the original reference MUST continue to work as if the reference target was not changed.
- [JCA90031] If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an operation is invoked on the ServiceReference.
- [JCA90032] If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an operation is invoked on the ServiceReference.
- [JCA90033] If the target service of a ServiceReference is changed, the reference MUST either continue to work or throw an InvalidServiceException when it is invoked.
- [JCA90034] A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain. This applies whether or not reinjection has taken place.
- [JCA90035] If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.
- [JCA90036] If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.
- [JCA90037] In the cases where reference reinjection is not allowed, the array or Collection for a reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference wiring or to the targets of the wiring.
- [JCA90038] In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents.
- [JCA90039] A reinjected array or Collection for a reference MUST NOT be the same array or Collection object previously injected to the component.
- [JCA90040] A remotable service can be published externally as a service and MUST be

translatable into a WSDL portType.

- [JCA90041] The @Scope annotation MUST only be used on a service's implementation class. It is an error to use this annotation on an interface.
- [JCA90042] An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present.
- [JCA90045] If a component implementation has two services with the same Java simple name, the names attribute of the @Service annotation MUST be specified.
- [JCA90046] When used to annotate a method or a field of an implementation class for injection of a callback object, the @Callback annotation MUST NOT specify any attributes.
- [JCA90047] For a @Property annotation, if the type of the Java class field or the type of the input parameter of the setter method or constructor is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <property/> element with a @many attribute set to true, otherwise @many MUST be set to false.
- [JCA90050] The number of Strings in the names attribute array of the @Service annotation MUST match the number of elements in the value attribute array.
- [JCA90052] The @AllowsPassByReference annotation MUST only annotate the following locations:
- a service implementation class
 - an individual method of a remotable service implementation
 - an individual reference which uses a remotable interface, where the reference is a field, a setter method, or a constructor parameter
- [JCA90053] The @Remotable annotation is valid only on a Java interface, a Java class, a field, a setter method, or a constructor parameter. It MUST NOT appear anywhere else.
- [JCA90054] When used to annotate a method or a field of an implementation class for injection of a callback object, the type of the method or field MUST be the callback interface of at least one bidirectional service offered by the implementation class.
- [JCA90055] A method annotated with @OneWay MUST have a void return type and MUST NOT have declared checked exceptions.
- [JCA90056] When a method of a Java interface is annotated with @OneWay, the SCA runtime MUST ensure that all invocations of that method are executed in a non-blocking fashion, as described in the section on Asynchronous Programming.
- [JCA90057] The @Callback annotation MUST NOT appear on a setter method or a field of a Java implementation class that has COMPOSITE scope.
- [JCA90058] When used to annotate a setter method or a field of an implementation class for injection of a callback object, the SCA runtime MUST inject a callback reference proxy into that method or field when the Java class is initialized, if the component is invoked via a service which has a callback interface and where the type of the setter method or field corresponds to the type of the callback interface.
- [JCA90060] The value of each element in the @Service names array MUST be unique amongst all the other element values in the array.
- [JCA90061] When the Java type of a field, setter method or constructor parameter with the @Property annotation is a primitive type or a JAXB annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition

into an instance of the Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled.

- [JCA100001] For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.
- [JCA100002] The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.
- [JCA100003] For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.
- [JCA100004] SCA runtimes MUST support the JAXB 2.1 mapping from XML Schema to Java and from Java to XML Schema.
- [JCA100005] SCA runtimes MAY support the SDO 2.1 mapping from XML schema types to Java and from Java to XML Schema.
- [JCA100006] For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the additional client-side asynchronous polling and callback methods defined by JAX-WS.
- [JCA100007] For SCA reference interfaces defined using interface.java, the SCA runtime MUST support a Java interface which contains the additional client-side asynchronous polling and callback methods defined by JAX-WS.
- [JCA100008] If the additional client-side asynchronous polling and callback methods defined by JAX-WS are present in the interface which declares the type of a reference in the implementation, SCA Runtimes MUST NOT include these methods in the SCA reference interface in the component type of the implementation.
- [JCA100009] SCA runtimes MUST support the use of the JAX-WS client asynchronous model.
- [JCA100010] For SCA service interfaces defined using interface.java, the SCA runtime MUST support a Java interface which contains the server-side asynchronous methods defined by SCA.
- [JCA100011] An SCA runtime MUST apply the JAX-WS annotations as described in Table 11-1 and Table 11-2 when introspecting a Java class or interface class.
- [JCA100012] A Java interface or class annotated with @WebService MUST be treated as if annotated with the SCA @Remotable annotation
- [JCA100013] A Java class annotated with the @WebService annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition instead of the annotated Java class.
- [JCA100014] A Java class annotated with the @WebService annotation with its endpointInterface attribute set MUST have its interface defined by the referenced interface instead of annotated Java class.
- [JCA100015] A Java class or interface containing an @WebParam annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface.
- [JCA100016] A Java class or interface containing an @WebResult annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface.
- [JCA100017] A Java class containing an @ServiceMode annotation MUST be treated as if the SOAP intent is applied to the Java class.

- [JCA100018] An interface or class annotated with @WebServiceClient MUST NOT be used to define an SCA interface.
- [JCA100019] A class annotated with @WebServiceProvider MUST be treated as if annotated with the SCA @Remotable annotation.
- [JCA100020] A Java class annotated with the @WebServiceProvider annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition is used instead of the annotated Java class.
- [JCA100021] A Java class or interface containing an @SOAPBinding annotation MUST be treated as if the SOAP intent is applied to the Java class or interface.
- [JCA100022] SCA runtimes MUST support the JAX-WS 2.1 mappings from WSDL to Java and from Java to WSDL.

4084

4085

D. Acknowledgements

4086 The following individuals have participated in the creation of this specification and are gratefully
4087 acknowledged:

4088 **Participants:**

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

Scott Vorthmann
Feng Wang
Robin Yang

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

4089

E. Revision History

4090 [optional; should not be included in OASIS Standards]

4091

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	* 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	* 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
cd01-rev1	2008-12-11	Anish Karmarkar	* Fixed reference style to [RFC2119] instead of [1]. * Applied resolutions of issues 20, 21, 41, 42, 43, 47, 48, 49.
cd01-rev2	2008-12-12	Anish Karmarkar	* Applied resolutions of issues 61, 71, 72, 73, 79, 81, 82, 84, 112
cd01-rev3	2008-12-16	David Booz	* Applied resolution of issues 56, 75, 111
cd01-rev4	2009-01-18	Anish Karmarkar	* Applied resolutions of issues 28, 52, 94, 96, 99, 101
cd02	2009-01-26	Mike Edwards	Minor editorial cleanup.

			All changes accepted. All comments removed.
cd02-rev1	2009-02-03	Mike Edwards	Issues 25+95 Issue 120
cd02-rev2	2009-02-08	Mike Edwards	Merge annotation definitions contained in section 10 into section 8 Move remaining parts of section 10 to section 7. Accept all changes.
cd02-rev3	2009-03-16	Mike Edwards	Issue 104 - RFC2119 work and formal marking of all normative statements - all sections - Completion of Appendix B (list of all normative statements) Accept all changes
cd02-rev4	2009-03-20	Mike Edwards	Editorially removed sentence about componentType side files in Section1 Editorially changed package name to org.oasisopen from org.osoa in lines 291, 292 Issue 6 - add Section 2.3, modify section 9.1 Issue 30 - Section 2.2.2 Issue 76 - Section 6.2.4 Issue 27 - Section 7.6.2, 7.6.2.1 Issue 77 - Section 1.2 Issue 102 - Section 9.21 Issue 123 - conersations removed Issue 65 - Added a new Section 4 ** Causes renumbering of later sections ** ** NB new numbering is used below ** Issue 119 - Added a new section 12 Issue 125 - Section 3.1 Issue 130 - (new number) Section 8.6.2.1 Issue 132 - Section 1 Issue 133 - Section 10.15, Section 10.17 Issue 134 - Section 10.3, Section 10.18 Issue 135 - Section 10.21 Issue 138 - Section 11 Issue 141 - Section 9.1 Issue 142 - Section 10.17.1
cd02-rev5	2009-04-20	Mike Edwards	Issue 154 - Appendix A Issue 129 - Section 8.3.1.1
cd02-rev6	2009-04-28	Mike Edwards	Issue 148 - Section 3 Issue 98 - Section 8

cd02-rev7	2009-04-30	Mike Edwards	Editorial cleanup throughout the spec
cd02-rev8	2009-05-01	Mike Edwards	Further extensive editorial cleanup throughout the spec Issue 160 - Section 8.6.2 & 8.6.2.1 removed
cd02-rev8a	2009-05-03	Simon Nash	Minor editorial cleanup
cd03	2009-05-04	Anish Karmarkar	Updated references and front page clean up
cd03-rev1	2009-09-15	David Booz	Applied Issues: 1,13,125,131,156,157,158,159,161,165,172,177
cd03-rev2	2010-01-19	David Booz	Updated to current Assembly namespace Applied issues: 127,155,168,181,184,185,187,189,190,194
cd03-rev3	2010-02-01	Mike Edwards	Applied issue 54. Editorial updates to code samples.
cd03-rev4	2010-02-05	Bryan Aupperle, Dave Booz	Editorial update for OASIS formatting
CD04	2010-02-06	Dave Booz	Editorial updates for Committee Draft 04 All changes accepted

4092